# Table Of Contents

1. Introduction  
2. Cymbal/DSQL Quick  
3. Loading Data, Defining Metadata, And Running Queries  
4. Daytona’s DSQL  
5. Cymbal Classes, Variables, Functions, Predicates, Assertions And Descriptions  
6. Procedural Cymbal  
7. Various Built-in Functions, Predicates And Procedures  
8. I/O: Reading And Writing On Channels  
9. Declarative Cymbal  
10. Tokens  
11. Associative Arrays  
12. Boxes: More Than Just In-Memory Tables  
13. Advanced Topics For Cymbal Descriptions  
14. Aggregation In Cymbal  
15. Macro-Like Devices: Macro Predicates, Views, lambda Opcofuns, Apply  
16. Path Recursive Queries Including Transitive Closure  
17. Updates, Transactions, Logging And Recovery In Cymbal  
18. Modes Of Use  
19. Parallelization Foundations  
20. Parallelization Made Easy  
21. Shared Memory In Cymbal  
22. Networking And Distribution  
23. Record Class Descriptions: Special Topics  
24. Conclusion  

Appendix: DSQL Grammar
Chapter 1. Introduction
-- Introduction --
-- Unique Architecture --
-- Convenient Data Format --
-- Easily Handles Large Amounts Of Data --
-- Powerful Query Language --
-- Modes Of Use --
-- Summary Of Advantages --
-- What Is To Come --

Chapter 2. Cymbal/DSQL Quick
-- Quick Querying Of Relational Databases --
-- Finding Suppliers With Unfilled Orders --
-- Finding The Average Quantity Ordered Of Each Part --
-- Finding Parts Ordered From A Given Supplier --
-- The for_each_time Looping Construct --
-- More Procedural Cymbal --
-- Associative Arrays --
-- The for_each_time Looping Construct --
-- Transitive Closure --

Chapter 3. Loading Data, Defining Metadata, And Running Queries
-- UNIX Flat Files --
-- Representing Missing Values In DC Files --
-- Comments in DC files --
-- Check_DC_Lines --
-- Further Syntactic Features Of And Limitations On Data File Records --
-- Object Records --
-- Daytona Data Dictionary --
-- Project Descriptions --
-- Application Archives --
-- Application Descriptions --
-- Record Class Descriptions: rcds --
-- Testing Out rcds --
-- Naming Conventions --
-- Keys And Indices --
-- Cluster B-trees --
-- Other Kinds Of Keys And Indices --
-- Non-indexed Unique Keys --
-- Eliminating Unwanted Index Entries --
-- Banning Indices --
-- More Performance Considerations --
-- Space Efficiency --
-- Shell Environment --

Copyright © 2013 AT&T
All Rights Reserved.
September 15, 2013
Chapter 4. DSQL: SQL For Daytona

Chapter 5. Cymbal Classes, Variables, Functions, Assertions And Descriptions
-- Composite Classes --
-- SETS, BUNCHES, LISTS, TUPLES --
-- INTERVALS --
-- ARRAYS --
-- CHANNELS --
-- VBL VBLS: Pointers in Cymbal --
-- Cymbal Variables Extended --
-- Cymbal Functions --
-- Keyword Arguments --
-- Cymbal Assertions And Built-In Predicates --
-- Cymbal Satisfaction Claims --
-- Logical Connectives --
-- The truth Function --
-- Extended Predicates --
-- Quantifiers --
-- VblSpecs --
-- Free And Bound Variables --
-- Variable Scoping --
-- Implicit Quantification --
-- Cymbal Descriptions --
-- Special Kinds Of Simple Notes --
-- Further Note Extensions --

Chapter 6. Procedural Cymbal
-- Assignments --
-- The Assignment Function set --
-- Substitution of LHS in RHS --
-- Otherwise: Assignments That Can Fail --
-- Assignment Grammar --
-- Semicolons --
-- Grouping --
-- Conditionals --
-- Loops --
-- Conventional Loops --
-- For_Each_Time Loops --
-- For_The_First_Time Loops --
-- Control-Break Programming With For_Each_Time --
-- Branches --
-- Procedure Calls --
-- Program Structure --
-- Version Identification For Code --
-- Defining And Importing Variables --
-- Defining, Importing And Initializing Array Variables --
-- Variable Specifications --
-- Reading From Channels By Matching REs --
-- Missing Values In I/O Channels --
-- Detecting And Handling I/O Errors --
-- Reading UNIX Flat Files --
-- Read PROCEDURE --
-- Reading Binary Objects By Reading With Stated Sizes --
-- Error Handling For The Read PROCEDURE --
-- Reading conventional ARRAYS and TUPLES --
-- Read PROC Miscellany --
-- Bipipe Sorting Example --
-- _fifo_ CHANS --
-- Miscellaneous CHAN Fpps --

Chapter 9. Declarative Cymbal
-- Declarative Cymbal --
-- Declarative vs. Procedural Semantics --
-- Display --
-- Display Output Formats --
-- More Display Examples --
-- OPCONDS --
-- OPCONDS In Display Calls --
-- Protecting Display Calls From Outside Elements --
-- Putting Outside Constant Values In Every Output Tuple From A Display --
-- OPCONDS in for_each_time loops --
-- Display Keyword Arguments And Their Functionality --
-- Procedural Semantics For Satisfying OPCONDS --
-- Finitely Defining Variables --
-- Satisfying Simple Satisfaction Claim OPCONDS --
-- Satisfying Conjunction OPCONDS --
-- Satisfying Disjunction OPCONDS --
-- Satisfying If-Then-Else OPCONDS --
-- Satisfying Existential OPCONDS --
-- Duplicates In Satisfaction LISTS --
-- Varieties Of Test Assertions --
-- Automatic Caching Of Ground Assertion Results --
-- A Final Procedural Semantics Example: Joins --
-- The Nature Of Cymbal --

Chapter 10. Tokens
-- Procedural Tokenizing --
-- Declarative Tokenizing --
-- Different Ways To Tokenize --
-- Different Sources For Tokens --
-- Handling Missing Values For Delimiter-Based Tokens --
-- Procedural tokens Exception Handling --
-- Miscellaneous Options To tokens --

Chapter 11. Associative Arrays
-- Dynamic Associative Arrays --
-- Defining/Declaring Dynamic Associative Arrays --
-- Managing Huge Dynamic Associative Arrays In Minimal Space --
-- Creating/Updating/Deleting Elements Of Dynamic Associative Arrays --
-- Creating/Updating/Deleting TUPLE-based Elements --
-- Working With Dynamic Associative Arrays Declaratively --
-- Declaratively-Defined Dynamic Associative Arrays --
-- Static Associative Arrays --

Chapter 12. Boxes: More Than Just In-Memory Tables
-- Reducing SetFormers/ListFormers To box Calls --
-- Technical Digression: An Undesirable Alternative --
-- Outside Variables And Box-Formers --
-- Building Boxes --
-- Build_Box Keywords --
-- Box Ancillary Variables And Assertions --
-- Extended SetFormer/ListFormer Syntax --
-- Box Variables --
-- Explicitly Typing Box Variables --
-- Using Boxes --
-- Use_Box Keywords --
-- Skolems/Placeholders --
-- Using Box Indices For Searching --
-- BoxFormer Predicates & Hybrid Build/Use --
-- Is_Something_Where And Is_The_Next_Where --
-- Is_The_First_Where And Is_The_Last_Where --
-- Incremental Additions And Deletions To Boxes --
-- Is_In_Again For LIST additions --
-- Updating A Box Element By Deletion Followed By Insertion --
-- Deleting Box Elements At Given Positions --
-- Using A for_each_time Loop To Add/Delete/Update Some Box Elements --
-- Updating Box Elements In Place --
-- Caching And Recomputing The Values Of BOX VBLs --
-- Sorting By v10sort Instead Of Boxes --

Chapter 13. Advanced Topics For Cymbal Descriptions
-- Keys, Indices And Descriptions: --
-- Finding Out What Indices Daytona Has Chosen, If Any --
-- Hard-coding Index Choice --
-- Boxes Of Key Field Values --
-- Partial Match Indexed Retrieval: --
-- Starts_With: Indexing On Prefixes --
-- Indexed Range Queries --
-- Indexed Range Queries For UINT --
-- Indexed Subnet Queries --
-- Hash Joins --
-- Outer Joins --
-- Outer Joins: System-supplied Dummy Values --
-- Outer Joins: DSQL and Hash Joins --
-- Miscellaneous Cymbal Description Capabilities: --
-- Total Missing Value Control --
-- this_is_a --
-- Describe & Dump Output PROCEDURES --
-- Safe Dirty Reads --
-- Ignoring Failed Opens --
-- Descriptions Using using_source --
-- Accessing Records By Position: --
-- there_isa_bin_first, there_isa_bin_last
-- Using Descriptions To Access Records By Ordinal Position --
-- Using Descriptions To Access Records In A Cluster By Ordinal Position --
-- Sampling Records Using Descriptions --
-- there_is_a_next --
-- skipping with there_isa --

Chapter 14. Aggregation In Cymbal
-- Scalar Aggregate Functions With No Grouping --
-- There Is Something About count --
-- Nested Scalar Aggregate Functions With No Grouping --
-- TUPLES Of Aggregate Functions With No Grouping --
-- Aggregate Functions With Grouping: Initial Formulations --
-- Scalar Aggregate Functions With Grouping Via BOX-formers --
-- TUPLES Of Aggregate Functions With Grouping Via BOX-formers --
-- top-k Queries --
-- High-level top-k LISTS --
-- top-k Querying From Basic Concepts --
-- top-epsilon Queries --

Chapter 15. Views And Macro Predicates
-- Macro-Like Devices: Macro Predicates, Views, lambda Opcofuns, Apply --
-- Macro Predicates --
-- Handling Outside Variables In Macro Predicates --
-- Perspectives On Macro Predicates --
-- Perspectives On The English Monarchy Via Macro Predicates --
-- The Types Of Macro Predicates Do Matter --
-- Algorithms As Macro Predicates --
-- Ground_In_Use --
-- Views --
Chapter 16. Path Recursive Queries Including Transitive Closure

Chapter 17. Updates And Transactions In Cymbal
Case Study Of A Malformed Transaction --
Locking Issues --
Handling Deadlock --
Updates To Horizontally Partitioned Tables --
Exception Handling For Transactions --
File Descriptors And Transactions --
Caveats --
Logging Transactions and Using Recover --

Chapter 18. Modes Of Use
To The Database Or Not --
Utilizing Answers, Executables, And Code --
Daytona For Answers --
Daytona For Executables --
Code Synthesis: Daytona For Code --
Writing Cymbal For A Data Manager Library --
Compiling A Data Manager Library --
Code Synthesis Quality Recommendations --
Code Synthesis v. Embedded SQL, Modules, CLI, ODBC, JDBC, RDA, etc. --
Other Programming Language Database Interfaces --
Cymbal Packages --
Generating Adhoc Cymbal --

Chapter 19. Parallelization Foundations
BUNDLES Of TENDRILS --
TENDRIL Clones --
Dividing Up The Work With from_section --
Boxes Using from_section --
Descriptions Using from_section --
Horizontal Partitioning And from_section --
from_section in general --
Dividing Up Work By Clone Number --
A Simple Clone Query --
Synchronizing Output To _stdout_ With Messages --
Waiting For Child Processes To Report Exit Statuses --
Clones Writing To Shared File CHAN --
Parents Talking To Children --
Child Is Expecting Exactly One Parental Message --
Child Is Expecting Several Parental Messages --
Children Talking Back To Parents Once --
Clones Talking Back To Parents Constantly Helped By next_io_ready_tendril --
Parents And Clones Conversing Frequently --
Merging Sorted Output With Pipes Back To Parent --
Parents And Children Talking Via Funnels --
Synchronizing Using TICKET_BUNCHES --
Chapter 20. Parallelization Made Easy
-- Parallelizing Displays --
-- Parallelizing Displays: Sequential Access --
-- Parallelizing Displays: Indexed Access --
-- Parallelizing Displays: Partitioning Keys --
-- Parallelizing Displays: Using Hash Joins --
-- Parallelizing Displays: Path Preds --
-- Parallelizing Boxes --
-- Parallelizing for_each_time --
-- Early termination for parallelized queries --

Chapter 21. Shared Memory In Cymbal
-- Using Shared Memory In Cymbal --
-- Creating And Administering Shared Memory In Cymbal --
-- Installing The Shared Memory Infrastructure --
-- Special Utilities --
-- Producer/Consumer Using Non-Concurrent Shmem Dynara --
-- Shared Memory And Parallelization --
-- Shared Memory, Parallelization, Hash Joins, Views, And SQL Group-By --
-- Concurrent Access To Shared Memory Dynara --
-- Concurrent Deletes Of Shmem Dynara Elements --
-- Concurrent Inserts Of Shmem Dynara Elements --
-- Concurrent Reads Of Shmem Dynara Elements --
-- Concurrent do_final Updates Of Shmem Dynara Elements --
-- Concurrent do_critical Updates Of Shmem Dynara Elements --
-- Concurrent One-line Updates Of Shmem Dynara Elements --
-- How General/Useful Are Shmem Dynara? --
-- Conventional Dynara As Tables With Multiple Indices --
-- Non-Concurrent Shmem Dynara As Tables With Multiple Indices --
-- Concurrent Shmem Dynara As Tables With Multiple Indices --

Chapter 22. Networking And Distribution
-- Simple Network Programming With _fifo_ --
-- Socket-Based Client-Server Basics --
-- Where The Server Says Goodbye First --
-- Where The Client Says Goodbye First --
-- pdq Network Query Server --

Chapter 23. Advanced Topics For Record Class Descriptions
All About Daytona

Rick Greer
AT&T Laboratories

1. Introduction

The Daytona data management system offers a full spectrum of data management and programming language services. It has all the data management basics: a high-level query language (Cymbal®), data dictionary, indexing, updates, transactions, concurrency, and recovery.

1.1 Unique Architecture

Daytona’s unique low-overhead architecture makes it attractive for running on platforms ranging from small ones with 64MB of memory to large ones like a 7x24 production call detail data warehouse with its 64 CPUs, 767 GB of memory and 312 terabytes of data circa 2009. This architecture is characterized by Daytona’s query processing paradigm: Daytona translates its Cymbal and SQL query languages into C and then causes functions from its own database libraries to be linked in to create the corresponding executable. Consequently, Daytona gets from the operating system alone the kind of services that other DBMS get from their server processes, i.e., locking, file systems, access control, scheduling, caching, low-level networking, etc.

In other words, the server processes at the heart of traditional relational DBMS are in many ways implementations of yet another operating system. Since all modern computing platforms come with (sophisticated) operating systems, the ones provided by traditional DBMS are in principle redundant. They are huge masses of code which at best have to be extremely clever at being efficient in order to overcome the fact that a good portion of what they are offering is already being done by the underlying operating system. Since Daytona eliminates the middleman by not relying on local database server processes, it has a head start on being more efficient -- and more reliable: since it uses far less code, there is much less to go wrong. A corollary is that if your machine is up, Daytona is up (since there are no database server processes to start). Contrast this absence of local database system server processes with the sometimes dozens of processes underlying a traditional DBMS installation.

(Please note that Daytona users remain free to use Daytona to create their own application server processes, which are in fact necessary for distributed applications: the point being made here is that there are no Daytona system daemon processes that are handling local user queries and data loading; in
fact, currently there are no Daytona daemon processes at all except the one that is used for the JDBC and Perl database network interfaces, if those are being used.)

Another consequence of Daytona translating Cymbal completely into C is that queries run fast. Most of Daytona’s competitors interpret their SQL. Compiling a language into native machine code is widely considered to produce executables that run faster than the corresponding source can be interpreted. Over the past 20 years, when tested on queries for real applications, Daytona has almost always been proved to run several times faster than its competitors.

1.2 Convenient Data Format

Daytona is also unusual in that it can store its data in an extension to the common ASCII flat file format favored by programs like Perl and awk. Such records consist of delimiter-separated fields with new-line terminated records, with optional list and set values for fields. As a result, users can use common text editors to examine their data and a variety of UNIX tools to generate and modify it.

In short, the Daytona architecture is so simple, flexible, and open to UNIX tools that users can easily integrate Daytona functionality into their applications and environment.

1.3 Easily Handles Large Amounts Of Data

Furthermore, when space is at a premium, Daytona has special compression-oriented datatypes that use space-efficient codes to store field values. An additional layer of compression is available by using semi-adaptive dictionary coding to compress entire records at a time. In the case of call detail, Daytona is able to use both methods to store data in 1/5 the space that a certain competitor would. Indeed, the best way to store a terabyte of data is to store 200 GB instead. In fact, when the reduction in I/O time is greater than the required decompression time, then data compression is not only space-efficient but time-efficient as well.

Daytona’s horizontal partitioning feature is another key player in the task of managing terabytes well. Horizontal partitioning allows a single end-user table to be implemented by transparently storing its rows in an arbitrary number of UNIX files according to application-specific criteria. These files may be on different disks and, when using NFS, even on different machines. Such application-specific partitioning criteria might include the geographical source of the data, which time-interval it was generated in, and who generated it. This idea can be taken quite a ways: the call detail data warehouse partitions its largest call detail table into over 100,000 files containing over 1 trillion records as of 2009.

As to scalability in a general setting, were each of these 90,000 files to hold 30 gigabytes (which they don’t for the call detail data warehouse), then over 2.6 petabytes would be used to store one table. In terms of information, if that data were compressed at a factor of four to one, then the uncompressed volume of information would be around 10.3 petabytes. To date, Daytona has easily and successfully managed data on as much disk as its customers have had the money or the interest to buy.

1.4 Powerful Query Language

Daytona provides this production-quality database management by means of Cymbal, a very-high-level, multi-paradigm language. Cymbal synthesizes such procedural language constructs as assignments, conditionals, loops, and (procedural) functions with complementary declarative constructs taken from symbolic logic, set theory, Cymbal database descriptions, and SQL. In particular, the full
ANSI ’89 standard SQL data manipulation language is available either standalone or embedded in the
procedural part of Cymbal, as is customary with fourth generation languages (4GLs). Daytona’s SQL,
called DSQL, extends this ANSI ’89 SQL DML to support (the large variety of) Cymbal types, Cymbal
functions, and other convenient additions.

In particular, Cymbal contains many constructs that are simply not present in other vendor 4GLs.
These include generalized transitive closure, fully general quantifiers, set- and list- formers, scalar and
tuple-valued multidimensional integer-indexed and associative arrays, explicit lists and sets, intervals &
lattices, aggregate function capabilities more general than that of SQL, standard deviation and
correlation statistical aggregate functions, quantiles, a full set of parallelization primitives, UNIX
regular expression matching and lexical analysis, a uniform I/O interface to files, uni- & bi- pipes,
sockets, the command line, and strings (i.e., in-memory tables complete with indices), data buffer
reuse, aggregate-valued fields, user-definable missing value strategies, polymorphic functions with
automatic type coercion, packages, static type checking, simple type inference, and keyword function
arguments with optional defaults.

A fluent Cymbal programmer tackles a programming problem by freely mixing constructs in the
same query program from whichever paradigm is most appropriate for the immediate task at hand.
There can well be SQL nested inside procedural statements and alongside declarative expressions
taken from logic and set-theory. Furthermore, Cymbal can easily be extended to call the user’s own C
functions.

1.5 Modes Of Use

Daytona can be used in essentially 5 different ways. The first is the traditional write-a-query-
and-get-the-answers mode. The next three modes occur when the Daytona user is writing an end-user
application for someone else.

The second mode is for the Daytona user to write parameterized Cymbal queries and compile
them to executables which can then be called (repeatedly) with appropriate parameters from within the
application being written.

The third mode, the code synthesis mode, is employed when Cymbal is used to write the high-level
specification of a data management library for the end-user application. That Cymbal code is then
translated into C library routines for linking with the user’s other C code. Code synthesis employs a
Daytona sandwich wherein user code calls Daytona-generated code which in turn may call other user
code, the latter illustrating that Daytona is arbitrarily C user-extensible. This strategy employs the
fastest possible linkage between application and data management code: simple C function calls within
the same process. No pipes, sockets, servers, or shared memory are needed. This, in part, is why
Daytona is ideal for embedded applications.

The fourth mode consists of using a GUI to generate complete Cymbal queries in response to ad
hoc user interaction and then to have Daytona compile and run those newly generated queries to
produce the desired answers.

The fifth mode is completely different from the others. It consists of using a network client to
access Daytona’s pdq server which itself is simply serving as a network proxy that will execute
Daytona’s usual shell-level commands on behalf of the remote user. (Thus pdq is not at all equivalent
to the typical database server process.) *pdq* is most commonly used to implement Daytona’s JDBC and Perl DBI interfaces; alternatively, through a simple client-server message protocol, it can provide network access to Daytona databases for other (remote) programs that employ that protocol.

### 1.6 Summary Of Advantages

Daytona’s architecture and languages offer several advantages.

**Speed**
Daytona is several times faster than its competitors, which is primarily due to the elimination of the server middleman and to the compilation of queries into customized machine code (as opposed to general purpose interpreter pseudo-code).

**Capacity**
By using the flat file format, Daytona enables its users to access their Daytona data directly with all of their favorite UNIX tools. Its horizontal partitioning and data compression features enable users to easily manage as much data as they have disk to store it in.

**Query Power**
The Cymbal query language (which includes the SQL DML as a subset) is a very powerful 4GL that has been evolved to meet AT&T application needs.

**Simplicity**
The simple, no-client-server architecture implies correspondingly simple OA&M since there are no server processes to keep alive and healthy: if your computer is up, Daytona is up. Also, users can leverage their UNIX expertise to great advantage with Daytona since Daytona’s minimalist approach leaves as much to UNIX as possible.

**Reliability**
Since it contains so much less code than others that contain their own operating systems, there is just much less to go wrong with Daytona. Also, the 1300+ query test suite that comes with Daytona serves to ensure that each new Daytona release is strictly better than its predecessors.

**Flexibility**
Daytona’s open architecture invites synergistic use of UNIX tools. The several powerful programming paradigms utilized by Cymbal promote flexible approaches to writing queries that quickly get the intended answers.

### 1.7 What Is To Come

The remaining chapters of this user’s guide are intended to introduce the Cymbal query language and to give the prospective user enough information to be able to use Daytona on any system where it has been installed.

- **Chapter 2** gives a quick introduction to writing queries in Cymbal/DSQL for relational databases.
- **Chapter 3** describes how to use Daytona to load data, define metadata, and run queries in the user’s UNIX environment.
- **Chapter 4** gives details on the use of DSQL, i.e., Daytona’s SQL; this chapter emphasizes Daytona’s extensions to SQL and is brief since many standard texts already treat SQL in much greater detail.
Chapter 5 introduces Cymbal’s datatypes, variables, functions, predicates, assertions and
descriptions which are needed in both procedural and declarative Cymbal.
Chapter 6 discusses procedural Cymbal.
Chapter 7 discusses in detail a variety of Daytona’s built-in functions, predicates, and procedures
which are available for use in both Cymbal and SQL.
Chapter 8 discusses Cymbal’s constructs for reading and writing from different kinds of I/O
channels including files, strings, and pipes.
Chapter 9 discusses declarative Cymbal with emphasis on the symbolic logic part.
Chapter 10 discusses the Cymbal tokens() family of functions.
Chapter 11 discusses Cymbal scalar and tuple-valued multidimensional associative arrays.
Chapter 12 discusses Cymbal boxes which in many ways are like indexable in-memory tables of
tuples.
Chapter 13 discusses advanced features of Cymbal record class descriptions, i.e., metadata.
Chapter 14 continues with the advanced topic of declaratively-defined aggregate functions.
Chapter 15 discusses views, macro predicates, lambda-like expressions, and apply in Cymbal.
Chapter 16 discusses generalized transitive closure and path PREDs in Cymbal.
Chapter 17 discusses updates, transactions, logging and recovery in Cymbal.
Chapter 18 discusses 4 distinct ways to use Daytona locally plus Daytona’s remote/network access
functionality.
Chapter 19 discusses the full family of basic parallelization techniques in Cymbal.
Chapter 20 discusses the extreme power and simplicity of high-level Cymbal parallelization
techniques.
Chapter 21 discusses how to use shared memory in Cymbal programs.
Chapter 22 discusses Daytona in a network setting including sockets and distributed query
processing.
Chapter 23 introduces several special data dictionary capabilities, including horizontal partitioning
which uses multiple files to implement a single record class, i.e., table.
Chapter 24 is the conclusion.
Appendix DSQL Grammar
Appendix Cymbal Grammar
Appendix Data Dictionary Grammar
Please feel free to browse at will. In fact, a good strategy is to begin by reading the first part of every chapter. Please remember that this is a reference manual: the organizing principle for the chapters is to group together related material with the same prerequisites. For example, as much procedural-only material as possible is grouped together and presented before going into the declarative material. Consequently, reading each chapter in depth and great detail is not the best way to get started quickly. To get started quickly, a breadth-first exploration is preferable over the depth-first approach that would occur when reading each chapter in sequence.

Those who are interested in running queries right away may wish to scan Chapter 2 to get an idea of what Cymbal/DSQL queries are like and may then proceed immediately to the Getting Started With Daytona tutorial, which is a separate document. Then continue with the more extensive introduction provided by Daytona Basics.
2. Cymbal/DSQL Quick

This chapter introduces Cymbal and its subset DSQL, Daytona’s SQL, by means of an informal sequence of examples to give the reader a feeling for the highlights of the languages. Obviously, the rest of the manual is devoted to explaining the nature of Cymbal in full detail. Those who’d like to run these examples on the sample database that comes with Daytona should consult the hands-on tutorial "Getting Started With Daytona: A Hands-On Tutorial" (a separate document). The next chapter, "Loading Data, Defining Metadata, And Running Queries", provides the full story on using Daytona in the UNIX environment. The appendices contain grammars for the languages.

2.1 Quick Querying Of Relational Databases

By way of terminology, a relational database can be thought of as a collection of tables (or relations). For simplicity, each table can be thought of as a sequence of distinct rows (or records or tuples) sharing a common format. Each row consists of a sequence of values for the corresponding sequence of column headings (or attributes) in the associated format. The values for any given column heading are typically constrained to be members of some datatype such as those of the integers or the strings.

For example, consider a SUPPLIER relation or table which has the attributes of Number, Name, City, and Telephone for organizing information about suppliers. Here is a partial listing of SUPPLIER:

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>City</th>
<th>Telephone</th>
</tr>
</thead>
<tbody>
<tr>
<td>400</td>
<td>&quot;Acme Shipping&quot;</td>
<td>&quot;St. Paul&quot;</td>
<td>&quot;612-149-5678&quot;</td>
</tr>
<tr>
<td>401</td>
<td>&quot;Pinnacle Export&quot;</td>
<td>&quot;Minneapolis&quot;</td>
<td>&quot;516-816-2925&quot;</td>
</tr>
<tr>
<td>402</td>
<td>&quot;Halycon Import&quot;</td>
<td>&quot;Indianapolis&quot;</td>
<td>&quot;516-541-5212&quot;</td>
</tr>
<tr>
<td>403</td>
<td>&quot;Tops Export&quot;</td>
<td>&quot;Phoenix&quot;</td>
<td>&quot;401-294-6943&quot;</td>
</tr>
<tr>
<td>404</td>
<td>&quot;Webley Rentals&quot;</td>
<td>&quot;Albuquerque&quot;</td>
<td>&quot;315-547-7678&quot;</td>
</tr>
<tr>
<td>405</td>
<td>&quot;Highpoint Import-Export&quot;</td>
<td>&quot;Wichita&quot;</td>
<td>&quot;802-054-6329&quot;</td>
</tr>
</tbody>
</table>

To get the suppliers from St. Paul and their phone numbers, Daytona users could use the following very simple SQL query (stpaul.1.S in the Daytona test suite):

```sql
select Name as Supplier, Telephone as Phone
from SUPPLIER
where City = 'St. Paul'
```

(The as construct in SQL allows the user to specify a label (e.g. "Supplier") to be used as a heading for the corresponding output column in place of the default label, which is the name of the field (e.g. "Name").)

As an illustration of the power of Daytona’s DSQL, the next query shows how procedural Cymbal
can be wrapped around SQL; processing this query will result in creating an executable that obtains from the command line the user’s chosen city regular expression pattern and then prints out all suppliers (and their phone numbers) from cities that match the pattern.

```plaintext
locals: STR .city_pat

set [ .city_pat ] = read( from _cmd_line_ );

select Name as Supplier, City, Telephone as Phone
from SUPPLIER
where City Matches .city_pat ;
```

The Cymbal predicate `Matches` offers full UNIX-style regular expression matching; this is much more powerful pattern matching than is provided by the ANSI SQL standard. Indeed, DSQL inherits many useful functions from Cymbal including some very powerful string substitution and formatting capabilities.

Only a few basic Cymbal constructs need to be defined in order to understand the Cymbal queries that appear in this chapter. First, Cymbal variables have names that begin with lowercase letters or underscores and continue with zero or more lowercase letters, underscores, or digits. To refer to the value of a variable, it is necessary to prefix the name of the variable with a "." as occurs with `.x`, which is read as "the value of the variable x". (Hence, using programming language terminology, Cymbal variables, just like UNIX® shell variables, must be explicitly dereferenced in order to refer to their values.) Cymbal variables take values from one of several classes, including integers like 45 and 1000, floats like 1.2, 1., and 23E4, and strings like "Tom" and "Helena’s first chance: ".

Note how the procedural Cymbal above coexists naturally and simply with the SQL as illustrated in part by the casual, unannounced use of the Cymbal variable `city_pat` in the `select` statement to effectively parameterize the `select` with outside information obtained at runtime. Other interfaces, like C-embedded SQL, of programming languages to databases are not this simple.

Secondly, as illustrated by `[ 1, 2, "New York" ]`, explicit lists of objects in Cymbal are represented by using brackets around sequences of comma-separated objects. Cymbal calls such constructs TUPLES. (Cymbal fully capitalizes all object class names.) Clearly then, the `read` function returns a list or TUPLE, which is expected to be singleton in this case.

The declarative analog of the St. Paul query would be written in Cymbal (stpaul.3.Q in the test suite) as:
with_title_line  "Get phone numbers of suppliers from St. Paul"
do Display  with_format  _table_
each[ .supplier, .phone ]
each_time(
    there_is_a SUPPLIER where(
        Name = .supplier
        and Telephone = .phone
        and City = "St. Paul"
    )
);      

Paraphrased in English, this query says:

With the title string  "Get phone numbers of suppliers from St. Paul"
do display  with_format  _table_
each list of values for the variables  supplier  and  phone
each time there is a row in the SUPPLIER table
where the Name field’s value is the value of the variable  supplier
and where the Telephone field’s value is the value of the variable  phone
and where the City field’s value is the string   "St. Paul".

One answer to this query occurs when  .supplier = "Acme Shipping" and  .phone = "612-149-5678"
since it is true that

    there_is_a SUPPLIER where(
        Name = "Acme Shipping"
        and Telephone = "612-149-5678"
        and City = "St. Paul"
    )

The system handles Display calls in the following way. The  each  argument tells it that each
answer to the query will consist of values for the indicated variables in the order given. The  each_time
argument characterizes when a list of variable values is an answer: that is, it is an answer if and only if
the corresponding  each_time  assertion is true. The system’s task is to rummage around in the database
in whatever manner it chooses and  each time  it finds a list of values which makes the  each_time
assertion true, it is to present that list to the user as an answer. The system is also required to be
zealous enough in its search to discover every unique answer. Unless explicitly requested by other
means, Display calls should not be expected to produce the answers sorted in any particular order nor
to necessarily eliminate duplicates.

This is an example of declarative or non-procedural programming using the paradigm of symbolic
logic: it works by the user presenting Daytona with a characterization or description of the answers,
not a sequence of steps whereby the system is instructed to compute them. In declarative
programming, it is up to the system to determine the precise steps needed to compute the answer.
Such a sequence of steps would be procedural programming which most people are more familiar with
due to their use of languages like C and Perl.
Cymbal uses both paradigms. The SQL and logic parts of Cymbal are declarative and yet Cymbal also contains in a fully integrated fashion, a more-or-less conventional procedural dialect as well. Each programming style has its advantages in different circumstances.

Daytona offers the user a choice of five output formats for requests. The default is to put the answers with comments into UNIX stdout which may then be redirected to a flat, delimited ASCII file. This is often an attractive option to take since then the output of the query is ready to be included in the database; all that needs to be done is to describe it to the data dictionary and to define indices for it. However, in this case, using the table option instead (recall \texttt{with\_format\_table}), the answers to this query would be printed out on the terminal as:

\begin{verbatim}
Get phone numbers of suppliers from St. Paul

--------------------------------
Supplier       Phone
--------------------------------
Acme Shipping  612-149-5678
Bouzouki Receiving 612-943-7416
Julius Receiving  612-309-3492
Sunshine Warehouse 612-303-7074
--------------------------------
\end{verbatim}

Observe that the variable values are printed out in the order indicated by their position in the \texttt{each TUPLE} and that they are labelled by the capitalized versions of the corresponding variable names which may, in fact, differ from the corresponding attribute names (e.g., ``Supplier'' is not the same as ``Name'').

The third output format is the packet format which prints each answer record as a list of name-value pairs: this is convenient for answers with so many columns that a table becomes unwieldy. The fourth output format is that of Cymbal descriptions which are, in effect, a variant on the packet format and which are ready to be processed by the \texttt{backtalk} deparser generator and macro processor. The fifth output format is in XML.

The tabular output could have been obtained by means of interaction with the Daytona menu system \texttt{Dice}. In order to have obtained it outside of Dice at the shell prompt, the user would first have used a text editor to type the query into a file, say \texttt{Q.1}, and would then have executed the command \texttt{DS QQ Q.1}. This command begins by showing the query to the user for a last perusal and then executes it all the way down to the answers. An alternative is the command \texttt{DS Tracy} which offers the user more options for the processing of the query.

The above query is actually a procedure call in Cymbal (which is overwhelmingly non-procedural!). The keyword \texttt{do} invokes the procedure \texttt{Display} with arguments corresponding to the keywords \texttt{with\_title\_line}, \texttt{with\_format}, \texttt{each}, and \texttt{each\_time}. The argument for the keyword \texttt{with\_title\_line} is a string title for the default output. The argument for the keyword \texttt{with\_format} is one of \texttt{table}, \texttt{packet}, \texttt{data}, \texttt{desc}, and \texttt{xml}. The argument for the keyword \texttt{each} is (and must be) a \texttt{TUPLE} of variable references. The argument for the keyword \texttt{each\_time} is some general logic assertion.

Clearly, the SQL query is shorter than the declarative Cymbal query and it is to be preferred in
this instance. But don’t be misled. There are many queries that when expressed in Cymbal are far more comprehensible than they are in SQL. There are also many queries that can be nicely expressed in Cymbal but cannot be expressed at all in SQL. Daytona offers both and both together; choose what’s best for the query at hand. Similar comments can be made when comparing procedural Cymbal with Perl.

### 2.2 Finding Suppliers With Unfilled Orders

The next query will make reference to the ORDER relation which uses the attributes of Number, Supp_Nbr, Part_Nbr, Date_Recd, Date_Placed, and Quantity. Here is a partial listing of ORDER’s records with the absence of a value indicating missing values:

<table>
<thead>
<tr>
<th>Number</th>
<th>Supp_Nbr</th>
<th>Part_Nbr</th>
<th>Date_Recd</th>
<th>Date_Placed</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>401</td>
<td>174</td>
<td>^4/24/84^</td>
<td></td>
<td>2311</td>
</tr>
<tr>
<td>2</td>
<td>492</td>
<td>156</td>
<td>^9/24/86^</td>
<td>^7/26/83^</td>
<td>2787</td>
</tr>
<tr>
<td>3</td>
<td>475</td>
<td>116</td>
<td>^10/6/84^</td>
<td></td>
<td>134</td>
</tr>
<tr>
<td>4</td>
<td>485</td>
<td>173</td>
<td>^4/21/84^</td>
<td></td>
<td>668</td>
</tr>
<tr>
<td>5</td>
<td>452</td>
<td>159</td>
<td>^10/4/85^</td>
<td>^4/11/83^</td>
<td>3920</td>
</tr>
<tr>
<td>6</td>
<td>499</td>
<td>185</td>
<td>^9/2/85^</td>
<td>^7/11/84^</td>
<td>4770</td>
</tr>
<tr>
<td>7</td>
<td>440</td>
<td>142</td>
<td>^12/28/85^</td>
<td>^4/19/84^</td>
<td>4733</td>
</tr>
</tbody>
</table>

So, raising the stakes somewhat, here is a query (late612.1.Q) that asks for the order number, order quantity in thousands, supplier, and supplier phone number for each unfilled order since the first of ’84 that has a quantity > 500 and that is from a supplier in the 612 area code.
do Display

    with_title_lines
    [ "Identify unfilled orders from suppliers in the 612 area code",
      "placed after 1-1-84 with quantities > 500" ]

each [ .order_number, .quantity_in_K, .supplier, .supplier_phone ]

each_time(

    there_is_an ORDER where(
        Number = .order_number
        and Supp_Nbr = .supp_nbr
        and Date_Placed >= '1-1-84'
        and Quantity = .quantity which_is > 500
        and Date_Recd Is _absent_
    )

    and .quantity_in_K = round_to_nearest( .quantity, 1000.0 )

    and there_is_a SUPPLIER where(
        Name = .supplier and
        Number = .supp_nbr and
        Telephone = .supplier_phone
        for_which( .supplier_phone Matches "^612" )
    )
)

Notice that when there is more than one title_line, the lines are put into a TUPLE argument to the keyword 'with_title_lines'.

This query shows that, in addition to assertions of TUPLE existence, there can be other assertions that also place conditions on variable values and that any of these kinds of assertions can be "anded" together.

In general, for Cymbal assertions \( A, B, \) and \( C, \)

\[
\neg A \\
\text{not } A \\
\text{if } A \text{ then } B \text{ else } C \\
\text{if } A \text{ then } B \\
A \text{ and } B \\
A \text{ or } B \\
( A ) \text{ if_and_only_if } ( B )
\]

are all Cymbal assertions where the first two are considered the same. Conjunction and disjunction both group from the left. The logical operators are mentioned above in order of decreasing precedence so that negation has precedence over conjunction which has precedence over disjunction and so on. Parentheses may be used to effect different groupings and precedences.

Note that the keyword 'absent' is used to assert that a field value is missing; to assert that a value...
is present, use the keyword _present_.

Observe that the presence of . supp_nbr in both the ORDER and SUPPLIER assertions serves to link these two assertions together in that for any values of the variables that make all of the conjuncts true, the value of the variable supp_nbr (i.e., . supp_nbr) is the Supp_Nbr attribute value for an ORDER record and the Number attribute value for a SUPPLIER record. The implication of course is that then . supplier_phone is the phone number of the supplier . supplier from the 612 area code whose unfulfilled order was placed after 1-1-84 and had a quantity greater than 500.

Also observe the use of the variable quantity_in_K which does not directly take its values from any database field, for indeed, its values are computed from the values of another variable. In general, new, non-database variables are easy to introduce and use in Cymbal, this not being the case for SQL.

Also, as might be expected, any of the standard arithmetic functions are available for use as well as such functions as floor, ceil, mod, and abs. round_to_nearest rounds its first argument to the nearest value of the second argument. The assertion that . supplier_phone Matches "^612" asserts that the value of the supplier_phone variable matches the regular expression pattern "^612". This requirement could also have been expressed by using the substring function in the assertion

\[
\text{substr}( \text{. supplier\_phone}, 1, 3 ) = "612"
\]

Notice that the >= predicate has been used here to refer to dates. In fact, the >= predicate symbol is polymorphic since it can be used to refer to the correct ordering predicate for integers, floats, strings, and dates.

### 2.3 Finding The Average Quantity Ordered Of Each Part

The next query (partavg.1.Q) shows one way to use aggregate functions in Cymbal. In the orders database, each part is often ordered more than once and even that from different suppliers; the goal then is to compute for each part, the average amount ordered over all of its suppliers.
do Display
   with_title_line
   "average quantity ordered of each part over all of its orders"

each [ .part, .avg_ordered ]

each_time(
   there_is_a PART where(
      Name = .part and
      Number = .part_nbr )

   and .avg_ordered =
     avg( over .quantity
         each_time(
            there_is_an ORDER where(
               Part_Nbr = .part_nbr
               and Quantity = .quantity )))
);

The new construct here is that of the built-in aggregate function \textit{avg} which takes two keyword arguments. Its \textit{over} keyword argument is the value of some variable and its \textit{each\_time} argument is an assertion which refers to the value of that variable as well as possibly other variable values.

In this case, each answer consists of a pair of values for the variables \textit{part} and \textit{avg\_ordered} such that somehow there is a value for the variable \textit{part\_nbr} so that \textit{.part} is a part and \textit{.part\_nbr} is the part number for that part and \textit{.avg\_ordered} is the average of the values for \textit{quantity} each time there is an order with that part number and that quantity.

The variable \textit{part\_nbr} plays a very important role here in that it communicates part number information from outside of the \textit{avg} function call to inside the call where it is used to determine exactly which orders will be considered for each average.

Notice that none of the variables need to have explicit definitions. This is because of the 4GL nature of Cymbal where datatype deduction is done by the system wherever possible. For example, the datatype of \textit{part\_nbr} is deduced from that of the corresponding field and the datatype of \textit{avg\_ordered} is deduced from that of the built-in function \textit{avg}.

The system is also on guard to protect the user against unexpected division by 0. In the event, that no orders were placed for a given part, the nonsense value of \textit{−9e36} is printed out for the corresponding average.

Cymbal’s aggregate functions are very powerful since the \textit{each\_time} assertion can be fully general.

Cymbal provides a full panoply of aggregate functions. In addition to the standard first-order statistical functions \textit{min}, \textit{max}, \textit{sum}, \textit{prod}, \textit{avg}, and \textit{count}, Cymbal also provides the second-order statistical functions \textit{var}, \textit{stdev}, \textit{covar}, and \textit{corr}.

Furthermore, Cymbal also has a TUPLE-valued \textit{aggregates} function that provides for the parallel computation of any list of aggregate functions. In conjunction with associative arrays or even with the Cymbal notion of a "box", Cymbal provides a grouping capability for applying TUPLES of aggregate
functions which is quite a bit more powerful than what SQL offers with GROUP BY/HAVING. These latter topics are covered in Chapter 11, Chapter 12 and Chapter 14.

### 2.4 Finding Parts Ordered From A Given Supplier

The next query provides another example of how the procedural portion of Cymbal can be used together with the declarative portion to create compiled parameterized queries. This goal of this query is to produce an executable which will take a supplier name as a runtime, command-line argument and therewith produce as output, information on parts ordered from that supplier in 1984, when that information is available. Such information is available if the given supplier is in the SUPPLIER table and if that supplier is neither Acme Shipping nor Tops Export, about which information is being kept secret. Here is the query (supporders.IQ):

```plaintext
local: STR(30) .supplier
set [.supplier] = read( from _cmd_line_ ) otherwise do Exit(1);
when{
   there_is_no SUPPLIER where( Name = .supplier )
   or .supplier Is_In { "Acme Shipping", "Tops Export" } 
} do {
   skipping 1 trailing 2 to _stderr_
   do Write( "No information about supplier ", .supplier, " is available." );
} else do {
   with_title_line "Orders placed in 1984 to Supplier " + .supplier
   do Display with_format _table_
      each[ .date_placed, .part, .quantity ]
      each_time(
         there_is_a SUPPLIER where( Name = .supplier
            and Number = .supp_nbr )
         and there_is_an ORDER where(
            Supp_Nbr = .supp_nbr
            and Part_Nbr = .part_nbr
            and Quantity = .quantity
            and Date_Placed = .date_placed
            which_is >= ^1-1-84^ & < ^1-1-85^ )
         and there_is_a PART where( Name = .part and
            Number = .part_nbr )
      );
}
```

There’s lots to say about this query. First, the query begins with a variable definition statement that defines the variable `supplier` to be a local string variable that takes values up to length 30. Such
variable definition statements may optionally be terminated with a semicolon.

The rest of this query consists of a sequence of procedural commands, beginning with an assignment statement specifying that the variable supplier is to get its value from the next item read from the command line. If no such value is there, then the program exits with an exit status of 1 and an informative message is provided automatically to stderr due to the use of an otherwise clause. Functions like read may employ keyword arguments exclusively within parentheses or else exclusively positional arguments (which read does not). Incidentally, since supplier is said to be STR(30), if the value given at runtime as a command line argument is longer than 30 characters, Daytona will detect this type violation and exit with a warning message.

The command following the read assignment is a conditional "when-do-else-do" statement. The condition of the when-do-else-do is a disjunction of two assertions, one of which makes a statement about suppliers in the SUPPLIER table. (A phrase like there_is_no SUPPLIER where( ... ) is considered the same as not there_is_a SUPPLIER where( ... ). It is just one example of a fairly large number of variants on there_is_a syntax that Cymbal employs.) This constitutes a small demonstration of the degree to which the declarative and procedural portions of Cymbal have been synthesized. In fact, the user is free to include assertions in when-do-else-do commands that also involve aggregate functions and there_exists and for_each quantifiers as well as any of the other constructs that can be used to build declarative logic assertions in Cymbal.

The second disjunct in this condition specifies that the value of supplier is a member of the set of strings consisting of two elements: "Acme Shipping" and "Tops Export". In Cymbal, finite sets that are known by an explicit listing of their elements are called BUNCHES.

Furthermore, in what are called intensional boxes, Cymbal also provides the means of characterizing sets by membership assertions. For example,

```
.supp_nbr Is_In { .x :
    there_is_an ORDER where(
        Supp_Nbr = .x
        and Quantity > 500 ) }
```

asserts that supp_nbr is a member of the set of supplier numbers whose associated orders have quantities larger than 500. The above box assertion can also be rewritten in a less mathematical notation as:

```
.supp_nbr Is_Something_Where(
    there_is_an ORDER where(
        Supp_Nbr = .supp_nbr
        and Quantity > 500 ) )
```

There are also list boxes and boxes that contain the extensions of path predicates. The box concept provides Daytona with its in-memory tables complete with multiple skip-list indices (not illustrated here).

The when-do-else-do is one of the basic primitives in the procedural portion of Cymbal along with assignments and procedure calls. It is the user’s option as to whether semicolons are procedural statement terminators, separators, or both. Or for that matter, semicolon usage may conform to C
conventions.

Notice that keyword arguments (such as `skipping` 1) to procedures like Write and Display may appear immediately before the call as well as within the parentheses, provided that, in the latter case, all of the arguments therein are then keyword and not positional. The `skipping` argument specifies how many new-lines to emit prior to writing anything else; `trailing` does the converse. `_stderr_` illustrates the before-and-after-underscore syntax that Cymbal uses in writing symbolic constants.

Notice the absence of format specifications in Write calls. Being a 4GL, Cymbal knows (in fact, usually deduces) the types of variables and then when asked, prints out their values using standard formats.

The above query highlights the true nature of Display. It really is just a high-powered procedure and therefore, calls to Display can appear in a sequence of commands anywhere any other procedure call can. The declarative part of a Display call is contained solely in its `each_time` argument; otherwise, it is a procedural construct. In other words, it is really a hybrid.

Note that the `with_title_line` argument is an (overloaded) call to the `plus` function which since its arguments here are strings, is resolved into string concatenation.

This Display call is interesting in that it involves what would be called in the relational setting a 3-way join of the SUPPLIER, ORDER, and PART tables. The nature of the `each_time` assertion here can be understood in exactly the same way that similar assertions were understood in the preceding examples. The only additional wrinkle here is the easy to understand use of the extended predicate `>= `1-1-84` & < `1-1-85`.

Note that SQL alone cannot express this query nor can it express the next one.

2.5 The for_each_time Looping Construct

Now suppose that the user wanted to completely control the printing of the order information instead of relying on the tabular formatting provided by Display itself. Then Cymbal’s `for_each_time` construct can be brought into play and the query rewritten (with other improvements) as:
set [.supplier] = read( from _cmd_line_ ) otherwise do Exit(1);

when{
    there_is_no SUPPLIER where( Name = (STR(30)).supplier )
    or .supplier Is_In { "Acme Shipping", "Tops Export" }
} do {
    skipping 1
    do Exclaim_Line( "No information about supplier .supplier is available."ISTR );
} else do {
    skipping 1
    do Write_Line( "Orders placed in 1984 to Supplier .supplier"ISTR );
    set .nbr_of_orders = 0;
    for_each_time [ .date_placed, .part, .quantity ]
    is_such_that(
        there_is_a SUPPLIER where( Name = (STR(30)).supplier and
            Number = .supp_nbr )
        and there_is_an ORDER where( 
            Supp_Nbr = .supp_nbr 
            and Part_Nbr = .part_nbr 
            and Quantity = .quantity 
            and Date_Placed = .date_placed
            which_is >= 1-1-84 & < 1-1-85 )
        and there_is_a PART where( Name = .part and 
            Number = .part_nbr )
    )
    do {
        set .nbr_of_orders++;
        skipping 2
        do Write_Line( "Date_Placed = ", .date_placed );
        do Write_Line( " Part = ", .part );
        do Write_Line( " Quantity = ", .quantity )
    } 
    skipping 2
    do Write_Line( "Number of orders = ", .nbr_of_orders )
}

A Write_Line call is the same as the corresponding Write call with a final "\n" argument appended; Write_Line is used to avoid typing backslash escapes. An Exclaim is a Write done to stderr instead of Write’s default stdout.

Note the appearance of an ISTR or two, which is an acronym for ‘interpolated string’. These quantities have type STRING but if any variable dereferences appear within them, then Daytona will
Anyway, the major attraction here is the for_each_time construct, which is one of the most significant illustrations of the degree to which the declarative and the procedural have been integrated in Cymbal. The general form of this construct is:

\[
\text{for\_each\_time} \ [ \text{variable\_value\_list} ] \\
\text{is\_such\_that} ( \text{assertion\_involving\_the\_variable\_values} ) \\
d\{ \\
\text{command\_sequence\_using\_the\_variable\_values} \\
\} \\
\]

The semantics specify that, as Daytona generates a sequence of TUPLES of variable values that satisfy the given assertion, it will execute the given command sequence for each TUPLE of values for the variables. The power of this construct derives from its ability to handle completely general assertions determining what should be done by completely general programs. It is the principal bridge between the procedural and declarative worlds. In fact, Display is implemented using for_each_time.

The command set .nbr_of_orders = 0 is a simple Cymbal assignment statement which should be read as "set the variable nbr_of_orders value to equal 0". The value of this variable is increased by one each time an order for the given supplier is found. The total number of orders found is then printed out at the end.

Since the variable supplier has not been given an explicit definition in this example, Daytona will deduce a type of STR(*) for it. This means that the variable can hold string values that are arbitrarily large. However, since the SUPPLIER table has an index on the Name field, Daytona requires that the use of .supplier in there_is_no and there_is_a conform exactly to the type of the field which happens to be STR(30). Consequently, this query casts the value of supplier appropriately to STR(30) where needed: this implies a truncation to size 30, if necessary. In addition to illustrating how to cast, this also implies the existence of a data dictionary where keys and indices are defined and it implies built-in optimization to detect opportunities to use the best indexing available. These topics are dealt with later.

### 2.6 More Procedural Cymbal

The Cymbal tokens function supports full regular expression lexical analysis using the regular expressions of egrep(1). To illustrate this and a few other procedural constructs, here is the previous command-line St. Paul SQL query rewritten in procedural Cymbal (stpaul.proc.IQ). Recall that this query proceeds by reading in a city from the command line and then by printing out the suppliers from that city along with their phone numbers. The logic here is complicated by the fact that since there are comments in the file, the file is inhomogeneous and not every line contains an answer worth printing.
set [.city_pat ] = read( from _cmd_line_ ) otherwise do Exit( 1 );
set .line_pat = "[^\t]*\t([^\t]*\t)\t" + .city_pat + "\t\([^\t]*\)"

set .in_chan = new_channel( for "john/d/SUPP" );
loop {
  set [.line ] = read_line( from .in_chan );
  when( At_Eoc[ .in_chan ] ) break;
  set [.supplier, .phone ] =
    tokens( for .line matching .line_pat );
  when( RE_Match_Worked )
    with_sep "|" do Write_Line( .supplier, .phone );
}
do Close( .in_chan );

Without going into too much detail at this juncture, this query begins by reading in a value for the
city_pat variable from the command line. If there are no command line arguments (i.e., we are at the
End Of this I/O Channel), then the query exits with an error status of 1. The query continues by
setting .line_pat equal to the desired line search pattern for producing the Suppliers and Phones from
lines that match the City pattern. Incidentally, the delimiter in SUPPLIER's file is the tab character.

Then, .in_chan is set equal to a newly opened I/O channel that will be reading in information from
the file "john/d/SUPP" where the SUPPLIER records are stored. Lines are read in from this channel
sequentially until the end of the channel is reached. When the tokens function is able to extract values
for the supplier and phone variables from the current line, these values are separated by a bar and
written as a line to the output channel (stdout, by default).

One thing this query illustrates nicely is that even with fairly sophisticated procedural processing,
declarative queries can have a lot going for them. In fact, there is actually a declarative usage of tokens
where it is used within an assertion instead of an assignment in order to define TUPLES of values.
Using this declarative style, the above query looks like this (with a few other modifications)
(stpaul.tok.2.Q):

set [.city ] = read( from _cmd_line_ but_if_absent [ "Chino"STR(25) ] )
  otherwise do Exit( 1 );

for_each_time [ .supplier, .phone ]
is_such_that(  
  [ ?, .supplier, .city, .phone ]  
    = ptokens( for "cat $ORDERS_HOME/SUPP | egrep -v 'Ã#'" upto "\t\n" )
  )
  do Write_Words( .supplier, .city, .phone );
}

This query is particularly illustrative. First, the but_if_absent keyword argument is used to provide
"Chino" as the value for city if there are no arguments on the command line. Note also how the type
STR(25) is given explicitly so that it can be inferred for city.
Next there is no database table involved in this query. This is clear proof that Cymbal can be used as a programming language in the complete absence of a database table, indices, and data dictionary. Instead, the \textit{ptokens} variant of \textit{tokens} is being used to access a data file directly. \textit{ptokens} works by reading its tokens by means of a pipe where in this case the tokens are separated by either a tab or a newline. \textit{ptokens} is needed because the data file $ORDERS\_HOME\_SUPP$ contains comment lines starting with \# that need to be ignored by means of \texttt{egrep -v}. (When treated as a database table, these lines are automatically ignored.)

Obviously, the lines that are of interest contain 4 tokens, our disinterest in the first token being indicated by the use of a so-called skolem \?. In the declarative setting, the call to \textit{tokens} (or to any of its variants) is considered to generate a stream of TUPLES of tokens which are considered to satisfy the associated equality if any associated tests are passed; in this case, the test is that the third token equal the value of \textit{city}. (This concept will be a surprise to people not familiar with declarative programming. Chapter 9 explains the declarative paradigm in much more detail than is appropriate here; after all, the goal of this chapter is just to give a flavor for the language.)

The call to \texttt{Write\_Words} is considered the same as a call to this:

\begin{verbatim}
    with_sep " " do Write\_Line( .supplier, .city, .phone );
\end{verbatim}

meaning that the three arguments will be written out on a line separated by a space.

### 2.7 Associative Arrays

Cymbal offers an extremely powerful associative array feature whereby scalars or TUPLES of scalars can be mapped to scalars or TUPLES of scalars. Here is a non-database Cymbal program that uses the \textit{ftokens} variant of \textit{tokens} to tokenize the \texttt{/etc/passwd} file into a sequence of the user login shells so that the number of each kind of shell can be counted and eventually printed (tok.7.Q):

\begin{verbatim}
local: INT .sh\_cnt[ STR : with_default @ => 0 ]
for\_each\_time .shell
    is\_such\_that( [ 6?, .shell ] = ftokens(for "/etc/passwd" upto ":\n" ) ){
        set .sh\_cnt[ .shell ]++;
    }
for\_each\_time [.sh, .cnt ]
    is\_such\_that( .sh\_cnt[ .sh ] = .cnt )
    { do Write\_Words( .sh, .cnt ); }     
do Write\_Words( "The total number of distinct shells is", .sh\_cnt.Elt\_Count );
\end{verbatim}

(The skolem term \texttt{6?} causes the tokenizer to skip over 6 tokens.)

The simple associative array \texttt{sh\_cnt} maps STRINGS to INTS. The program increments the cumulative count of each kind of shell each time an instance is found. (The default specification \texttt{@ => 0} serves to start each cumulative count out at 0.) Note the declarative way that the second
**for each time** accesses the contents of the associative array so that they can be printed out. Actually, a Cymbal box can be (implicitly) used to sort the output lexicographically by shell:

```c
for_each_time [ .sh, .cnt ]
    is_such_that( [ .sh, .cnt ]
        Is_The_Next_Where( .sh_cnt[ .sh ] = .cnt ) in_lexico_order )
    { do Write_Words( .sh, .cnt );
    }
```

Boxes are good for sorting and for eliminating duplicates.

Daytona maintains the current number of mappings defining an associative array and makes it available by means of a structure dereference of the array using the member tag `Elt_Count`.

This program is actually doing what SQL users would call a group-by. By means of associative arrays, Cymbal can express many kinds of group-bys that are impossible to express in SQL.

### 2.8 Transitive Closure

Consider the following tiny table which has two fields per record: Name and Children. Children is one of Daytona’s LIST/SET-valued fields so that, for example, Steve is being said to have the set of three children: Ray, Roger, and Romy.

```
Steve: [Ray:Roger:Romy]
Ray: [Paula:Pat:Penny:Pandora]
Paula: [Mindy:Max]
Roger: [Pablo]
Romy: []
Penny: [Martin:Milton]
Pandora: [Maybelle]
```

Without getting into the details of transitive closure and path PREDs (see Chapter 16), the next query (steve.1.Q) shows how easy it is to display all the descendants of Steve:

```c
do Display each[ .desc ]
    each_time( .desc Is_A_Descendant_Of "Steve" );
define path PRED: Is_A_Descendant_Of [ .x, .y ]
    by_stepping_with(
        there_is_a PERSON where( Name = .y and one_of_the Children = .x )
    )
```

(Actually, the definition of `Is_A_Descendant_Of` is contained in the separate environmental file `orders.env.cy` where Daytona automatically accesses it as needed.)

By the way, the answer is that everyone is a descendant of Steve.

### 2.9 Updates And Transactions

In addition to full SQL transactions and updates, Daytona provides more powerful and flexible
updating capabilities with Cymbal. The following query serves to give the flavor of transactions in Cymbal (supp.IQA):

```cymbal
do Add_Supplier( with_name "Blue Bells" with_number 777 with_city "Troy" );
do Add_Supplier( with_name "Acme Shipping" with_number 777 );
global_defs:
/* City defaults to "New York" */
define PROCEDURE transaction task :
Add_Supplier(
    with_name STR(30) .supplier,
    with_number INT( _short_ ) .nbr,
    with_city STR(25) .city = "New York"
)
{
    // since no Phone given, it is treated as missing
    do Change so_that(
        there_is_a SUPPLIER where( Name = .supplier and
        Number = .nbr and City = .city )
    );
}
```

This program consists of two calls to the `Add_Supplier` transaction task defined in the `global_defs` section of the program. `Add_Supplier` takes keyword arguments, one of which (`with_city`) has a default for use if the argument is omitted in any call. Cymbal has exactly one construct for changing things in the database and that is the `Change` procedure call. In this case, `Change` is being used to ensure that there is a SUPPLIER with the stated attributes in the table and in fact, it will add such a one to the table in the event it isn’t already there (which is probably the case). Notice that Cymbal employs the C and C++ comment conventions.

When possible, SQL transactions are available for their conciseness but it is Cymbal that is expressive enough to cover a much more extensive space of transactions.

This concludes this brief tour of Cymbal. The next chapter discusses how to use Daytona in the user’s UNIX environment. Subsequent highlights include parallelization and networking.
3. Loading Data, Defining Metadata, And Running Queries

This chapter explains the mechanics of using Sizup to validate and index data and of using Tracy and Stacy to process Cymbal and SQL queries on Daytona UNIX flat file (i.e., DC) databases. The topics in sequence are: data file formats, project archives, application archives, keys and indices, performance considerations, shell environments, Daytona commands, including Sizup and Tracy, and built-in m4 macro processing. At the end, tips are provided for how to resolve certain common problems. A hands-on introduction to this material may be found in the “Getting Started With Daytona Step-By-Step” tutorial (a separate document).

3.1 UNIX Flat Files

Data stored in Daytona’s UNIX flat file format is said to be stored in the DC format (i.e., for ‘D’elimited ‘C’-language-based Daytona data backend.) A DC data file is a generalization of the common UNIX flat file whose data file records consist of a sequence of ASCII field value strings separated by occurrences of a field value separator character and terminated by a new-line character. Here is such a data file record (aka DFR):

105|bolt|azure|1.5

The Daytona default field separator character is |; as explained in the next section, the user may specify any other ASCII character if desired with the exception of the new-line character and the DC forbidden characters. (The DC forbidden characters are the following characters which Daytona forbids being in DC data files: ‘\0’ (null), ‘\001’ (‘A), and ‘177’ (DEL).) The user may wish to avoid tabs as field value separators since text editors usually do not distinguish tabs from spaces when displaying them on the screen. The maximum record length supported by the DC backend is 64K (i.e., 65535) bytes.

Daytona LIST/SET-valued fields extend the usual UNIX flat file format by enabling a field to have a LIST or SET of scalars as a value. These values are separated by the LIST/SET-valued field separator character (which defaults to the field separator character) and these values are enclosed as a group by the TUPLE delimiting characters which by default are [ and ]. Here are several data file records with fields Parent and Children, the latter being LIST/SET-valued:

Ray:[Paula:Pat:Penny:Pandora]
Roger:[Pablo]
Romy:[]

3.1.1 Representing Missing Values In DC Files

For DC, a field has a missing value in a given data file record if there are absolutely no characters present at its location in the record. Consequently, such an absence appears in a DC data file record as two juxtaposed field separators or by a record starting or ending with a field separator or in the case of LIST/SET-valued fields, by having the TUPLE-delimiter-end character immediately follow the TUPLE-delimiter-start character. (In the LIST/SET-valued field case, since it doesn’t make sense to have one or more (but not all) of the elements of the TUPLE field value be missing, it is illegal to have two juxtaposed separators or any other indication of missing values except for the total absence of all
elements for the TUPLE.) Just for emphasis, a field value consisting of one or more blanks is not considered to be missing.

One implication of this is that, on the surface, there is no way to represent an empty-string value for a string field since the absence of any characters in the data file representation of the value of a field is taken to mean that that field’s value is missing (or absent) for that data file record, not that it is the empty string. The presence of something (in this case, the empty string "") is not distinguishable from the absence of everything in this case. However, the use of Default_Values (elaborated upon later in this chapter) enables the user to require Daytona to interpret the absence of a field value in the data file to mean the presence of the specified Default_Value, which could well be (for STRING/STR FIELDS) "" (which, once again, is a string and not a system ‘missing value’ indicator). Now, there is an important caveat here: if a Default_Value is used for a field, then there is no such thing as a missing value for that field anymore. Consequently, the system is no longer going to do any automatic missing value handling for that field. In this case, if some missing value handling is needed, then the user must do it for themselves. This means that they will have to distinguish a special value (such as "" or −1) as their field missing value indicator for that field and their Cymbal/DSQL code will have to test for it. Using Cymbal/DSQL _absent_, _present_ or NULL in this situation will either have no effect or will generate error messages.

What other DBMS would call a null value, Daytona is calling a missing value. The rationale is two-fold. First, there are many possible semantics for a null value (such as not-applicable), but in Daytona, the semantics are that the data simply do not provide any value at all for a given field in a given record. Second, languages like SQL enable their variables to take null values as values, i.e., a null value is a special symbol whose presence indicates the absence of any ‘real’ value for the variable. This is not possible in Cymbal: no variable can have as its value a null value; it can only assume values from its type (i.e., integer values for integer variables, date values for date variables etc.) This is all consistent with Daytona’s fundamental philosophy concerning missing values (from which all of its related behavior can be derived), i.e., “you can’t work with something that’s not there”. In short, note that Daytonas’s notion of a missing value is based totally on syntax in the data file; once information has been read in from the data file into program variables in memory, there is no notion of missing value. Handling missing values at the Cymbal level is discussed in Chapter 13.

(There are Daytona types that cannot even appear in data files as values which can have truly C-null values because they are implemented as pointers to C structures. Such types include CHANNEL and TENDRIL but since their values can’t be stored in a data file, they can’t in fact enter into a discussion about database null values. Also, note that the empty string "" is a string that is empty; it is not the absence of a string and hence it is not a missing or null value.)

### 3.1.2 Comments in DC files

Comments may also be added to DC files. The comment beginning character is user-definable and is % by default. Comments begin with a comment beginning character and continue to end at the last character before the next field value separator, new-line, or TUPLE-delimiter-end character. This means that entire lines may be comments, that data file records may end with comments, and that field values may end with comments. By using line and field comments, a data file record may thereby be padded out so that all subsequent Daytona updates to that record will be made in place. Comments can be used to identify the source of the data as well as to identify the field names for the data file.
records. In the latter case, it is a good idea to use the special “msg)flds)” format for identifying the fields in a data file record, as in

%msg)flds)Number|Name|City|Telephone_Number

Certain of Daytona’s file format filters are sensitive to this particular kind of comment and will use it in several ways, one of which is to produce column headings for tabular output. Comments may be placed immediately before the new-line terminating a data file record, thereby providing the ability to comment individual data file records.

### 3.1.3 Check_DC_Lines

When dealing with unruly data sources, it is frequently very helpful to be able to quickly examine a DC file to look for corruption and to test for general sanity. The Check_DC_Lines program will help the user do this. It can be provoked into providing its usage by giving it a +? argument:

```
prompt) Check_DC_Lines +?
```

In the order given, the arguments are the delimiter character(s), the comment character, the tuple delimiters (if any), the expected number of fields, the maximum number of errors to report, save-to-file directives, a verbosity directive, and the data file. The defaults for these arguments are as indicated. Note the missing default for tuple_delims: if the user does not explicitly provide a value for this keyword, then no LIST/SET/TUPLE/STRUCT syntax sanity checking will be done. If the verbosity directive is present, then Check_DC_Lines will summarize its run by printing out the total number of lines, total number of records, total bytes processed, average record length, min/max line length, and total errors. This program will alert the user to NULL characters (i.e., binary 0) appearing in the data, the `A (ctrl-A) character appearing, the DEL character appearing, bytes with the 8th bit set appearing, and lines which do not have the same number of fields. It will also print out an error message if the optional number-of-fields argument is provided and a counterexample exists. When `-variants_ok` is used, Check_DC_Lines will not consider the presence of lines with differing numbers of fields to be an error; this supports data files containing variant records. When `-save_faults` is used, the records with errors will be saved to a file whose name ends with `.faults`. When `-save_nonfaults` is used, the records with no errors will be saved to a file whose name ends with `.nonfaults`. The base of these file names is that of the DC_file, if provided, or else check_dc_lines_output.

Note that Check_DC_Lines is only on the lookout for records which are toxic to Daytona and/or which are impossible to occur during normal functioning; it doesn’t know enough to check for any other kinds of unwanted phenomena, and especially so at the application level in the sense of what the user considers to be good data as specified in any associated rcd. Check_DC_Lines is faster than Sizup, which itself by default does further validation regarding the degree to which the data is described accurately by its rcd.
3.1.4 Further Syntactic Features Of And Limitations On Data File Records

To facilitate examination using vi, data file records may contain escaped or embedded new-lines which consist of two characters, a backslash followed by a new-line; this sequence can only appear immediately after the last byte of a field value (or LIST/SET-valued field element) with two exceptions: the sequence cannot precede either a new-line or a TUPLE-delimiter end character. (Also, please recall that the field value does not include any comment that may have been appended.)

While it is permissible in general for backslashes to appear in DC records, they may not appear immediately preceding any of the record delimiters with the exception of new-line as noted above.

Furthermore, completely empty lines are allowed and are simply ignored; they can be used to enhance readability of data files as well.

Data coming in from foreign environments such as those provided by Microsoft and IBM may well need a little editing before being suitable for Daytona. For example, it is good to remove leading and trailing blanks, leading 0s for numbers, trailing 0s for decimals, and so on. $DS_DIR/sqz* contains a variety of sed edits that can be adapted for local use.

3.1.5 Object Records

A data file record is just the computer implementation of a more abstract concept, the object record. More detail will be given on object records in Chapter 5 but for the moment, just think of an object record as consisting of a set of attribute-value pairs where the value for any given attribute might well be missing. Clearly, the ‘attribute’ concept is an abstraction of the ‘field’ concept. A set of object records all of which have the same attributes is called a record class. (In the literature, a record class is also known as a relation or table.)

Notice how much implementation detail has been abstracted away:
1. The concepts of a field separator, TUPLE delimiter characters, a comment character, records terminated by new-lines are all gone.
2. There is no analog of the notion of an order of appearance for the field values in a data file record nor for the notion of an ordering of those records in a file.
3. There is no analog of the notion of where the data files are stored on the machine.
4. The data file record field values are really all just ASCII character strings whereas for the object records, many other types such as integers and dates are possible.
5. Data file record field values could actually be encoded for compression or encryption purposes.
6. Object records could be represented and stored as data file records in one file or in separate files on different machines or on tape or as the output of pipes or even using a file format different from that of DC.

In short, the object record abstraction abstracts away many DC-format implementation details that would otherwise cause needless complication when writing queries.

(Caveat: the database community tends to regard ‘attribute’ and ‘field’ as synonyms and that
practice may be sporadically employed in what follows.)

3.2 Daytona Data Dictionary

Daytona users organize their data by means of a hierarchical classification scheme. As previously discussed, sets of fields are collected together into records and sets of records are collected together into files. One or more files of records comprise a (base) record class, the latter occurring when horizontal partitioning is used. A Daytona application is a group of record classes that are related in some way meaningful to the user. For example, the set of record classes associated with writing billing queries could constitute the “billing” application. orders and misc are the two applications that come with the Daytona test suite. Lastly, a Daytona project is group of applications. Whereas a Daytona user can maintain simultaneous access to any of several applications, they can only maintain access to at most one project at any given time. Not surprisingly, daytona is the project used by the Daytona test suite. It is not necessary for the user to specify a project: the null or empty project would then be the default. Of course, certain functionality outlined below would not then be available.

These choices of project and applications are made known to Daytona by means of the values of the shell variables DS_APPS and DS_PROJ discussed later in this chapter.

The Daytona data dictionary consists of descriptions of instances of the file, record class, application, and project concepts. In other words, using the special Cymbal description language, the user describes to the system particular files, record classes, applications, and projects. For example, descriptions of Daytona data files would include specifications for the user’s choice for comment characters and field separators. A description of a record class would include such information as the Cymbal datatypes of record field values.

3.2.1 Project Descriptions

Here is a daytona project description, which is meant to be read like a long, highly-structured English sentence:
there_is_a PROJECT meaning daytona where(
    Apps = [ orders, misc ]
    and Query_Path = `\$ORDERS_HOME/Q`
    and Executable_Path = `\$ORDERS_HOME`
    and Query_Output_Path = `\$ORDERS_HOME`
    and Txn_Log_Dir = `\${ORDERS_DATA:-$ORDERS_HOME}/TLOG`
    and Comment = `the underscore beginning _Query_Preprocessor_I
        nVocation neutralizes it`
    and _Query_Preprocessor_I
        nVocation = `cpp -P`
    and Max_Log_File_In_K = 1024
    and Max_Do_Queue_In_K = 16384

    and there_is_a MAKE_GOODS
    of_this PROJECT where(
        Comment = `the value of Make_CC could even be $(MY_CC)`
        and Make_CC = `$(DS_CC)`
        and Make_AR = ar
        and Make_LD = ld
        and Comment = `C++ needs a different cpp to prevent
            ‘Unreasonable include nesting’`
        and Cpp_Path = `$(DS_CPP)`

    and there_is_a FILES
    of_this MAKE_GOODS where(

        there_is_a FILE_BASE meaning daytona_aux
        of_this FILES where(
            Comment = `If no Source note, then . is assumed`
            and Depends_On = [ $(DS_DIR)/R.sys.h ]
            and Source = `\$ORDERS_HOME`)

        and there_is_a LIBRARY meaning libEnv.a
        of_this FILES where(
            Source = `’$(DS_DIR)’`
            and Comment = `the single quotes allow it to be
                expanded later during the make process`)

    )

) ) )

Notice that this project description makes use of the same "there_is_a" construct that appears in Cymbal queries. Actually, the user can write either the above English-keyword form or else the following more terse version (which means exactly the same thing and which is really more convenient):
Either of these representations may look formidable at first but each has 3 good things going for it:

1. it nicely organizes a lot of heterogeneous information (and indeed, it does it much better than a bunch of tables would),
2. it is extremely flexible, thus enabling Daytona developers to add new functionality to Daytona without adversely impacting current users,
3. users, in fact, can add their own information to the data dictionary, just as long as it doesn’t try to override or give a different meaning to system information.

Notice that the #{ ... }# descriptions are isomorphic to a subset of XML, one difference being that
Cymbal descriptions come with far richer semantics. At some future date, Daytona will enable users to use Cymbal to query the data dictionary for the information it contains.

Each one of the #{ ... }# constructs is describing something and is therefore considered to be a description. The object class of the object being described is given first and then its name, if any, is given in parentheses after that. So, this project description begins by saying that "there is a PROJECT meaning daytona where a value of the Apps attribute is the TUPLE \[ orders, misc \]." These applications constitute the applications that are permissible to use with the daytona project. Typically, one or more attribute-value pairs like the one for Apps will follow the object class and name of the object being described so as to give more information about it. LIST/SET-valued attributes are not used as often as the atomic/scalar-valued attributes but they are quite handy on occasion.

These <...> attribute-value pairs are called notes since they constitute little notes of information about the object being described in the #{ ... }# description.

(On rare occasions, the attribute values in #{ ... }# descriptions contain punctuation that would confuse the parser in which case they need to be enclosed in hats as illustrated by \[ .x > 1 \]. In general, hatted constructs are called THINGS and are described in Chapter 5. Defining an element to be either a scalar description attribute value or any atomic (non-TUPLE) value inside a TUPLE attribute value, when does an element have to be hatted? For the purposes of description syntax, suffice it to say an element must be quoted with hats if it contains characters like [, -> that would confuse the lexical analyzer as it tries to determine tokens between the , and the > that ends an angle-bracket note or the [, ] used to delimit elements of TUPLES. However, hatting elements never hurts. In general, the lexical analyzer is smart enough to support practically any common thing appearing without hats; this includes Type specifications like ( 0->1 ) INT which one might otherwise reasonably think of as causing trouble due to the > . So, if there is trouble, put the string in hats, i.e., make it hatted -- can’t hurt. Of course, the Daytona C program that does the lexing and parsing contains the exact characterization of when a string needs to be hatted -- and it is very forgiving and understanding and allows maximum freedom from hatting. But the simplest thing to do is to follow the stricter characterization above. By the way, hats can occur in a hatted string by escaping them with a preceding backslash.)

The Query Path, Executable Path, and Query Output Path notes are used by Dice to find the indicated kinds of files in the file system. The Txn Log Dir note is explained in Chapter 17.

More substantive information can be found in the descriptions that are nested within this PROJECT description. The nesting of description A within description B indicates that the object described by A is related to the object described by B. So, the description of the MAKE GOODS associated with the daytona PROJECT contains information specifying modifications to the process for making query executables for daytona. Please note that a MAKE GOODS description can also appear in an apd file that describes an APPLICATION, as will be introduced shortly. The user may provide a Make_CC note in order to specify the path to a C compiler other than the default cc and likewise for Make_AR and Make_LD. (For some C compilation systems, these three form a family and must all be "right for each other"). Note the Comment notes that are used to convey a little commentary; with one exception, the system itself does nothing with these notes except leave them alone. In that regard, by and large, the user can add their own notes to an rcd and have them ignored. (The exception is Synop which is happy to print Comments out.)
The FILES description associated with the MAKE GOODS for a PROJECT (or APPLICATION) is used to specify o’s and a’s that are to automatically placed in the link editor’s search sequence. In this case, FILE_BASE daytona_aux associated with this FILES description specifies that the file daytona_aux.c is to be compiled down to a .o and included in the link editor search sequence. Daytona will manage the making of this .o by respecting the usual make(1) dependencies between the .o and the .c and between the .c and the .h. The making is handled by executing $DS_DIR/mk DS <file>.o. The Source note specifies what directory contains the .c file; if there is no Source note given then the directory containing the par is assumed (or aar, in the case of an apd). The TUPLE-valued Depends_On note can be used to specify makefile dependencies of the .o on corresponding target files, which must be specified as complete file paths down from the root. Such files could be any kind of file but .h and .c files would be the most common. A FILE .o can also be specified; in this case, the .o is already assumed to have been made by some other make(1) process. A LIBRARY can also be specified as shown; it too is assumed to have been made and kept up-to-date by some other make(1) process. The syntax used for the TUPLE elements must be acceptable to make(1) as a dependency. For example, $(DS_DIR)/R.sys.h will work and $DS_DIR/R.sys.h will not.

The MAKE GOODS can be of use when the user is extending the system with their own C code. See the end of Chapter 6 for details.

A project description is stored as the pj.d.<project> member of a UNIX ar(1)-variant archive file named par.<project>, as in par.daytona. Daytona uses a modified form of ar(1) called ds_ar. ds_ar enables the user to have pj.d files with names of length up to 80 on systems which support file names upto that length. For those rare 14 character max-file-name systems, project names must not be longer than 7 characters in order to avoid subsequent trouble with file name lengths of certain related files. (see the DS_FNLNAME discussion in the Shell Environment section below.)

The archiver used by Daytona goes by the name $DS_AR: The user can set $DS_AR to be ar, if they wish to use the ar(1) archiver that comes with their system (although caveat emptor).

A $DS_AR archive can be thought of as being the analog of a UNIX directory which would be containing the pj.d file. But of course it is not really a directory: it is a special ASCII file that is maintained in an idiosyncratic format by the $DS_AR archiver. Since the only safe way to modify a $DS_AR archive is ultimately by means of $DS_AR, Daytona provides several utility interfaces discussed later that make it easy to create and manipulate pj.d files as well as apd.s (application descriptions) and rcd.s (record class descriptions).

The shell variable DS_PROJ can be left as the empty string if none of the features associated with the above pj.d information are needed.

An easy way to make a prototypical par for a new Daytona project is to invoke the command Archie as follows:

```
Archie -toc -proj new_project
```

An easy way to make a prototypical par for a new Daytona project that supports a given list of applications is to invoke the command Archie as follows:

```
Archie -toc -proj new_project -apps_for_proj space_separated_sequence_of_apps
```

Archie is discussed more later in this chapter.
3.2.2 Application Archives

A Daytona application archive (or aar) is a UNIX $DS_AR archive that contains one application description and zero or more record class descriptions (rcds) for each of the record classes in the application.

Generally speaking, application archives are not needed if the database-specific portions of Daytona are not used. So, for example, procedural Cymbal requests that make no references to record classes do not need to be processed in the context of any particular application.

The application archive file name must begin with the aar. prefix, the application description file member must begin with the "apd." prefix, and each of the rcd member files must begin with the rcd. prefix. When working the System V archiver, application names cannot exceed 7 characters.

An easy way to make a prototypical aar for a new Daytona application is to invoke the command Archie as follows:

```
Archie -toc -app new_app
```

Archie is discussed more later in this chapter.

3.2.3 Application Descriptions

Here is the application description for the orders application as contained in the archive member file apd.orders:
The MAKE_GOODS are analogous to the ones used for PROJECTs. If a Source note for a data file is not explicitly given in the associated rcd either on the FILE node or the BINS node, then the value of Default_Data_File_Source value, if any, on the APPLICATION node is used instead. If even that is not there, then the directory containing the aar is used. A similar default mechanism is in place for Random_Acc_Bufsize which is described in Chapter 23.

### 3.2.4 Record Class Descriptions: rcds

Here is an rcd for the ORDER record class which is from the sample "orders" application that comes with Daytona:

 #{ RECORD_CLASS ( ORDER )
   <Multiplicity, 0->>
   <__Last_Validated_For_Rcd.UTC, 2001-08-02@04:15:16>
 #}
  BINS
  #{ FILE ( ORDER )
  <__Max_Rec.Len, 77>
  <Unit_Sep, :>
  <Source, ${ORDERS_DATA:=-$ORDERS_HOME}>
  <Comment_Beg, #>
  <__Nbr_Of_Recs, 1000>
  <__Size, 34675> }#
 )#
 #{ KEYS
   #{ KEY ( n )
     <Unique, yes>
     <Field, Number>
   #}
   INDICES
   #{ INDEX ( n )
     <Kind, btree>
     <Unique, yes>
     <__Nbr_Of_Keys, 1000>
     <__Avg_Reach, 1.000>
     <__Btree_Height, 1>
     <__Nbr_Of_Keys_1, 1000>
     <__Avg_Reach_1, 1.000> }#
   )#
   #{ KEY ( sp )
     <Fields, [ Supp_Nbr, Part_Nbr ]>
   #}
   INDICES
   #{ INDEX ( sp )
     <Kind, btree>
     <__Nbr_Of_Keys, 952>
     <__Avg_Reach, 1.050>
     <__Btree_Height, 1>
     <__Nbr_Of_Keys_1, 102>
     <__Avg_Reach_1, 9.804>
     <__Nbr_Of_Keys_2, 952>
     <__Avg_Reach_2, 1.050> }#
   )#
   #
   #{ KEY ( p )

Copyright © 2013 AT&T
All Rights Reserved.
September 15, 2013
<Field, Part_Nbr>
#{ INDICES
  #{ INDEX ( p )
    <Kind, btree>
    <Comment, can’t use MROW if there are several writers!>
    <Simul_MROW, no>
    <__Nbr_Of_Keys, 101>
    <__Avg_Reach, 9.901>
    <__Btree_Height, 1>
    <__Nbr_Of_Keys_1, 101>
    <__Avg_Reach_1, 9.901> }
}
#
#{ KEY ( dr )
  <Field, Date_Recd>
  #{ INDICES
    #{ INDEX ( dr )
      <Kind, btree>
      <__Nbr_Of_Keys, 517>
      <__Avg_Reach, 1.934>
      <__Btree_Height, 1>
      <__Nbr_Of_Keys_1, 517>
      <__Avg_Reach_1, 1.934> }
  }
#
#{ KEY ( dp )
  <Field, Date_Placed>
  #{ INDICES
    #{ INDEX ( dp )
      <Kind, btree>
      <__Nbr_Of_Keys, 511>
      <__Avg_Reach, 1.957>
      <__Btree_Height, 1>
      <__Nbr_Of_Keys_1, 511>
      <__Avg_Reach_1, 1.957> }
  }
#
#{ KEY ( ln )
  <Fields, [ Last_Flag, Number ]>
  <Unique, yes>
  #{ INDICES
    #{ INDEX ( ln )
      <Kind, btree>

<Unique, yes>
<__Nbr_Of_Keys, 1000>
<__Avg_Reach, 1.000>
<__Btree_Height, 1>
<__Nbr_Of_Keys_1, 2>
<__Avg_Reach_1, 500.000>
<__Nbr_Of_Keys_2, 1000>
<__Avg_Reach_2, 1.000> }#
)

#{ FIELDS
#
#{ FIELD ( Number )
  <Type, INT>
  <__Position, 1> }#
#
#{ FIELD ( Supp_Nbr )
  <Type, INT(_short_)>  
  <Validation_RE, "^[1-9][0-9][0-9]+">  
  <__Position, 2> }#
#
#{ FIELD ( Part_Nbr )
  <Type, INT>
  <Validation_RE, "^[1-9][0-9][0-9]+">  
  <__Position, 3> }#
#
#{ FIELD ( Date_Recd )
  <Type, ( 0->1 ) DATE(_yyyymmdd_)>  
  <__Position, 4> }#
#
#{ FIELD ( Date_Placed )
  <Type, ( 0->1 ) DATE(_yyyymmdd_)>  
  <Min_Value, 1789-01-01>  
  <__Position, 5> }#
#
#{ FIELD ( Quantity )
  <Type, ( 0->1 ) INT(_short_)>
  <Max_Value, 32760>  
  <__Position, 6> }#
#
#{ FIELD ( Last_Flag )
  <Type, STR(1)>  
  <Default_Value, 0>  
  <Validation_RE, "^[01]">  
  <__Position, 7> }#
#
There is a lot of information carried by this rcd as well as the apd and pjd. An appendix to this
This rcd begins by saying, in effect, that "there is a RECORD_CLASS meaning ORDER where Multiplicity is 0->". This Multiplicity attribute value merely says that there can be 0 or more records in this object class. (This is usually not a particularly important piece of information; it would be marginally more interesting if the value were 1-> at which point it describes a constraint that could be violated.) The _Last_Validated_For_Rcd.UTC is Daytona-system attribute used to determine when an rcd needs to be re-validated. All Daytona-added notes have attribute names that begin with _ _, i.e., two underscores: the user can simply ignore these notes.

The nested BINS description is saying that there are BINS associated with the RECORD_CLASS ORDER and one of those BINS is a FILE named ORDER which can be found in the Source directory $(ORDERS_DATA:_-ORDERS_HOME) and which has the other indicated attributes. While it is sometimes convenient to have the FILE name be the same as the RECORD_CLASS name, it is certainly permissible for them to be different. One reason to do so is the convenience of using punctuation like periods in FILE names which would be illegal in RECORD_CLASS names. Also, when a RECORD_CLASS has been horizontally partitioned into many FILES, it is common for the FILE names to be different from the RECORD_CLASS name.

Incidentally, if no Source note is given, then the default is the directory in which the aar file is to be found unless overridden by a Default_Data_File_Source note in the apd. As illustrated here, the Source can consist of any file path expressions acceptable to ksh (when available) or the operating system are permissible. This includes $s, ~s, and even back-quote command substitution using shell functions. Ordinarily, the Source note, if given, would not have a value this complicated. A very useful characteristic of the Source note is that it gets evaluated at run time when the query is running, not at compile-time when Tracy is compiling the query. On the other hand, Daytona does not support using shell expressions as the names of FILES or FILE_INFO_FILES.

The Unit_Sep note specifies which character is used to separate field values. If a different separator is desired for use within a LIST/SET-valued field, then a Unit_Seps note would be used as follows in which the ‘top-level’ separator is said to be | and the LIST/SET-valued field separator is said to be :

  <Unit_Seps, |:>

The comment-beginning character is specified using a Comment_Beg note and similarly, the characters that begin and end TUPLES stored in the data are indicated by the Tuple_Delims as in:

  <Tuple_Delims, ^[]^

(The quoting hats are needed for Tuple_Delims because of the special-purpose that brackets ordinarily play in supporting description note LIST relatives.) The ORDER table does not have any LIST/SET-valued FIELDS and hence, no Tuple_Delims note is needed. There are no default Tuple_Delims and so when Tuple_Delims are not explicitly specified by the user in the rcd, then there are none. The only delimiters that have to be specified either by the user or by default are Unit_Sep and Comment_Beg.

Once again, the attributes whose names begin with "_ _" are generated by Daytona and should therefore not be entered by the user. For the ORDER file, the _Max_Rec_Length attribute value says that the longest data file record is 76 characters long and _Nbr_Of_Recs says that there are 1000 data
file records in ORDER. In some rcds such as those associated with horizontal partitioning, entire subtrees (i.e., nested descriptions) are generated by the system. Such subtrees are flagged with a `_System_Generated, yes` note to indicate that users should just ignore them.

So, the RECORD_CLASS is stored in just one BIN, i.e., the FILE ORDER, and it has some KEYS and FIELDS. Talking about the FIELDS description next, each ORDER object record has 7 FIELDS with the indicated Types. The Type values are written using Cymbal syntax for types with the option of prefixing them with Multiplicity indicators.

For example, the Multiplicity indicator `0->1` for the Type of the FIELD Quantity conveys that any given ORDER record may or may not have a value for Quantity, i.e., it is OK for the Quantity FIELD value to be missing. The default value for Multiplicity is 1 which says that there must be exactly one value for the FIELD.

Here is how to specify the LIST/SET-valued FIELD Children for the PERSON RECORD_CLASS:

```cymbal
#{ FIELD ( Children )
   <Type, LIST[ ( 0-> ) STR(*) ]>
   <__Position, 2> )#
```

This says that for any given PERSON record, there can be zero or more values in the Children FIELD TUPLE value. `( 0-> )` is the default here; the other permissible value is `( 1-> )`. These BOXES can be further specified, if desired, by the use of the keywords `with_no_duplicates` and one of `with_lexico_order` and `with_reverse_lexico_order`. Updating processes will cause the specified order to appear in the data file record. Such FIELDS can also have `Min_Value`, `Max_Value` and `Validation_RE` integrity constraints.

The Default_Value specified for FIELD Last_Flag is 0. Whenever a data file record fails to give a value for a FIELD with a Default_Value, Daytona behaves as if the Default_Value was given. The handling of Default_Values varies according to whether the FIELD’s Multiplicity specifies that missing values are allowed or not. Suppose they are allowed, as would be specified, for example, with a `(0->1)` Multiplicity. In order to maintain consistency among the original data, Sizup batch adds, and record-at-a-time adds, whenever a record-at-a-time add fails to provide a value for an optional FIELD with a Default_Value, then no value is placed in that FIELD’s position in the newly added data file record. In other words, if `|` is the Unit_Sep, then the meaning of `||` in the data file record for that FIELD is whatever the value of the Default_Value attribute is for that FIELD in the rcd. (Consequently, once data has been loaded for that table, it is obviously a big mistake to change the semantics of `||` by changing the Default_Value specification. Likewise, it is a big mistake to not have a Default_Value for a FIELD, load a bunch of data, and then add a Default_Value for the FIELD while also making it have a `(0->1)` Multiplicity.)

On the other hand, if missing values are forbidden, then it is clear that any Default_Value cannot be used in a read query as a substitute for a missing value because there can’t be any missing values in the data. Therefore, for record-at-a-time adds, the Default_Value will literally be placed in the FIELD’s position in the new data file record if no value has been specified for that FIELD in the Cymbal add query.

Currently, for an optional LIST/SET-valued FIELD, the only kinds of Default_Values allowed are the empty TUPLE `[]` or a singleton TUPLE, as in:
Note that <Default_Value, [ ]> is not at all the same as <Default_Value, ^[]^>, where the latter value is taken to be the two characters open-square-bracket, close-square-bracket as opposed to the empty LIST/TUPLE, which is the former. In other words, the hat-quotes take away the special meaning of the brackets as indicating a LIST constant.

Notice that the Multiplicity (0->1) at the start of the FIELD Type informs Daytona that Children may or may not have a missing value in a record. For LIST/SET-valued FIELDS of Multiplicity 1 (the default), Default_Value TUPLES of any length are allowed. (Note the other Multiplicity above ( 0-> ) which specifies the allowed lengths of the LIST, given that it exists.) Updates are handled as they are for scalar-valued FIELDS.

Another restriction is that there can be no empty STR/LIT/THING elements of a Default_Value for a LIST/SET-valued FIELD. The reason is that ["""] and [] at the Cymbal level would look the same in the data file, which is just not supportable.

Also note that a empty LIST [ ] value for a LIST/SET-valued FIELD does behave differently from a missing value for that FIELD. When querying using:

```plaintext
there_isa FAMILY where( Children = .kid_list)
```
a missing value for Children in a record will cause that record to be skipped over (as is the case for all missing values) whereas if a data file record contains [], then .kid_list will be set equal to that empty LIST.

On the other hand, both situations result in skipping over the record when the query is:

```plaintext
there_isa FAMILY where( one_of_the Children = .kid)
```
because neither can provide a value for kid.

The Validation_RE specified for the FIELD Supp_Nbr defines an egrep(1) regular expression editing check which Sizup and any Cymbal updates will perform against all values of that FIELD when they do their data validation. (See the discussion of string functions in Chapter 7 for a definition of permissible regular expressions.) The Validation_RE is matched against the string representation of the FIELD value as the Write PROC would print it: consequently, the fact that a FIELD is an INT( _short_ ) or a HEKA is irrelevant for the purposes of checking the Validation_RE.

Since Validation_REs are awkward for expressing bounds on valid values for FIELDS, Daytona also offers the Min_Value and Max_Value attributes for specifying such bounds. For example, attaching a <Max_Value, 123.45> note to FLT Price FIELD description in an rcd would result in the system ensuring that only prices < 123.45 could be stored as Price FIELD values. The Min_Value and Max_Value bounds constraints are enforced during Sizup validation and during record-at-a-time update. For comparison purposes, the attribute values for Min_Value/Max_Value notes are taken to be members of the Type of the FIELD.

The ORDER RECORD_CLASS has 6 KEYS: n, sp, p, dr, dp, and ln . Observe that KEY ln is a
Unique key consisting of the FIELD TUPLE [ Last_Flag, Number ], this clearly being the value of the Fields note. This means that there is at most one object record in the data file that has any given sequence of values for the Fields in the KEY. (This mandated uniqueness within data files only (not within the entire RECORD_CLASS) is discussed further in the context of horizontal partitioning in Chapter 23.) If a KEY has exactly one FIELD, then a Field note (with no brackets) is permissible; in all cases, a Fields note is OK to use. KEYS are usually (but not always) associated with indices. An INDICES description under a KEY description describes each of the INDICES that are to be built to support indexed lookup using the KEY. Any alphanumeric name is acceptable for an INDEX, although in light of possible file name length restrictions, smaller names are preferred and it is usually convenient to give a solitary INDEX the same name as its KEY. The INDEX ln for KEY ln is a unique B-tree index. While a Unique KEY implies that each of its INDICES are Unique, there are cases, such as with cluster B-trees, where an INDEX is Unique and its KEY is not. The KEY dr is a non-unique KEY of the ORDER KEYS. (The default for Unique is no.) This means that any given value of KEY dr may correspond to 0 or more ORDER object records. For FILE ORDER, there are 517 distinct Date_ Recd keys and on the average, each one is associated with 1.934 ORDER object records.

3.2.5 Testing Out Rcds

Suppose one creates one or more rcds and would like to see if Daytona thinks they are valid. Assuming that the rcds are in an aar findable through DS_APPS and DS_PATH, this can be done by calling Tracy as with:

\[
\text{Tracy -val_rcd_for } <\text{recls}>+
\]
\[
\text{Tracy -save_val_rcd_for } <\text{recls}>+
\]

The latter (only) will save rcds back in their aars, most notably after recomputing their __Position notes, assuming that no validation errors have been found. However, note that this saving of rcds will not result in any change in their mod times given in their aars nor in their __Last_Validated_For_Rcd.UTC note; nonetheless, they will have had their __Position notes recomputed as well as any other system-generated annotations.

3.2.6 Naming Conventions

The understandability of Cymbal queries is promoted by using certain naming conventions that allow the syntax of a word to suggest its type and role in the query as a whole. For example, record class names must fall into the UPPER syntactic class, i.e., begin with a capital letter and continue with only capital letters, underscores, or digits (e.g., “REGIONAL_MARKET” ). However, Daytona imposes no naming conventions for objects which are outside of Cymbal; so, for instance, the names of the UNIX files that contain those tables may be any names that are valid in UNIX; however, when working under System V Release 3 and earlier, in order to accommodate various file names generated by Daytona, user data file names should not be longer than 8 characters.

Cymbal attribute (i.e., field) names must be UPLOWs meaning that they consist of letters, digits, and underscores with the first letter being upper-case and some subsequent letter being lower-case (e.g., “Name” or “Child_1” ). Since SQL queries can be written in any combination of upper and lower case letters, a certain convention is used to enable the system to quickly generate field names that
can be found in the application archive (i.e., the data dictionary). This convention requires that fields used in SQL queries have names in the application archive that not only are UPLOWs but also are such that a capital letter follows every underscore: "Supplier_Number" is one such whereas "Supplier_number" is not.

3.3 Keys And Indices

In general, a key is a sequence of one or more FIELDS and a key-value is a sequence of values for the corresponding FIELDS in a key. Indexing enables one to take a key-value and quickly find all records whose key FIELDS have values equal to the key-value. An index is a data structure that implements this mapping from key-value to corresponding records containing the key-value. Currently, Daytona’s indices are all B-tree files provided by Peter Weinberger’s B-tree package [Weinberger, 1981]. Although they are usually never seen by users, each entry in a Daytona B-tree is an ASCII string that is the concatenation of a key-value with a coded form of the disk file offset and length of a corresponding data file record. Common practice is to use the word ‘key’ to mean either the key or the key-value, depending on context.

A key is unique if and only if any key-value for that key is associated with at most one record in the data file. A key is non-unique if and only if any key-value for that key is associated with 0 or more records in the data file. The __Avg_Rach note generated by Daytona for INDEX nodes in rcds quantifies the amount of non-uniqueness for that INDEX: an __Avg_Rach of 4.5, for example, says that on the average, each index entry reaches out or points to 4.5 data records. Daytona enforces the uniqueness of keys defined to be unique in the rcd by generating run-time checks whenever appropriate. It is illegal for the value of any unique key FIELD to be missing; however, it is permissible for the FIELDS of non-unique keys to have missing values and keys for such are stored in the indices.

Daytona automatically creates indices for the keys and indices that are specified in the aar. It is very important to be aware that the judicious specification of keys and hence the creation of indices can make the difference between now and never with regards to getting the answers. If Daytona’s performance is slow, it is most likely because either the best keys have not been defined or because the query has not been structured to take advantage of them (see next paragraph). Keys can make the difference between seconds and hours in query response time. Generally speaking, they must be chosen and they must be chosen well. Complete help on key specification will be given in a subsequent edition of this document. For the moment, an excellent rule of thumb is to specify a B-tree key on every field that will be involved in the likes of an SQL equality join condition. For example, if conditions like "EMP.NBR = DEPT.EMP.NBR" will be used in SQL queries, then keys should be specified on the NBR field in the EMP table and the EMP_NBR field in the DEPT table. For Cymbal, the analogous rule is that if a variable value links two Cymbal descriptions in a query, then a key should be specified for each of the corresponding fields. Also, if queries are frequently written that work with all records that have specified values for a field or group of fields, then that field or group of fields should be defined as a B-tree key.

Daytona automatically selects the best key to use when processing queries. Sometimes though, it can make a mistake. It can certainly make a mistake if the statistics generated by the index builder Sizup and stored in the rcd are meaningless because Sizup was run on an empty data file (this can well
happen in applications whose task is to build data files from scratch). Similarly, there are ways to
prevent Sizup from keeping any statistics on the data: in such a situation, Daytona does not have the
information it needs to make educated guesses as to the best index to use. See Chapter 13 for details
on how the user can require the system to use a particular index.

Some queries ask Daytona to examine records which contain one of several stated values for a key.
Depending on the circumstances, the query may run fastest if a linear scan is used or it may run fastest
if the key’s index is used. See Chapter 13 for details on the Cymbal syntax that can be used to select
either option.

When the query translator Tracy is run with the -FYI option, fyi messages will be run indicating
which indices are being used.

Daytona provides a significant economy when multi-field keys are used in that the index built for
a multi-field key of length \( k \) is actually worth \( k \) indices; e.g., an index built for the key
\([\text{Aa}, \text{Bb}, \text{Cc}, \text{Dd}]\) is actually 4 indices in one, i.e, indices for \([\text{Aa}]\), \([\text{Aa, Bb}]\), \([\text{Aa, Bb, Cc}]\), and
\([\text{Aa, Bb, Cc, Dd}]\).

There is a proviso, however, which is that an implicit index derived from an existing index may
not be as efficient as creating that index independently and separately, although it will of course be
ter than no index at all. For example, if there is no index specified for \([\text{Aa, Bb, Cc}]\), then the index
created above for \([\text{Aa, Bb, Cc, Dd}]\) will be used as if it were also an index for \([\text{Aa, Bb, Cc}]\). But it
may not be as good as using an index created especially for \([\text{Aa, Bb, Cc}]\) and the reason has to do
with how the entries in the index are sorted. Index entries are sorted by key-value and then within
key-value they are predominantly sorted by the file offset of the associated data file record.
Consequently, when retrieving records with given values of \([\text{Aa, Bb, Cc}]\), an index especially created
for \([\text{Aa, Bb, Cc}]\) will for the most part visit the records in the order of their occurrence in the file,
implying a sort of clustering that will not necessarily be present if the \([\text{Aa, Bb, Cc}]\) records were to be
visited in their \([\text{Aa, Bb, Cc, Dd}]\) order in the file.

Incidentally, the size of a B-tree index built for a multi-field key can be reduced by ordering the
FIELDS in the KEY so that the later FIELDS vary more in their values than the earlier ones. This is
because the B-trees use prefix compression on the key values and so if there is little entropy in the first
field (for example), then it will be compressed out most of the time.

Daytona does allow for just one LIST/SET-valued field to be part of a single- or multi-field key.
In this event, for a given record, a key-value is constructed for each element of the LIST/SET-valued
field in conjunction with that record’s values for any other fields in the key. So, for a key \([\text{Children},
\text{City}]\) and a record with Children value \{ \text{Tom, Sue} \} and City value \text{San Francisco}, Daytona
will create two index entries for this key and record, namely, the analogues of \(\text{Tom}\mid\text{San Francisco}
\) and \(\text{Sue}\mid\text{San Francisco}\).

### 3.3.1 Cluster B-trees

A cluster B-tree is a B-tree which associates key values with clusters of records, instead of with
individual records as regular B-trees do. The larger and fewer the clusters get, the smaller the cluster
B-tree gets. In fact, when used in appropriate settings, cluster B-trees get remarkably small. We have
seen cases where a regular B-tree takes up several megabytes at the same time a cluster B-tree pointing
to the same records takes up one or two kilobytes.

Cluster keys can be multi-field but they cannot (currently) contain any LIST/SET-valued field. There can be one or more clusters corresponding to a given cluster key value, where a cluster is defined as a maximal physically contiguous sequence of records all of which have the same values for the fields making up the cluster key. So, if a file of employee data records were sorted by Zip_Code and then Salary_Band within Zip_Code, then the sequence of records corresponding to Zip_Code = 07076 and Salary_Band = "B" would constitute one cluster and would therefore be pointed to by exactly one entry in the corresponding cluster B-tree.

Clearly, in order to take maximum advantage of cluster B-trees, the data should be (perhaps intentionally) pre-sorted on the fields of the cluster key so that the associated clusters are as few and as large as possible. A cluster B-tree is said to be unique if for any given value of the associated cluster key there is at most one cluster containing records for that key value. Of course, only in the degenerate (and not very useful) case would a unique key be associated with a unique cluster B-tree. Declaring a cluster B-tree to be unique results in a small execution savings; consequently, its utility mainly lies in enforcing a uniqueness integrity constraint.

Here is how to specify a unique cluster B-tree in the data dictionary:

```plaintext
#{ KEY ( cwl )
  <Fields, [ Color, Weight_Class, Location ] >
#{ INDICES
  #{ INDEX ( cwl )
    <Kind, cluster_btree>
    <Unique, yes>
  }# }# }#
```

Note the Unique cluster_btree INDEX with its non-Unique KEY.

Not only are cluster B-trees small, when used by queries, they usually speed up execution tremendously since all records with common values are stored together physically. The resulting decrease in disk seek times can greatly reduce query execution times.

Due to the implementation of cluster B-trees, an associated .siz file must also exist. Indeed, for use in certain situations, there is another kind of cluster B-tree that Daytona supports. This is the all_cluster_btree. The difference is in how the two handle singleton clusters, i.e., clusters consisting of a single record. A cluster_btree considers a singleton cluster to be a special case by mapping its B-tree key directly to the offset in the data file, bypassing any use of the .siz file. On the other hand, an all_cluster_btree implements all clusters the same way, which is to say, by mapping each B-tree key to a pair consisting of the number of records in the cluster and the offset in the .siz file corresponding to the first record in the cluster. As a result, an all_cluster_btree supports indexed access which can also (always) provide a value for the argument of `with_bin_pos_vbl`.

### 3.3.2 Other Kinds Of Keys And Indices
3.3.2.1 Non-indexed Unique Keys

In very rare instances, it may be useful to have a unique key which works by a linear search which terminates that search when the key is found. This is accomplished by using a scan-and-quit key as in:

```
#{ KEY ( 1 )
  <Unique, yes>
  <Field, Number> }#
```

Note the absence of any INDEX sub-description since there is no index to be made.

3.3.2.2 Eliminating Unwanted Index Entries

Consider a situation where some of the FIELDS specified in a KEY can have missing values in the data file record (see previous definition). In this case the corresponding KEY FIELD descriptions specify a Multiplicity of \((0\rightarrow1)\) and a missing field value in the data file record corresponds to either a missing value or the Default Value for the FIELD in the corresponding object record. It may be that, for whatever reason, there is no particular interest in using an index to query the table for those data file records with missing or Default Values for one or more FIELDS of the given KEY. For example, with a one-FIELD KEY, it could be the norm that data file records have no value for that KEY; hence, if one queried on that occurrence, the answer would access most of the records of the file, in which case, using an index would be inefficient compared to using sequential access. Or perhaps, it may be that the only queries that the user has any interest in running seek records which have specified non-(missing or default) values for all the FIELDS of the KEY in question. Consequently, the user would benefit by being able to direct Daytona not to put an entry in the INDEX for a KEY for any data file record where one or more of the KEY FIELD values is missing or default. This will make the index file smaller and will thereby speed up access for those data file records that are indexed.

This goal can be easily accomplished by adding a `<Ignore_DFR_Missing_Values, yes>` note to any desired INDEX description:

```
#{ INDEX ( dr )
  <Kind, btree>
  <Ignore_DFR_Missing_Values, yes>
  <__Nbr_Of_Keys, 517>
  <__Avg_Reach, 1.934>
  <__Nbr_Of_Keys_1, 517>
  <__Avg_Reach_1, 1.934> }#
```

This is illustrated in rcd.ORDERA. The default value for Ignore_DFR_Missing_Values is no. What `<Ignore_DFR_Missing_Values, yes>` implies is that if one or more of the FIELDS in the KEY for the INDEX should be missing a value in the data file record, then that INDEX will not have a corresponding entry pointing to that data file record. (An alternative would have been to ignore the data file record if all of its FIELDS had missing values. However if one assumes that the FIELDS are largely independent of each other, then the chances that all are missing for a given record are much smaller than for the chances for any one of them individually to be missing and hence, such a
compound circumstance would probably not tend to occur often enough to warrant it alone being
ignored in the INDEX.)

Caveat: Suppose <Ignore_DFR_Missing_Values, yes> has been specified and that the user
would like to find all records which have some missing values for FIELDS in the given KEY.
Fortunately, Daytona can determine this at compile-time (through Tracy) and therefore, it will use
sequential scan or some other index to compute the answer (and not coincidentally, it will avoid using
the given INDEX which is extremely important to do because there are no keys in the INDEX that will
help to answer such a query!). Unfortunately, when the rcd specifies Default_Values for some of the
KEY FIELDS and some of those Default_Values are asserted to be equal to KEY FIELD values in the
query, then there is no way for Tracy to determine in general at compile-time that records with the
Default_Value for some KEY FIELD are being searched for. Now, if the user specifies a constant equal
to the Default_Value for the KEY FIELD, then Tracy will rule out searching using that KEY and
associated INDEX. Otherwise however if a non-constant, general term is offered, the given INDEX
may well be selected for search, at which point, the query would produce incorrect answers because the
INDEX has no entries corresponding to those records with a Default_Value for some KEY FIELD.
Fortunately, there is a runtime check to catch this problem which will cause the query executable to
abort if the program detects an attempt to search the index using a Default_Value for one or more of
the KEY FIELDS.

3.3.2.3 Banning Indices

Now if the information is to come in from a disk file, then it is best if Sizup is not run on the
data file because that takes time, which is of the essence when dealing with end users. However,
Daytona’s access methods by default assume that there will be at least a .siz index file for them to use
to speed up sequential access. They need to be informed that it is not going to be there. Here is
another rcd fragment illustrating how this is done:
The _Indices_Banned_ note hanging off the KEYS description informs the system neither to build nor to expect any indices whatsoever; this includes not building a .siz file and not building the .+.T free tree used to manage free up record slots for transactions. Note that it is still OK to have a unique KEY specified but just no indices, i.e., the physical data structures that support basic KEY operations -- those are what are banned.

### 3.4 More Performance Considerations

Along similar lines, while Daytona does many optimizations for the user, it does not yet do join order optimization. However, when writing SQL queries, it is well worthwhile to arrange the tables in the FROM clause so that adjacent tables have a join condition in the WHERE clause. See the join performance example in Chapter 13.

The Cymbal analog of this is to arrange conjunctions of Cymbal descriptions so that there is a linking or shared variable between adjacent description conjuncts and so that descriptions of smaller object classes appear first. By ensuring that adjacent tables/conjuncts are linked together with join conditions, the user can avoid highly expensive Cartesian products.

It is also useful to sort a file so that groups of records that are likely to be retrieved together are physically near each other. For example, in the sample _orders_ application, if an important query is to retrieve all orders belonging to a given supplier, then that query will run faster if the ORDER file has been sorted by Supp_Nbr. Sorted data files are also candidates for cluster B-trees and all of the benefits that they bring.

Since the DC format requires that data access routines scan over all intervening characters from the start to reach the _nth_ field of a data file record, Daytona’s speed will increase if the more frequently
used and shorter fields appear closer to the beginning of the record.

Using LIST/SET-valued fields increases retrieval speed. To see this, consider two schemas: [Mother, Father, Children] and [Mother, Father, Child], where Children is a LIST/SET-valued field and Child is not. For the Sue & John family with eight children, to get the names of the children would require eight index searches on [Mother, Father] = [“Sue”, “John”] if Child were used but only one if Children were used.

The data buffer reuse provided by the this is a construct (see Chapter 13) will improve performance over making the same access multiple times.

As an rcd performance consideration, if the user is working with tens of megabytes of data and some of the fields look like integers but are really no more than integer codes which never have arithmetic performed on them, then it may be worthwhile to specify them as strings (i.e., STRs) in their rcd; otherwise, if they are specified as INT, then Daytona will take the extra effort to convert them to integer each time they are referenced which is pointless if they are never involved in arithmetic computations.

Horizontal partitioning can speed access for large amounts of data since the placement of data into many files will result in smaller indices per file than if all the data were in just one file. Also, different data files for the same record class may be placed on different disks thus leading to the use of multiple disk controllers.

When the same executable is likely to be invoked often, performance can be increased by using chmod to set the ‘sticky bit’ on the executable file. This causes the text of the executable to remain in memory once loaded and to be shared amongst processes executing it, thus conserving memory.

For applications which need the very highest update performance or which are constrained by the local maximum number of open files, adding a <Reuse_Freed_Space, no> note to the FILE description in an rcd will cause the system to dispense with maintaining the ‘free tree’ that stores the location and sizes of deleted records so that that space can be reused. If a Reuse_Freed_Space rcd annotation is not present for a FILE description, then a Default_Reuse_Freed_Space annotation will be used instead as soon as it is found by looking at any ancestor FILE_INFO_FILE, BINS, or APPLICATION description.

By using an Indices_Source note on a FILE rcd node, the user may specify that the B-tree and siz indices be located in a directory (the Indices_Source) on a disk not containing the data. Since Daytona indexed access involves scanning one index and the data file, using two disk controllers may substantially speed up processing. Similarly, by placing several data files likely to be accessed in the same query on several disks, multiple disk controllers will be used. A Default_Data_File_Indices_Source note can be placed in the associated BINS description or in the application description for the associated apd. Caveat: if indices are built in the data file directory and then an Indices_Source note is added to the rcd, please remove the no-longer-useful, old indices, especially the siz file, or else Sizup may become confused.

Another option is to add a <Siz_Access, no> note to the KEYS description; this will result in the system not using a .siz file for sequential access (a .siz file will be created but it will only contain a few bytes of status information). (The ability to update <Siz_Access, no> files is not yet implemented.) Of course, .siz files are the default and should only be dispensed with when either file descriptors are limited or disk space is constrained. (The .siz file consists of a header and four bytes for every data file
record slot: those bytes serve to address each data file record. Thus, the .siz file serves as an ordinal index of the data file in that it makes it quick and easy to locate the \ith data file record: this greatly speeds up sequential access because it enables Daytona to avoid having to read each character on its way to the end of each record, as it looks for the newline to tell it that the end has been reached -- and that the next record is just a byte away.)

To take advantage of symmetric multiprocessors, Sizup can be caused to clone itself a specified number of times, with each clone working on its share of the total job in parallel.

By attaching a \texttt{Seq\_Acc\_Bufsize} or \texttt{Random\_Acc\_Bufsize} note to the FILE description of the relevant rcd, the user can specify a large enough buffer to contain the entire data file in the user-process’s memory space. See Chapter 23 for details.

In terms of what not to do, don’t build an index on everything or to put it differently, have a good reason for requesting any given index. Paradoxically, too many indices can clutter up the disk to the point that performance is degraded for any one index individually (this has been observed in practice). Also, it takes time to build and maintain the indices and each index will take up a file descriptor during update transactions.

Also, if a record class has only a few records in it (like less than 100 or so), then it will probably slow things down to specify B-tree indices for it. Instead, the user can specify Unique KEYS that have no B-tree index associated with them: Daytona will then search for a unique keyed record by scanning the file until it finds the one it is looking for and then terminating the scan. Even when Daytona scans such a file in its entirety, it can be faster than it would take to read in the root block of a B-tree or to open a B-tree file for update totalled over each B-tree index specified. Running a test for your application will help you decide whether to have B-tree files or not for your small record classes.

### 3.5 Space Efficiency

The size of a DC data file can be reduced in a number of ways. For example, suppose a file has many records whose value for the same field is almost always the same, like, say, 0. Then all those common values can be eliminated by using a Default\_Value.

The compression-oriented types HEKA, HEKSTR, HEKINT, ATTDATE, and others typically cut the space required by a field in half. This is just field-level compression. Record-level compression is described in Chapter 23.

### 3.6 Shell Environment

In order to create and modify rcds and aars and in order to process queries with Tracy and Stacy, it is first necessary to establish the appropriate shell environment so that the relevant Daytona software can be found and accessed. If the Korn shell is the login shell, then the first step is to modify the user’s .profile by including the command

\[
\texttt{. \$X/DS\_Set}
\]

where \$X evaluates to the directory where Daytona has been installed. (The \$X is necessary if DS\_Set is not already in the user’s PATH.) Because DS\_Set modifies the user’s PATH variable by appending \$DS\_DIR, its invocation must appear after the initial definition of the PATH variable and so the best
place to put it is at the end of the user’s .profile (but before any automatic window creation commands). *The dot that begins this command is absolutely necessary so that the necessary shell variables will get set in the current shell.* If the user has set +allexport, then an unnecessary number of DS_variables will be maintained in their shell environment. On some systems, users can mandate that ksh is their login shell by executing the command `chsh` or some variant of the `passwd` command; if this is not possible, then the system administrator must make this change. If ksh is not the login shell, then the user must get into ksh after logging in and then execute a `DS_Set` command (using the preceding ".").

The current values for all of the Cymbal environment variables may be displayed by executing the command `DS.Env`. (To display the value of just one environment variable, give `DS.Env` an argument as in `DS Env DS_REL` to display the Daytona release identifier.) The user is welcome to change as desired the variables in the upper section of the output of `DS.Env`, namely `DS_PROJ`, `DS_APPS`, `DS_PATH`, `DS_FLAGS`, `DS_RFLAGS`, `DS_TFLAGS`, `DS_ZFLAGS`, `DS_SFLAGS`, `DS_CFLAGS`, `DS_LDFLAGS`, `DS_KSHON`, and `DS_SQLONLY`. These variables can be changed at any time and are set and changed like any other shell variables are. They can also be changed with a command like this:

```
. DS_Set "one_or_more_variable_definitions"
```

(There will be no need for an absolute path for `DS_Set` here if the system can find `DS_Set`, as it can if `DS_Set` has been previously dotted, since then `$DS_DIR` has been appended to `$PATH`.)

Typically, the first invocation of `DS_Set` will give values to `DS_APPS`, `DS_PATH`, and perhaps `DS_PROJ` as illustrated by:

```
. X/DS_Set " DS_PROJ=billing DS_APPS='orders:misc' DS_PATH=.:"xxx/app_10:"HOME"/d "
```

The value of `DS_PROJ` is a single alphanumeric identifier indicating the current project, where the empty string is the default. Users may also wish to specify the value of the `DS_APPS` variable at this time, especially if not specifying `DS_PROJ`. `$DS_APPS` is a blank or colon separated list of applications that the user would like to have accessible in their current environment. Think of them as the active or current applications. If `$DS_PROJ` is also specified, then each application mentioned in `$DS_APPS` must appear as a value of the `App` attribute in the corresponding `pjd` or else warning messages will appear; on the other hand, it is permissible for `$DS_APPS` not to mention every application for a project, if `DS_PROJ` is specified. Note the pattern of single and double quotes in the above: this portrays the most fool-proof way to correctly handle the most general case where embedded spaces may appear in the value of a shell variable.

Alternatively, if `$DS_PROJ` is specified, then when `DS_Set` is run with `DS_APPS` said to be equal to `...`, then `DS_APPS` will be expanded into a colon-separated list of the applications specified in the corresponding `pjd`.

When queries are processed, the aar files for all applications mentioned by `$DS_APPS` will be consulted, in the order of their appearance in `$DS_APPS`. `$DS_APPS` is also used to help find rcd files for editing as described before.

In order for Daytona to know where to search for the aar files corresponding to referenced applications, the par file for the project (if any), and other environmental files, the `DS_PATH` variable should be set to be a colon separated list of directory paths. By default, `$DS_PATH = "."` (i.e., the current directory).
As it turns out, many of the variables shown below the blank line in the DS Env output are not shell variables at all. They used to be but are no longer. Their values are kept in the file DS.sys.env.sh in $DS_DIR where the Daytona system executables are kept. Their values can be retrieved programmatically by using getdsenv as in getdsenv DS_BTDIR. These values are set at installation time and generally remain immutable after that. If you have permissions sufficient to write this file, and you wish to change some (like DS_MAKE or DS ACROREAD) and you want your change recorded in DS.sys.env.sh for future sessions, then execute a command like:

```
.DS_Set " DS_MAKE='dmake -j 14' DS_INSTALL=y 
```

Note how the single quotes for the DS_MAKE value will preserve the embedded spaces.

Daytona consults the shell environment before DS.sys.env.sh and so, per-user, per-session overrides of the DS.sys.env.sh variables can be achieved by changing one’s shell environment on a per-shell basis as desired -- but only for the shell variables listed in the for statement at the top of the DS Env script. Here is how to do it for the remainder of the current shell’s lifetime:

```
.DS_Set " one_or_more_variable_definitions "
```

Here is how to do it for one command only:

```
DS_TFLAGS=-TC DS Compile test.Q
```

In rare circumstances, . DS_Unset may be used to remove all Daytona shell variables from the user’s (current) shell environment by running the ksh ‘unset’ command on them.

Regarding DS_FLAGS and DS_RFLAGS, et al., using " DS_FLAGS=-FYI " will cause Daytona to print more informative messages than it usually does (FYI means "for your information"). Using " DS_FLAGS=-W " or " DS_FLAGS=-WARNING " will cause Daytona to print warning and error messages only; this is the default. Warning messages are intended to alert the user to probable errors. If "DS_FLAGS=-E" or " DS_FLAGS=-ERROR " then only error messages will be printed; this can be fairly dangerous to do since it will cause messages that indicate probable errors (i.e., warnings) to be discarded. If DS_FLAGS contains -BELLs, then Daytona executables will begin beeping at you when they finish.

DS_RFLAGS can be any space-separated sequence of +FYI, +W, +WARNING, +E, +ERROR, +U, +R, +S, +NS, +T, +NT, and +DHO. Saying

```
DS_RFLAGS="+NS +T"
```

causes user executables to send fyi-style messages to stderr (+NS) and to forego checking for up-to-date indices (+T or +trustme). +FYI, +W, +WARNING, +E, +ERROR are the analogs of their − counterparts. +U signifies the usage feature described later. The value of DS_RFLAGS is inserted into argv of Daytona-generated executables just after argv[0], which is the program name. +NT or +notrustme are the opposite of +T and will cause Sizup to be invoked to check if the indices are up-to-date (+NT is the default). +T should be used only after careful deliberation: if in fact the user is wrong and the indices are out of date, then the query, in its attempt to use these invalid indices, may very well produce very strange behavior and not give any good clue as to why. To specify for a given record class that no query is to ever automatically run Sizup on any indices for this class, just append a
<Trust me, yes> note on the KEYS description; this is of value when using directory-based horizontal partitioning. +S means silence and +NS means not silent: when Daytona is not being silent, it will send various informative messages to stderr. The default is for Daytona to be silent. The +R option causes the executable to report on the circumstances of its creation: these include the path to its source Cymbal or SQL file, the DATE_CLOCK when its C code was created, the value of DS_REL when its code was generated, and if that DS_REL value disagrees with the current DS_REL value, then a warning message is printed and the program exits with exit status 1. The +DHO, standing for Display_heading_only, is an option used by the Daytona implementation to run a query executable in such a way that any Display call will result only in printing its heading and furthermore, no included transaction PROC will be run. The initial ‘+’ was chosen instead of ‘−’ so as to allow the user complete freedom to choose whatever ‘−’ command-line options they want without fear of collision with any Daytona command-line options.

DS_ZFLAGS contains arguments to Sizup that the user wishes Sizup to always have when invoked via prompting.

DS_TFLAGS contains arguments to Tracy or Stacy that the user wishes Tracy or Stacy to always have when invoked via prompting.

DS_SFLAGS provides arguments to Squirrel, the SQL parser, whenever it is called by Daytona commands (see Chapter 4).

DS_SQLONLY=y mandates that all queries processed by Tracy invoked via prompting will be 100% DSQL, i.e., with no procedural Cymbal preceding or following DSQL statements. This would allow the queries to use SQL-style comments and parenthesized unions without further ado. This DS_SQLONLY option is only rarely used: Tracy can almost always figure out when DSQL is being used and in that event, do the right thing by it. (One of the ways it does that is by assuming that a pure SQL query is contained in any file whose name ends with a .S suffix.)

DS_KSHON takes values y or n depending upon whether ksh (as specified in DS_KSH) is being used or not.

DS_FLNAMEMAX takes values between 14 and 80 and is considered to report the maximum file name lengths for the UNIX platform at hand. There is no need to set DS_FLNAMEMAX for SUN, Linux, or HP platforms or for platforms where you are content with a 14 character limit. When DS_FLNAMEMAX is greater than 14, it is necessary for ds_ar to be $DS_AR.

The DS_CC variable specifies the path to the default C compiler that Daytona will use to compile the C code it generates. The DS_CFLAGS variable defines CFLAGS for the C compilation of query programs. The DS_LDFLAGS variable defines LDFLAGS for the C compilation of query programs. The DS_LDLIBS variable defines libraries to be searched by ld(1) during the C compilation of query programs; when setting this using DS_Set, be sure to preserve the libraries that the system has seen fit to request.

The DS_CPP variable value is the path of the desired C pre-processor for C++ to use when compiling Daytona-generated C++-compatible C code. It must be used in conjunction with Cpp_Path as described earlier in this chapter.

Daytona now supports the user specifying their own make(1) invocation by executing the likes of:
. DS_Set 'DSMAKE="gmake -j 4"'

or

```
export DS_MAKE="whatever"
```

gmake is GNU make which has the effect of distributing the making of Daytona executables among several CPUs. Note the elaborate, fascinating but necessary use of quotes above to protect the -c from confusing the shell.

### 3.6.1 Shell Environment: Where The Examples Are

The data and aar for the orders application are contained in $DS_DIR/EXAMPLES/usr/orders as are a number of sample queries in $DS_DIR/EXAMPLES/usr/orders/Q. The user should feel free to copy the orders directory and to try running some of the sample queries against the sample data (after, of course, building the indices as described below). In fact, please use the "Getting Started With Daytona" tutorial to help you in this adventure. Some of these queries are part of the Dayton installation test suite and are so marked; they are designed to exercise the system in creative ways and are therefore not intended to make much sense in and of themselves.

### 3.6.2 Shell Environment: Faking It

Sometimes users wish to use an executable written in procedural Cymbal on a machine where there is no Daytona installed. Since Daytona executables like to "phone home" by evaluating some Daytona shell variables, (simple) special measures need to be taken to fake the executable out. All that is necessary is to make sure that the shell environment that the executable runs in has exported DS_DIR="". (Note that this is not the same as saying that DS_DIR is unset; someone will have to have explicitly defined DS_DIR to be the empty string.) When an executable is run under these circumstances, the following warning will be printed:

```
warning: found DS_DIR="" in shell environment;
will assume that the Daytona DS_DIR shell env is not needed and hope for the best.
```

### 3.7 Daytona Commands

```==>
The command DS by itself will show the user what Daytona commands are available.
```
% DS
commands to try:

(to get usage help for any command, invoke with a -? or +? option, whichever works)
(commands invoked using DS with no arguments usually offer interactive prompting)

DS_Set sets the Daytona shell environment
DS Env [vbl] displays the Daytona shell environment
Dice the EASEL menu/screen interface to Daytona
Daisy the Daytona interactive SQL DML and DDL processor
[DS] QQ quickly edits & runs queries; few options
DS Expr quickly evaluates a Cymbal term
[DS] Tracy/Stacy translates Cymbal/SQL into C and calls DS Exec; many options
DS Exec makes and/or runs executables from C; many options
DS Compile just compiles queries to executables; no options
DS Mk "DS Mk <name>" makes executable <name> from DS code
DS Edit supports editing rcd.*, *.Q, *.S, etc.
DS Vi/Vu supports vi editing/reading rcd.*, *.Q, *.S, etc.
DS Emacs/Emucs supports emacs editing/reading rcd.*, *.Q, *.S, etc.
DS Joe/Jou supports joe editing/reading rcd.*, *.Q, *.S, etc.
[DS] DC-rcd creates a record class description (rcd) for data
[DS] Synop provides synopses of application and project archives
[DS] Synop_fpp provides synopses of functions/predicates/procedures
find-it tools: ar_of, ar_or_env_fl_for, env_fl_for_fpp, ds_whence
[DS] Archie performs archive maintenance
DS Show displays phys/virt tables, data files or executable output
filters: DC-pkt DC-prn
DS Mk_DE makes data entry programs from rcds
[DS] Sizup validates and builds indices for data
DS Msgmrg merges Sizup.msgs with data
file tools: Get_Lines, Delete_Faults, Trunc, Split
find lockers: Lock_Blockers_For_File_Path, Lock_Blockers_For_Shmem
IPC info: Show_Shmem, Rm_Shmem, Describe_Shmem, Show_Sem, Rm_Sem, Describe_Semid
[DS] Check_DC_Lines does a quick sanity check on a DC data file
[DS] Check_Indices cross checks indices against their data for a RECORD_CLASS
[DS] Checkup checks that the metadata (pjd/apd/rcd) is consistent with what it describes
DS Resync regenerates RECORD_CLASS I/O files and other objects and/or indices
DS Relocate relocates _System_Generated pjd/apd Source values to new $DS_PATH
DS Clean removes RECORD_CLASS I/O files and other objects and/or indices
DS Clean_Misc removes miscellaneous DS-generated files
DS Basics the Daytona Basics book presented via acroread
DS All_About the All About Daytona reference manual presented via acroread
DS Tutorial the Hands-On Tutorial presented via acroread
DS Man DS Man lists topics; DS Man <topic> provides the associated man page
DS White_Paper updated SIGMOD conference paper presented via acroread

Copyright © 2013 AT&T
All Rights Reserved.
September 15, 2013
Think of DS as meaning Daytona Shell. Most of the command-line oriented DS Commands can be invoked with the -? or +? option to provoke them to display their usage.

### 3.7.1 DS Commands: Documentation

This All About Daytona reference manual can be viewed in Acrobat PDF format by executing DS All About. DS Tutorial will invoke acroread to display the hands-on tutorial, DS White Paper will invoke acroread to display the Daytona white paper, presented at the 1999 SIGMOD database conference. The command DS Man will show what man pages are available, which can then be viewed with the likes of DS Man Tracy. DS Course will invoke acroread to present the vugraphs comprising the Daytona course.

All of the documentation is located in $DS_DIR/DOCS .

### 3.7.2 DS Commands: Displaying Data

The DS Show command is used to print out portions of record classes, entire data files, or executable output. Simply invoke it and answer the questions. When the record class option is chosen, DS Show will use DS QQ to compile and execute a query to print out the requested records in any of the table, packet, Cymbal description, XML, safe, or unfiltered formats. Obviously, the data dictionary is used here to locate and work with the data files belonging to a record class. Furthermore, the user can optionally provide the contents of a Cymbal there isa where argument to constrain the output according to that condition. Since the heart of the freshly generated query that DS Show runs uses the Cymbal Describe procedure, its approach to missing field values in data file records is not to skip over the record when it encounters them but rather to just avoid printing anything having to do with any missing value. Thus DS Show is the closest to the SQL SELECT * WHERE that Daytona offers. Describe is described in Chapter 13, where in addition its treatment of Default_Values is explained.

If DS Show is asked to work with a single data file, then the same output formats are available but the user will have to provide the path to the file as well as the unit separator, comment character, and other related syntactical details. When asked to work with data files (as opposed to record classes), DS Show will demonstrate that it does not know how to display plaintext values of FIELDS having
compressed types like HEKA, nor does it know how to work with files written using record-level compression. Such capabilities are available only when DS Show works with a record class. DS Show will also directly display the output of executables: just give it the name of the executable and various other syntactical specifications.

Note that DS Show can also be called with arguments so as to display (portions of) record classes, whether conventional disk tables or views:

**usage:** DS Show [ <recls> [ <there_is_a_where> ] [ <p|t|d|x|s|u = t> ] ]

**example:** DS Show PARTCU 'Intro_Date' `1982-01-01` and Color Matches "'bl'' u

The optional third argument with values p|t|d|x|s|u and default t specifies one of the formats known to the Cymbal Display procedure as _packet_, _table_, _desc_, _xml_, _safe_ and _data_ (i.e., unfiltered), resp.

Since DS Show is a shell interface to DS QQ, DS_TMPDIR controls where any temporary files are created.

Daytona also provides a full spectrum of file reformatting filters. Each filter will either convert files in any of the popular micro formats (such as Lotus) into the comma-separated value (CSV) format or will convert in the other direction from CSV to whatever. In particular, there are filters which will convert back and forth between CSV and DC format. Two filters come with Daytona, namely, DC-prn and DC-pkt. Contact a Daytona developer for directions on how to get the other filters.

### 3.7.3 DS Commands: Creating And Editing Application Archives

Returning to the discussion of aars proper, the easiest way to create one and the rcds that go in it is to use the DS DC-rcd command to generate an rcd that describes a given data file. Just follow the prompts and the rcd will be created. One strong advantage of using DC-rcd is that it will attempt to infer the datatypes and sizes of the fields in the particular data file being processed. DC-rcd also guesses that the first field would make a good key but of course, it has no way of knowing. So, it is necessary for the user to modify the rcd, making sure that the appropriate keys are specified. DC-rcd will take field names from a msg)flds comment if it is present. Also, it is useful to remember that DC-rcd is happy to work with as few as one, presumably prototypical, record.

The procedure for modifying an rcd depends on where it is. One of the prompts that DS DC-rcd gives allows the user to capture the rcd either in a regular UNIX file or to place it in some specified aar archive. rcds that are plain UNIX files can be modified directly by using *vi*. On the other hand, rcds in aar archives must not be modified by using *vi* directly on the aar archive: *vi* has no way of respecting the integrity of the internal format of $DS_AR archives. In fact, the rcd must be gotten out of the aar archive before *vi* can be used on it.

Now, in this regard, Daytona provides a general purpose archive manipulation facility which is accessed by the command DS Archie. Archie allows the user to find out information about what is in an archive, to put files in and to get files out of an archive, and to rearrange an archive. Archie is truly general purpose in that it works on any ar(1) archive in addition to providing some special services for par and aar archives, which themselves are built using the special Daytona version of ar which is called ds_ar. If Archie creates temporary files for its own private use, it creates them in /tmp by default, or else in $DS_ARTMPDIR if DS_ARTMPDIR is set. When executing user-specified -Checkout and -Return,
the corresponding pj’d’s, ap’d’s, and/or rcd’s are created by default in the same directory as the archive (par or aar) containing them or else in $DS_ARTMPDIR if DS_ARTMPDIR is set. Also, when other Daytona system programs like Tracy have a need to create temporary pj’d’s, ap’d’s, and/or rcd’s, they are created by default in the same directory as the archive (par or aar) containing them or else in $DS_ARTMPDIR if DS_ARTMPDIR is set.

So, in principle, the user could use Archie to get an rcd out of an aar archive and then edit that temporary, standalone rcd file using vi; on completion of the edits, Archie would then be used to replace the rcd in the aar archive. Fortunately though, Daytona provides the DS Edit utility to take care of all the Archie details involved in editing a apd’s and rcds contained in an aar archive and pjd’s contained in par archives. (DS Vi and DS Emacs are both editor-specific interfaces to the editor DS Edit which itself uses the user’s $VISUAL editor, $EDITOR editor, or vi, whichever exists first. If emacs on the user’s system is known by some other name (such as gmacs), then set and export the $EMACS variable to be this other name and DS Emacs will choose the right one.) For specificity, DS Vi is discussed below but similar comments hold for DS Emacs and DS Edit.

DS Vi is sensitive to several Daytona name prefixes. In particular, DS Vi provides special services for pj’d, ap’d, and rcd files. For example, by saying DS Vi rcd.SUPPLIER, Daytona will use the user’s Daytona shell environment to search for an aar containing rcd.SUPPLIER and having found it, will get a lock on the rcd (thus preventing others from changing it), will automatically extract the rcd from its archive into a temporary file, will enable the user to edit this temporary file using vi, and on completion of the edits, will put the modified rcd back in the archive and will remove the lock and the temporary file. All of this background activity is hidden from the user: they just issue the DS Vi command, make their changes using vi, and exit vi as usual. Unless explicitly over-ridden, DS Vi searches for pjd’s, apd’s, and rcd’s in archives associated with the applications listed in $DS_APPS.

If DS Vi -env name is invoked, then the *.env.cy files are searched for the first occurrence of define."\<name\>" so that the editor can open up centered on that line. This use of DS Vi will even handle a sequence of names.

If DS Vi is invoked with the -dmtm option, then the modification time for any archive member will not be changed. This should be used with caution since it will defeat Daytona’s attempts to synchronize files according to their timestamps.

If DS Vi is invoked with the -EK option, then the rcds will be presented for editing in the English-keyword format; they are stored in the pound-brace format and so a little extra time is needed to convert them to the English-keyword format.

Changing the modification time of rcds causes Daytona to take steps to ensure that the entire system remains synchronized. Since this may involve the time-consuming rebuilding of indices, it is important to avoid changing the rcd unnecessarily. By saying DS Vu or DS Emacs, the user can gain read-only access to files, which enables multiple users to view an rcd for reading simultaneously and which prevents any change in the modification time of the rcd. When DS Vi is used, Daytona will only change the modification time of an rcd if changes have actually been made to it.

3.7.4 DS Commands: Producing Data Dictionary Reports

The command DS Synop displays selected portions of the data dictionary in a compact,
convenient form. Options control what information to extract and print and where to find it. In terms of what to print, by using the -pjd, -apd, and -recls options, one may display synopses of project descriptions (pjd$s), application descriptions (apd$s), or more commonly, record class descriptions (rcd$s) or view definitions. By default rcd synopses contain the names and data types of the fields for each chosen record class, and the names and locations of the data file or files that contain the data. For example, if the user has set their Daytona environment to point at the sample project daytona that comes with Daytona (which contains the applications orders and misc), then the command

```
DS Synop -recls ORDER
```

will result in the following output:

```
-- RECORD_CLASS: ORDER
     FILE: ORDER
     Source: $(ORDERS_DATA:-"john/d)
     FIELDS:
     Field 1: INT Number
     2: INT(_short_) Supp_Nbr
     3: INT Part_Nbr
     4: DATE(_yyyymmdd_) Date_Recd
     5: DATE(_yyyymmdd_) Date_Placed
     6: INT(_short_) Quantity
     7: STR(1) Last_Flag
```

The user can cause additional information to be included or omitted in the synopsis by using the -keys, -stats, -bins+, -bins- or -fields+ options. One may request synopses of individual rcd$s, or for the full set of those belonging to some application or even all the applications for a given project. Here is the usage for Synop:
usage:  DS Synop
       -proj  <project>
       -app(s)  <applications>
       -rec(ls)  [<record_classes>]
       -apd
       -pjd
       -toc
       -keys
       -stats
       -fields+
       -bins+
       -bins-
       -phys_only
       -virt_only  |  -view_only
       -proj_only
       -usr_only
       -all

Like other Daytona tools, Synop uses the environment variables DS_PROJ, DS_APPS, and DS_PATH to locate data dictionary information. The user can override DS_PROJ and DS_APPS by using the command line options -proj and -apps, resp. When invoked without any arguments, Synop interactively leads the user through the available choices. For more information about Synop, see the manual page in Appendix D.

3.7.5 DS Commands: Finding Information On fpps

Synop has a sibling which prints out prototype information on fpps:

% DS Synop_fpp -name Print_Str1_Matrix
# FILE  daytona.env.cy  (/l/alex11/rxga/day/DS.g/d)
PROC  :  Print_Str1_Matrix( alias LIST[ 0-> ) TUPLE[ STR(1), INT(_long_), INT(_long_) ]
             :  with_no_deletions  with_no_duplicates  with_default_arbitrary_order | .my_alias_matrix )

Here is the usage:

usage:  DS Synop_fpp
       [  -proj  <project>  ]
       [  -app(s)  <applications>  ]
       [  -name(s)  <fpp_name_REs>  ]
       [  -toc  |  -compact  |  -full  ]
       [  -fun_only  |  -proc_only  |  -pred_only  ]
       [  -sys_only  |  -proj_only  |  -usr_only  ]
       [  -add_imports  ]

DS Man  Synop_fpp  will  print  out  additional  information  about  using  this  command.

Copyright © 2013 AT&T
All Rights Reserved.
September 15, 2013
3.7.6 DS Commands: Finding Where Stuff Is

`ar_or_env_fl_for` will print out the complete path of the environment file that contains the definition of the `rcd` for its `RECORD_CLASS` argument, which could be either a view or a regular disk-based table. `ar_of` does likewise but only for regular disk-based table `RECORD_CLASS`es. `env_fl_for_fpp` will print out the complete path of the environment file that contains the definition of its `fpp` argument. `ds_whence` prints out the path of the directory in `$DS_PATH` which contains its file argument. Here are some examples:

```
ar_or_env_fl_for DYNORD_P
ar_of SUPPLIER
ar_of SUPPLIER 2.0
env_fl_for_fpp ingres_date
ds_whence aar.orders
```

`DS Man find-it` will print out additional information about using these commands.

3.7.7 DS Commands: Project And Application Synchronization

In general, as Daytona processes queries in the context of the primary specifications contained in project and application archives as well as `*.env.cy` and for that matter, the data files themselves, it creates a variety of derived artifacts. These artifacts include indices, the so-called `fio` C files (for File Input/Output), compiled packages, various `.o`'s and executables. Changing the primary specifications can invalidate any or all of the derived artifacts unless those previously existing derived artifacts are resynchronized with their corresponding primary specifications by recreating said artifacts. Failure to do so, i.e., continuing to use out-of-date derived artifacts, can become exceedingly detrimental, as occurs, for example, when they are included as is into new executables and thereby cause `make` or execution errors or when old executables are run against data whose layout/indices have changed, thereby causing query aborts or incorrect answers. One common scenario leading to desynchronization occurs when a new Daytona release is installed.

Daytona determines synchronization or the lack of it solely by comparing modification times between primary and derived items: the derived items have to be newer than the primary items that they depend on. So, for example, `.siz` (and all) B-tree index files have to be newer than their data files and their `rcd`'s. (Note that while the modification times of `pj`d's, `apd`'s, and `rcd`'s are monitored closely by Daytona, Daytona takes no notice of the modification times of the archives themselves that contain these primary items.) Obviously, Daytona minimizes desynchronizations by managing the modification times appropriately when Daytona tools are used to work with the various items; when users go outside of Daytona and inadvertently or deliberately change modification times on their own, they risk causing desynchronization and its attendant misfortunes.

3.7.7.1 DS Commands: Resynchronizing Projects And Applications

To get a project, if any, and associated applications back in sync again unconditionally, use `DS Resync`. In particular, to sync up just one application, execute `DS Resync -i -o <app>` to cause
Daytona to recreate all of its derived files for application `<app>` and use `DS Resync -i -o -` to sync all of them up (over the list in `$DS_APPS`). (Note that the `-i` option will cause all indices to be rebuilt which could be very time-consuming and disruptive -- and may not be necessary: Daytona support can inform you as to whether index rebuilding is necessary, which it only rarely is. In general, the `DS Resync` usage is:

```
DS Resync [ [ -i(indices) | -o(objects) ] ] -cyd(_mod_tm) | -rm_sysgen ]
[ -create | -truncate | -clean_slate ] [ <app_seq> | - ]
```

At least one of `-i` or `-o` must be specified, unless one is using `-cyd_mod_tm` or `-rm_sysgen`. A - instead of a colon-or-blank-separated `<app_seq>` is equivalent to all the applications in `$DS_APPS`. By default, whether or not any applications are specified or implied, if `$DS_PROJ` exists and `-o` is specified, then all of the objects associated the project `<proj>` are recreated. These consist of `<proj>.env.o`, `<proj>.o`, `<proj>.4c, and the package LIBRARIES and the `*.o` that are specified using FILE_BASE under the MAKE_GOODS subtree in the corresponding pjd. In certain situations, this default processing of project-related files will result in error messages if the user has the UNIX permissions to modify application files but not project files. To get `DS Resync` to work in this situation, use a command like:

```
DS_PROJ="" DS Resync -o myapp
```

If applications are specified, then application resync'ing is done in addition to any project resync'ing. The `-i` option causes all record class indices in the designated applications to be redone. The `-o` option causes all Tracy-generated RECORD_CLASS I/O .c and .h files (fio files) to be generated and compiled again. In addition, all of the system-maintained `*.env.c` are recompiled as well as the `*.c` that are specified using FILE_BASE under the MAKE_GOODS subtree in all corresponding apd’s and pjd. Lastly, under `-o`, all of the system-maintained `*.env.cy` are reprocessed into their corresponding `*.env.4c` as well as this causing all packages imported into these `*.env.cy` to be recompiled. In short, under `-o`, all of a project’s and specified applications’ derived code files are recreated. Any `-create`, `-truncate`, or `-clean_slate` options are passed on to any Sizup call generated by Resync (due to `-i`) where they are considered to appear after `$DS_FLAGS $DS_ZFLAGS`.

The `-rm_sysgen` option is used in those rare instances when it is desired to remove all of the `__System_Generated` LIBRARY and FILE descriptions under MAKE_GOODS in the pjd and and specified apds: these are typically those LIBRARY descriptions Daytona creates to record Cymbal package dependencies. (Of course, the user needs to have and to specify applications in order for `-rm_sysgen` to do its work on apds.) `-rm_sysgen` can be used in conjunction with `-o` and/or `-i`. If `-rm_sysgen` is specified along with `-o`, then the `-rm_sysgen` activity is done first.

The `-cyd_mod_tm` (for Cymbal description mod time) option causes the modification times of all pjds, apds and rcds to be made equal to the current time; obviously, that will certainly cause previously generated artifacts to become out-of-date. `-cyd_mod_tm` cannot be used when either `-o` or `-i` are being used.

`DS Resync` is a good thing to do in response to hearing that a new Daytona release has been installed:

_Caveat: The most common way by far for users to cause Daytona to generate error messages is to attempt to work with a new Daytona release in an environment containing derived files generated by a previous release of_
Daytona. So, whenever a new release of Daytona is installed on your system, please run DS Resync on all of your applications (and projects). Usually, you will not have to rebuild the indices and so usually, running

```
DS Resync -o -
```

will be fine. Check the release notification message you receive from the Daytona support team to see if index rebuilding is necessary, which would only rarely be the case. You should run this Resync command using nohup so that you can catch all the output and grep it for errors: there should not be any errors and so if you find some, please contact the Daytona support team.

To simply remove all of the index files and RECORD_CLASS I/O (fio) files for an application, execute "DS Clean <app> -i -o". In general, DS Clean usage is:

```
DS Clean [ -i{ndices} | -o{bjects} ] [ <app_seq> | - ]
```

At least one of -i or -o must be specified. A - instead of a colon-or-blank-separated <app_seq> implies all the applications in $DS_APPs. If there are no applications given or implied, then nothing happens unless -o exists or is implied and $DS_PROJ exists, in which case, all of the objects associated the project <proj> are removed. These consist of <proj>.env.o, <proj>.o, <proj>.4c, all Cymbal package (__System_Generated) LIBRARY descriptions from the pjd and their corresponding *.a, as well as the *.o that are specified using FILE_BASE under the MAKE_GOODS subtree in the corresponding pjd.

The -i option causes all indices in the applications specified to be removed. The -o option causes all Tracy-generated RECORD_CLASS I/O .c, .h, and .o files to be removed for all applications specified. In addition, all of the system-maintained *.env.o are removed, all Cymbal package (__System_Generated) LIBRARY descriptions and their corresponding *.a are removed as well as the *.o that are specified using FILE_BASE under the MAKE_GOODS subtree in all corresponding apd’s. Lastly, under -o, all of the system-maintained *.env.4c are removed. In short, under -o, all of a project’s and/or application sequence’s derived code files are removed.

Incidentally, the fio files for specific RECORD_CLASSes, APPLICATIONs and PROJECTs can be recreated on demand by invoking Tracy using one of:

```
Tracy -gen_fio_for_recls <recls>
Tracy -gen_fio_for_app <app>
Tracy -gen_fio_for_proj_only
```

In the case of -gen_fio_for_app, all fio files for the RECORD_CLASSES in the APPLICATION will be created (unconditionally). In the case of -gen_fio_for_proj_only, this continues to be the case plus some project-specific and project-only C files are created.

### 3.7.7.2 DS Command: Detecting Desynchronization

Desynchronization can occur when apd’s, rcd’s, and pj’d’s are modified either by using DS Vi (without -dmtm) or when Archie is used to construct new aar’s and par’s from constituents taken out of a source code control system. In the latter case, it’s possible for the modification times to be anything, past or present. As a result, derived items could be thought by Daytona to be inconsistent with their primaries when in fact they are not – or equally bad, vice-versa. As mentioned above, all the apd’s, rcd’s, and pj’d’s can be made to have current modification times simply by executing DS
Resync -cyd_mod_tm -; this will cause Daytona to automatically rebuild the now all out-of-date derived artifacts as it encounters them during its normal processing of queries, etc.

The question to be addressed now is, how does one detect that derived artifacts have become out of date? In the case of indices, this is done by running Sizup -FYI -jt on all RECORD_CLASSes of interest. In the case of executables, executables run with the +R argument alone will report whether they were made under a Daytona release different (older) than the current one. Additionally, executable sensitivity to rcd changes can be tested by creating a shell environment where $DS_ZFLAGS contains -jt (and no +T in $DS_RFLAGS so that Sizup will be called to do -jt) and then invoking executables of interest with no arguments; those executables will not go any further than to report, if it is the case, that they are out-of-date with respect any rcd’s that they use (as well as reporting, if it is the case, that they were built by a previous Daytona release).

Furthermore, the user can determine what tables are used in any given query executable by running the SCCS what(1) command on that executable and grepping for RECORD_CLASS I/O FILE.

```
SUPPLIER RECORD_CLASS I/O FILE generated Tue Mar  3 00:00:34 EST 2009 \FOR APP orders FROM DS_REL = 22.0:02-26-09
```

### 3.7.8 DS Commands: Issues Accompanying The Building And Deployment of Applications

A number of issues arise when big projects use Daytona to create applications. One such issue occurs as an unwanted side-effect of using a source code control system to store the pjds, apds, and rcd’s. It is natural during a project build/make to pull the rcd’s out of source code control and then to use Archie to construct an aar in the same directory. Unfortunately, by default, Daytona programs like Tracy use the directory containing the aar as the directory to contain temporary rcd’s; likewise for programs like DS Edit that use Archie to -Checkout/-Return rcds, by default using the same directory containing their aar to hold the temporary rcd’s. Any such temporary rcd’s are eventually removed. This can cause some consternation on the user’s behalf when they discover that an rcd that they themselves had placed in the aar’s directory has now gone missing. Fortunately, by setting DS_ARTMPDIR to the directory of their choice, a user can ensure that rcd’s they place in the aar’s directory are not considered temporary, removable-hence-removed files by Daytona.

Another issue arises in the context of building a Daytona application on one platform for installation on another. During the process of building on the build platform, Daytona can well annotate the pj, if any, and apd’s with _System Generated LIBRARY descriptions that are specific to the build platform -- and which if copied verbatim to the install platform, would simply be incorrect. Typically, these are the LIBRARY descriptions Daytona creates to record Cymbal package dependencies. The problem obviously is how to obtain correct _System Generated LIBRARY descriptions in the pjd/apd’s on the install platform. The guaranteed way to do this is to use DS Resync -rm_sysgen - on the install platform to remove all of those LIBRARY descriptions (created by the build platform) and then to rebuild all the Cymbal packages on the install platform by using DS Compile, which by the way, will serve as a check that the install platform is indeed isomorphic/equivalent to the build platform. However, this is a sort of counter-intuitive approach since the application had just been built on the build platform, so why have to rebuild it again?

Copyright © 2013 AT&T
All Rights Reserved.
September 15, 2013
So, Daytona offers **DS Relocate**.

\[
\text{DS Relocate} \quad [\quad \langle \text{app\_seq} \rangle \quad | \quad - \quad ]
\]

What Relocate does is to visit the pjd, if DS PROJ has a non-empty value, and all apd’s specified, search out all _System Generated LIBRARY descriptions and all Source notes there and underneath them in dependent LIBRARY and FILE_BASE descriptions, and then find the associated files on the install platform and update the Source notes to reflect the new locations. The reason this maneuver is not needed for other FILE/FI

## 3.7.9 DS Commands: Running Queries

The command **DS Tracy** runs Tracy and the command **DS Stacy** runs the SQL-only translator, Stacy. Both of these commands execute shell command files which will prompt the user for needed information. For example, the user will be prompted to identify an application if no default one is acceptable and of course, the user will be prompted for the name of the file containing the query. There is also an opportunity to select various sorting and output format options.

Also, by means of the **-SOC**, **-ABC**, **-ZDC**, **-NPC**, and **-COU** command line options, Tracy and Stacy will include additional code in their output for runtime checking for string assignment overflows, array bounds violations, zero-divide attempts, and dangerous transaction logic, respectively. (More on **-COU** in Chapter 17.) This can be indispensable since C compilers do not provide this kind of facility. The absence of such checks can lead to programs that core dump in a non-explanatory way. Here is how to compile a query using these additional checks:

\[
\text{DS\_TFLAGS}="-SOC \ -ABC \ -ZDC \ -NPC" \ \text{DS Compile} \ \langle \text{query\_fl} \rangle
\]

When string overflow checks are requested, all application object files will be recreated with the C compiler called with a **-DDOING_SOC** macro-defining argument (and likewise for **-ABC** and **-NPC**). If the user has any C code like `usr.env.c` or that they have included in the make process by annotating **MAKE\_GOODS** in an apd or the pjd, then they are welcome to write `#if defined(DOING_SOC)` statements that include additional runtime error checking of their own.

When the user attempts to access an element of a non-associative array at a non-existent location/index, then trouble, often segmentation violations (SIGSEGV), can arise. However, by giving the **-ABC** flag (for array bound checks) to Tracy, the user requests that Daytona instrument generated C code so as to cause a useful error message to be output when Cymbal code attempts to access non-associative array elements outside the bounds of their array.

On some machines (for example, Suns), dividing by zero does not cause program termination; instead, special values like `Inf`, `inf.0`, `NaN`, and `nan.0` are returned (by the IEEE floating point code used by the operating system) and execution continues. Unfortunately, while clever, this is not mathematically correct: division of anything by 0 is undefined and is an error. Programs should stop when this happens and tell you exactly where the error occurred. **-ZDC** ensures that this happens when dividing or doing modulo by INT, UINT, FLT, MONEY 0.
Some Cymbal constructs such as VBL VBLs are implemented using C pointers. Unfortunately, this opens the door to user Cymbal code accidentally attempting to dereference a null pointer. This will unfailingly result in a segmentation violation (SIGSEGV) at runtime. However, by giving the -NPC flag (for null pointer checks) to Tracy, the user requests that Daytona instrument generated C code so as to cause a useful error message to be output when Cymbal code attempts to dereference a null pointer at the C level.

When -SOC, -ABC, or -NPC are used, the corresponding R.mk make file will cause all of the application objs (i.e., $(APP_OBJS) ) to be regenerated so that these flags will be used in them as well. Then, when the testing is over, to get the test code instrumentation out of these object files, simply rerun Tracy without using -SOC, -ABC, and -NPC so as to get an appropriate R.mk and then execute

```
  mk R rebuild_app_objs
```

The -TC (tracing-checks) option will cause Tracy/Stacy to generate code that prints to stderr information on most state changes that occur during the execution of a Cymbal program. This includes each assignment made to a user variable (-VTC), user-defined fpps being called (-FTC), channels (file or tendril) being open/closed (-CTC), BINS being accessed (-BTC), records being read in (-RTC), records being changed by transactions (-RTC), and processes exiting. This can be an invaluable aid in debugging declarative Cymbal where it is impossible to insert printing statements and where so much is being done for the user by so little Cymbal code.

Actually, -TC is shorthand for the combination of -VTC, -FTC, -CTC, -BTC, -RTC, standing for variable, fpp, channel, bin, and record trace checks (resp.). The printing of this tracing information can be controlled in an executable compiled with -TC (or any of its component flags) by (procedurally) setting Cymbal variable vbl_tracing, fpp_tracing, chan_tracing, bin_tracing, rec_tracing, to be _true_ or _false_ at various convenient points in the associated Cymbal program; by default, its value is _true_.

In specifying these checks to Tracy, a useful tactic is illustrated by "-TC -!BTC" which enables all of the -TC checks but then disables -BTC.

There are two important caveats in using -VTC. The first is that when array element values change, the element is looked up again in order to report its new value. This is probably undesired if a function which changes the state of the program is being called in order to compute the array indices needed to look up that array element. Secondly, any integer array indices that are printed are relative to the C implementation: they may not correspond to what the user is thinking of at the Cymbal level.

The -IVC (implicit-variable-checks) option causes Tracy to emit a warning each time a user variable is encountered which does not have an explicitly given scope. The warning identifies the line number of the first occurrence of the variable in the query.

The -VHC (variable-homonym-checks) option causes Tracy to print out the disambiguated names of all homonym variables. For more information on homonym variables, please see the Variable Scoping section in Chapter 5.

The -DUV (describe-user-variables) option causes Tracy to print out such descriptive information as the type and scope on each variable defined or imported into each task.

The simplest way though to process a query is to enter `DS QQ <request_file>`. `DS QQ` begins by invoking `DS Edit` on the designated request file. If the file name ends with .Q (for Cymbal or DSQL) or
.S (for pure standard SQL), then the path .:/Q:.:/q:.:/q is searched for the file; else the current directory alone is searched. If no file by the given name is found, one will be created. After possibly modifying the request file, the user exits DS Edit and QQ will continue with running the request for the user all the way down to formatting the answers. If the <request_file> is -, which indicates stdin, then there will be no interaction with DS Edit.

The -t, -p, -d, -x options inform QQ that the desired output format is tabular, packet, Cymbal description, or XML respectively; the default (i.e., -u) is unfiltered. The filtering options assume that the data is being presented in a form compatible with what the corresponding filters like DC-prn require.

If DS QQ is invoked with the -v verbose option, then not only will messages indicate each step of the query evaluation but also the output will be saved in a temporary file that QQ will identify. If the (query) input to DS QQ is from stdin, then the output will be to stdout; otherwise, it will be sent to $PAGER if it exists, else to stdout.

Here is how to just run a query and get the answers formatted with no interactive fuss:

```
DS QQ -t - <Q/stpaul.1.S
```

If DS QQ creates temporary files, it creates them in /tmp by default, else in $DS_TMPDIR if DS_TMPDIR is set. DS QQ cleans up after itself by removing temporary file artifacts, with the exception of the executable that it makes and leaves behind, which is named R.QQ.

Sometimes, all the user wants to do is to evaluate a Cymbal constant expression quickly, where a constant Cymbal expression is a constant term, meaning a constant or a function call with constant term arguments. DS Expr reads a term from stdin and prints out its evaluation to stdout. Here is a sample use:

```
% echo "4*5" | DS Expr
20
```

Here is another example using the -verbose option:

```
% echo "today()+10" | DS Expr -v

today()+10 = 2005-09-07
```

(The stdin input to DS Expr may alternatively be terminated with a `D, as is the UNIX convention.)

Caveat: DS Expr is implemented by means of DS QQ and the _Show_Exp_To macro: thus, unfortunately, it is subject to the limitations of ds_m4. In particular, any trouble it may have with double-quotes can be avoided by using LITERAL constants instead of STRING constants.

If DS Expr creates temporary files, it creates them in /tmp by default, else in $DS_TMPDIR if DS_TMPDIR is set.

Observe though that DS QQ is designed for queries that don’t require user interaction at runtime since it provides no opportunity for the user to interact with the executable at runtime by, for example,
providing runtime arguments. The easiest, no-fuss way to create an executable directly from a Cymbal or SQL query is to use DS Compile. Once made, the executable can be invoked from the command line as usual and then the required runtime user interaction can take place. The first command line argument to DS Compile is the request file and the second argument is the name of the corresponding executable. If the first argument is missing, then a usage message will follow plus an opportunity to be prompted. If the first argument is −, stdin is assumed. If the second argument is missing, then R is assumed; if it is −, then the name of the executable is formed in the following way:

1. If the query file name ends with .[qQsS], then the executable name is formed from the query file name by replacing that suffix with .X .
2. If the query file name ends with .cy, then the executable name is formed from the query file name by removing that suffix.
3. Otherwise, the executable name is formed from the query file name by appending .X .
4. Independently of the above, if the query file path begins with q/, then the executable file path will begin with bin/ .

DS Compile uses a lock-file-based locking mechanism to ensure that there are never two DS Compiles working at the same time in the same directory.

For any of DS Tracy, DS Stacy, DS Compile, and DS QQ, if the request file argument is −, then stdin is assumed to contain the query.

Some applications may compile many queries down into executables that are then stored in some library for subsequent invocation, possibly by some GUI (graphical user interface). The question then arises, for any given executable, how does the user run it, i.e., what is its usage? Daytona provides a very handy mechanism for adding a +? and +U usage option to the executables that it generates. All that is necessary is to define a STR Cymbal variable usage in the Begin task (or default main program) whose default value matches "[ \t\n]"Usage:"[ \t\n]RE . Here is such a definition:

locals: STR .usage = "Usage: R <city_pattern>"

Incidentally, Archie, Edit, and QQ are written in Cymbal. Their Cymbal source can be found in $DS_DIR/EXAMPLES/sys .

Although rarely used, the user does have the option of modifying the code that Tracy and Stacy write. In any event, this code can always be compiled and executed by issuing the command DS Exec. The default names for these program code files all begin with “R”. The DS Exec script will enable the executable name to be changed. The command DS Mk R will make the executable (without executing it). Typing in the executable name will execute it. (But make sure that you have no ksh alias for R.)

3.7.10 DS Commands: Sizup For Validation And Index Building

Sizup is the Daytona program which creates and maintains indices of various types for DC data files. Daytona will cause Sizup to automatically rebuild the indices any time data is about to be accessed which is in a file newer than its index files. Sizup can take a fair amount of time to run and so the user may wish to get the job done in advance by explicitly invoking Sizup with the command
**DS Sizup**; as with all of the *DS* commands, *DS Sizup* operates in both prompting and direct command mode, with the prompting mode, not atypically, offering more options. Of course, the one-time creation of the index files more than pays for itself by supporting faster execution of queries.

*Please note that when SQL and Cymbal queries update the data, any associated indices are updated as well at the same time. Given that the indices were valid for the data before a transaction is run against it, they will be valid afterward as well. Consequently, there is no need to run Sizup on data that has been modified by Daytona transactions.*

For reference in the rather lengthy discussion concerning Sizup that follows, here is the Sizup usage information produced by invoking Sizup with the `-?` option:

**Usage:** Sizup -m{andatory} -just_testing
- app <app> -fls <files> -recls <record_classes> -fls_via_stdin
- n{ullsok} -dmrcd -lock_patience <patience> -ro_data
- no_validate -validate -validate_only
- save_faults -delete_faults
- source <src> -indices_source <dir>
- rec_map_spec_file <file>
- clean_slate -create -truncate
- adds_fl <file>
- add_new_keys
- packing -padding
- max_doable_fl <file_count>
- nonempty_fl_only
- parallel_for <clone_count>
// the following are typically not used by end-users
- check_rec_syntax -no_check_rec_syntax -no_check_ascii
- sort_key_sort -bt_key_sort
- do_adds [ _in_place_ | _recursive_ ] -adds_indices_source <dir>
- qc <executable> -qc_abort_dont_rebuild -timeit
- arch <fl> -aar <fl>
- do_section <sect_nbr>/<tot_sects>

In more detail, whenever Sizup runs, it checks each data file to see if there is something to do. Its test for something to do is to see if the data file is newer than the corresponding system ".siz" file which must accompany every data file or if the rcd associated with that data file is newer than the ".siz" file. If the ".siz" file is newer than its data file and the rcd is older than the ".siz" file, then Sizup does nothing with that data file.

Because Daytona DC data file query programs use an index file for each access they make to the data files, on some machines, at most 8 accesses can be created for any one query: older versions of UNIX tend to have a 20 file limit on the number of files that can be open at the same time and 8*2 + stderr + stdout + stdin = 19 open files.

Sizup operates in two modes: create and update. Under create-mode, which is the default, Sizup’s goal is to ensure (or possibly just to test) that all indices exist and are up-to-date. Under update-mode,
Sizup makes additions to existing index structure: it either adds new records on the end of an existing file (while incrementally updating the existing indices) or it creates newly specified indices to complement existing ones or, in the case of horizontal partitioning (see Chapter 23), it creates the indices for new files for record classes that already have other Sizup-processed files associated with them.

Sizup can be run under 3 levels of urgency. Under the default level, if indices are out of date, then either they get remade if running under create-mode or warnings will appear if running under update-mode.

Under the mandatory level of urgency, which is indicated by a -m or -mandatory command line option, regardless of the mode, Sizup guarantees that, no matter what it takes, the indices will be up-to-date by the time it finishes: this is accomplished by unconditionally rebuilding the indices.

Under the “just testing” level of urgency, which is indicated by a -jt or -just_testing command line option, Sizup will only go so far as to notify the user with a message saying whether or not the indices are up-to-date and why they are not if they are not; if they are, it returns a 0 exit status, if not, then 1.

The -fsls keyword followed by one or more file paths is one way to inform Sizup which files to work on. (The file path may be either the base file name or else, the fully rooted path to (and including) the base file name.) If a file path argument to -fsls is prefixed with @<record_class>= (as in @SUPPLIER=SUPP), then Sizup will only search for that file path in the rcd for the indicated record class; otherwise, it has to search the rcds one-by-one in a sequence determined by $DS_APPS in an effort to find the first record class description that refers to the given file path; this inefficient search should be avoided for the additional reason that a file path may appear in more than one rcd. Not only is the @-syntax faster for Sizup, it also reduces lock contention on the rcds. Alternatively, the -recls option for Sizup can be used instead of or along with the -fsls option to specify that all the files associated with a given record class are to be processed by Sizup. This option is handy in the context of the horizontal partitioning feature described in Chapter 23. -recls arguments are handled completely independently of any -fsls arguments; in particular, they do not specify any type or rcd information for any -fsls arguments. Indeed, all a -recls specification is is an abbreviation for a -fsls of file paths (using the @<record_class> prefix) that belong to the rcd corresponding to that record class. At heart, Sizup thinks just in terms of files, meaning that anything that refers to record classes in Sizup arguments is translated into its equivalent in terms of files.

Sizup can only work on the record classes associated with just one application at a time. This application can be either be deduced from the $DS_APPS environmental variable or it can specified explicitly by using the -app keyword.

In order to keep Sizup from having to consult the environment in order to figure out where things are, the aar can be specified explicitly with the -aar option, in which -app should not be specified. The greatest savings will result when the aar file is specified using an expanded path from the root (expanded in the sense of no $s or ~s).

3.7.10.1 Sizup: Computing And Storing Index Statistics

One of Sizup’s tasks is to compute several file and index statistics for query optimization use by Tracy. In the horizontal partitioning case, whether or not FILE_INFO_FILE (see later) is being used, the
statistics for the record class are an integration of the statistics for each BIN, to the extent possible. As a consequence of this and other considerations, Sizup will modify the rcd. In some applications, most of the time, the user does not want Sizup to modify the rcd. For example, if an application works by creating new data files completely from scratch by using SQL inserts, then the data files will be initially empty and it is pointless to have Sizup compute index statistics on empty data files. In this case, it is better to just continue to use some typical index statistics which had been computed earlier when ‘sizing up’ a data file that had some data in it or to use the previously mentioned keyed for index keyword in Cymbal queries to hard-code the choice of keys. So, if the -dmrcd (for ‘don’t modify the rcd’) option is used, then Sizup will not make any modifications to the rcd and since it does not need to modify the rcd, it will also refrain from getting exclusive locks (both file lock and lock file) on the rcd. This can be a valuable benefit of using -dmrcd if Sizup runs take so long that they overlap substantially with query writers wanting to compile their queries: in order for a query to be translated and compiled, relevant rcds must be accessed for reading and that can only happen if there are no exclusive locks on those rcds. An additional sometimes useful side-effect of -dmrcd is that any index statistics currently existing in the rcd will not be subject to change.

The -dmrcd option is also important for those situations where the Daytona user as a database administrator (DBA) has created an application for other people to use. It is convenient in these cases to have just one aar for everyone to use when they run Sizup and Tracy. In this setting, these people are really just ‘read-only’ users of Daytona: since the application has been built for them, they will just be running queries and building indices but they will not be doing DBA activities like adding new RECORD_CLASS descriptions and new KEY descriptions to the aar. For reasons of convenience or in order to enforce this ‘read-only’ status, the permissions on the aar may make it writable only by the DBA, not by the users of the application. In this case, Sizup invocations will have to use -dmrcd in order to prevent attempts to write in the aar.

A very useful way to automatically ensure that Sizup is run with the -dmrcd option even if it is not explicitly given on the command line nor is it included in $DS_ZFLAGS is for the DBA to change the permissions of the aar to be read-only by everyone, including the DBA (i.e., ’chmod 444’). Daytona tests for this case and if it holds, then no Daytona program will attempt to modify the aar.

The user can specify in the rcd that the system is not to maintain any index statistics. One advantage of this in the case of static horizontal partitioning is that if the statistics are truly not needed and if the number of bins is large, then the size of the rcd will be substantially reduced by not carrying any index statistics. Sizup will also run faster when statistics are not kept. To achieve this, simply place a <Maintain_Index_Stats, no> note under the KEYS description in the rcd. If this note is added when previously index statistics had been kept, then they will be removed from the rcd.

Whenever a query executable runs, if it is run without the +T (or +trustme) option, then it will call Sizup both to verify that all of the indices are up-to-date and that the executable is newer than the rcd. In order to prevent the recomputation of indices from automatically invalidating previously compiled executables, by default, even when Sizup modifies an rcd, it does not change its modification time. However, it will change the modification time if -mandatory is used and -dmrcd is not.

3.7.10.2 Sizup: Data Validation

By default, Sizup validates the data being processed by checking to see that it conforms to its
record class description. For starters, the `-check_rec_syntax` flag (which is on by default) does a quick
sanity check on the format of each record by seeking to the last field of the record, which happens by
skipping over separators and TUPLE delimiter characters. If a record has been truncated or otherwise
seriously abused, then this check will pick it up. This check can be turned off by using the
`-no_check_rec_syntax` flag.

The `-no_check_ascii` turns off the checking for bytes (in non-record-compressed data) with the 8th
bit set that is the default when validating. Using this flag is not recommended.

Secondly, Sizup checks any Validation_RESs, Min_Values, and Max_Values specified for FIELDS in
the rcd. The validation option increases the runtime of Sizup, of course, but this is well worth it. In
fact, totally puzzling havoc can result if bad data, which has not been caught by a Sizup validation run, is used
in a query. Validation can be disabled by using the `-novalidate` option with the advantage being
substantially faster Sizup executions; however, the user assumes all risk if this option is selected. In
order to validate pipe_/pipein_ FILES, the `-validate_only` must be explicitly specified. In general, for all
record classes, `-validate_only` implies Indices Banned, no Siz_Access, -validate, and -mandatory. Since
the indices will not be rebuilt using `-validate_only`, Sizup will not be able to detect duplicate key errors
in this mode.

The Sizup validator considers missing values in the data file records to be errors unless explicitly
told otherwise in the rcd or by its command line. To totally turn off this feature, the `-nok` or `-nullsok`
flag can be used on the command line. To allow missing values for specific fields, the rcd for the
chosen fields should be modified so that the Multiplicity designation `(0->1)` appears in the Type
note value under the corresponding FIELD description. Note that if the field is LIST/SET-valued, then,
to allow a (non-missing) but empty LIST/SET value, `(0->)` should appear inside the LIST/SET
Type specifying the Multiplicity of the number of elements in the TUPLE.

In the event that Sizup discovers errors in the records it is building indices for, the user may wish
to use the command DS_Msgmrg to annotate the error report with the actual data file records involved.

However, Sizup now has much more sophisticated features for dealing with errors. First, by
specifying `-save_faults`, the user will cause Sizup to create a file containing information pinpointing the
exact location of each faulty record as well as displaying the record itself. The fault file is placed in the
same directory as the data and has a name consisting of the data file name suffixed by `.faults`, as in
ORDER.faults. Here is some sample output:

```
#/home/john/d/ORDER
# subsequent comment format: line_nbr|rec_nbr|rec_offset|rec_len

#1|1|0|7
0::)

#14|10|405|28
9:490:172:12/16/86:05:863:0
```

Each fault generates an entry that consists of a comment record (consisting of the line number, the
record number, the record offset, and the record length) followed by the faulty record itself. Please
note that in the case of duplicate unique keys only, information on the bad record is put into
Sizup.msgs but not into the .faults file.

There are occasions where the best way to handle faults is to delete the faulty records. For example, some data file contain 30 million records with two that are faulty. It could well be advantageous just to delete the two faulty records and get the bulk of the data loaded and perhaps return later and add the repaired faulty records to the rest by means Sizup batch adds. One way to delete faulty records is to call the Delete_Faults program with usage:

```
usage: Delete_Faults [ <.fault_fl = - > ] [ <com_char = % > ]
```

Delete_Faults processes a fault file by writing a Daytona delete byte at the start of each faulty record. A subsequent Sizup will (of course) ignore the now-deleted faulty records and validate and create indices for the remaining good ones. The fault file can be edited to remove reference to particular faulty records, with the understanding that those records will be repaired by other means, like perhaps a text editor. The space taken up by now-deleted faulty records may be removed by running a subsequent Sizup with the -packing option. When KEY uniqueness constraints are violated, all but one of the records with the replicate key values will be deleted.

When Sizup is run with the -delete_faults option, it will automatically delete faulty records as it finds them. That particular Sizup run will end with error messages and with unusable indices; however, just rerunning that particular Sizup invocation again exactly will be successful because the previously faulty records will have been deleted by the first run.

Nota bene: there is one exception to this rule of "try the same thing twice to fix faults using -delete_faults". This occurs when doing _in_place_ Sizup batch adds _and_ when there are duplicate key errors that involve records in the adds file. In this case, the first batch adds run will fail and only the bad records associated with non-duplicate_key errors will be deleted. Then when the exact same batch add invocation is executed again, all the records in either the base or adds file that have duplicate key values will be deleted. This is good but unfortunately, this process corrupts the indices for the base table. Not to worry though, as the error message indicates, just add a -mandatory to the same Sizup batch adds invocation and the resulting Sizup batch adds will at last accomplish the desired batch adds, all faulty records having been deleted. Once again, this necessity to run Sizup four times to overcome faulty records only occurs when there are duplicate records in a _in_place_ Sizup batch adds situation and actually, can be handled just by reading the error messages and doing what they say.

### 3.7.10.3 Sizup: Index Validation

It is possible for the data to get out of sync relative to its indices without the system being aware of it -- this would happen if new data was moved into place preserving the timestamp of the old data files. It is also possible for there to be a Daytona bug that causes incorrect indices to be built. To prove or validate that the existing indices are indeed correct, the user can run DS Check_Indices. This shell function takes a list of record classes as command line arguments and checks the indices for each of them. It does this by writing a shell script named `ck.RECLS.sh` where `RECLS` is the specified RECORD_CLASS. It is this shell script that does the actual checking of the indices. By invoking DS Check_Indices with the -n option, the shell script alone will be generated -- but not executed. This way, portions of the script can be isolated and run individually. To check if the Daytona keys in a B-tree are unique, DS Check_Indices will run Check_Btree_Unique as appropriate.
run separately and has this usage:

```
Check_Btree.Unique [ [ '<comch=%>' '<unitsep=|>' ] <bt_f1.T> ]
```

### 3.7.10.4 Sizup: Miscellaneous Features

When a query (automatically) calls Sizup, the default behavior is for it to rebuild any out-of-date indices it finds before proceeding to process the query. In some situations, so much time and effort has been invested in making the indices that the user may wish to ponder the situation at leisure before possibly rebuilding the indices. This can be accomplished by putting the `-qc_abort_dont_rebuild` keyword in the value for the shell variable DS_ZFLAGS.

When the `-clean_slate` option is used, the data files specified to Sizup will be truncated to 0 bytes (or created with 0 bytes, if they didn’t already exist) before creating the indices. This is done in a concurrent manner. Consequently, using the `-clean_slate` option is a good way to “reset” or “initialize” a file to get ready to receive new data.

`-clean_slate` is equivalent to specifying both `-create` and `-truncate`. For the record classes and/or data files specified as arguments to Sizup, `-create` will cause Sizup to create 0-length data files when no data file is present; `-truncate` will cause Sizup to truncate to 0 length any data files that exist. In either event, empty indices are created for the 0-length data files. As an additional feature, `-create` will create any intervening directories in the path to the data file or index files that are not already there, thus allowing for Sizup to be used to initialize a directory tree. Also, for `_fifo_` FILES, `-create` will create the file as a FIFO.

Suppose a bunch of indices have already been created by Sizup and then it is discovered that yet another index is needed in addition to the previously created one. If Sizup is run with the `-ank` or the `-add_new_keys` update option, then only the currently non-existing indices will be created. Of course, the new keys must be described in the rcd. This can save a lot of time when working with large data files.

Suppose some indices have already been made for a given data file "file1" and then some additional new data file records become available in file "newrecs". In order to concatenate these records on the end of the existing data file and to update the existing indices (without remaking them from scratch), use the `batch adds` feature by simply executing the command (for application app3):

```
DS Sizup -app app3 -fls file1 -adds_f1 newrecs
```

Note however that the btrees created/extended by Sizup batch adds can be quite bloated (as in twice the size) of indices that are created by a regular Sizup run on a data file all by itself. So, to save storage, it makes sense to do a regular Sizup on the first/initial base file and then Sizup batch adds on all increments to that file.

(Daytona’s default algorithm for doing batch adds is called in-place; there is another an older vestigial algorithm called recursive, which is not even supported in a LARGEFILES environment -- so, don’t try to use it.)

There are occasions when the user wishes to forbid Sizup to make any indices at all, including the ".siz" file. Such a goal is accomplished by placing an `<Indices_Banned, yes>` note on the KEYS description in the appropriate rcd. In this event, Sizup will not even be called by query executables to
check to see if the data should be validated or not. This is contrasted with the case where
<Indices_Banned, no> and <Siz_Access, no>: here, a small "siz" file is created with some status
information but no indexing capability; Sizup will use the status information and the modification time
of the .siz file to determine whether or not to validate the data.

If the -packing option is used, then Sizup will remove any unused space from the data file that had
been left there by previous Daytona record deletes. As usual, if the indices are up-to-date, then -packing
will not have any effect unless -mandatory is specified.

Since Daytona updates will modify data file records in place if there is enough room in the file
between the start of the record and the next new-line character, it can be beneficial to pad records with
comment characters so as to leave room for accommodating future modifications without relocating the
record elsewhere in the file (which is a more expensive operation). As usual, if the indices are up-to-
date, then -padding will not have any effect unless -mandatory is specified. By using a Pad_To_Len rcd
annotation, the system will automatically pad records with comment characters to the specified
minimum length. For example, a <Pad_To_Len, 56> note for the FILE description in an rcd specifies
that the system is to maintain a minimum record length (with comments and the new-line) of 56
characters when the occasion arises. And the occasion arises whenever record-at-a-time updates are
done and when Sizup is invoked with the -padding option. If a Pad_To_Len rcd annotation is not
present for a FILE description, then a Default Pad_To_Len annotation will be used instead as soon as it
is found by looking at any ancestor FILE_INFO_FILE or BINS description.

Ordinarily, Sizup creates indices in the same directory as the data file resides. By annotating a
FILE rcd description with a Indices_Source note, the user can require that the B-tree and .siz indices be
created in the indicated directory. Such placement provides some flexibility with regard to where files
are stored and, since the indices can be placed on a different disk from the data, may even (no
guarantees) speed up query processing by enabling data-record and index-entry retrieval to each be
handled by a separate disk controller.

The argument to the optional keyword -source is taken to be the value of the Source note for any
FILE nodes encountered during the Sizup run. The argument to the optional keyword -indices_source is
taken to be the value of the Indices_Source note for any FILE nodes encountered during the Sizup run.
The argument to the optional keyword -adds_indices_source is taken to be the value of the Indices_Source note for the processing of the adds file in a recursive batch adds Sizup run; has no
meaning or use for in-place batch adds, which is the default. The argument to the optional keyword
-rec_map_spec_file is taken to be the file path to the file specifying how to map the data file records on
input. Most commonly, this specifies the file containing the compiled compression dictionary.

When, at the start of a run, Sizup is inventorying the status of the files it has been assigned to
work with, unless otherwise instructed, it obtains an exclusive lock on each file in turn. In part, this is
to determine whether or not the file is currently being worked on by a transaction or by another Sizup.
In order to keep this inventory process from taking an indefinite period of time due to blocking
indefinitely on pre-existing locks, Sizup by default waits for 15 seconds and if it has not been able to
obtain the lock by then, it exits with an error message. The -lock_patience option allows the user to
specify a different time interval, if desired. This can be either be _wait_on_block, _fail_on_block, or a
non-negative number of seconds. An argument of 0 (i.e., no patience) is equivalent to _fail_on_block_.
To cause Sizup to forbear getting any kinds of locks for any file, just use the -ro_data option which will
also prevent Sizup from modifying any data file in any way. See the discussion of <Read Only, yes> in Chapter 23.

Even when Sizup is building indices for a file in regular or batch adds mode, queries that do not get file locks (so-called dirty readers) will nonetheless (almost always) run successfully in the sense of not ending with an error message and getting information from all available records that had been indexed at the time. Of course, such readers will not benefit from transactional consistency. For maximum benefit, such dirty readers use both with dirty reads ok and either ignoring failed opens or with ignoring failed opens handler in their there_isas (see Chapter 13).

3.7.10.5 Sizup: Working With A Large Number Of Files

Frequently horizontally partitioned record classes are implemented using hundreds, if not thousands, of files. Suppose a user wishes to run Sizup on only a (sizable) fraction of those files. It can be a bit cumbersome to try to feed all of the associated file names to Sizup by means of the shell on one very long or backslash broken-up line. Certainly, manually invoking Sizup hundreds of times will prove to be very time-consuming. The preferred alternative is to invoke Sizup with the -fls_via_stdin option (instead of the -fls option) which will enable Sizup to obtain the file names it needs from stdin. Here is a sample invocation:

Sizup -fls_via_stdin -app orders -FYI -m <some_data_files

some data files is a file consisting of a sequence of lines containing one data file name per line; using the @-syntax of @<record_class>=<file_path> is preferred due to its inherent efficiencies.

Now, there may be so many files (in a single record class or not) to do that they simply cannot be done in one run of Sizup because that would result in Sizup simultaneously opening more files than the operating system will allow. One simple way around this is to use the -max_doable_fls keyword as in

Sizup -rec WIDGET -max_doable_fls 64.

This invocation will do up to 64 files in the WIDGET record class. If there are 640 files in the WIDGET record class that need indexing, they can all be done by invoking the above command 10 times -- and that means exactly that command executed 10 times.

However, suppose that there is a subset of the files to be possibly re-indexed and that they can be characterized by conditions on their horizontal partitioning attributes. Then this can be achieved by a simple Cymbal program (making use of language constructs to be discussed in detail later) (as hparti.sizup.1.Q illustrates):
fet [ .region, .cat, .uni, .fl, .src ]

ist(

there_is_a_bin_for ORDERB
with_bin_vbl fl
with_source_vbl src
where( Region = .region which_is < 13
and Category = .cat which_is = "A" | = "G"
and Uniquifier = .uni )

)

set ? = shell_exec(

"Sizup -fl @ORDERB=.src/.fl -app misc -FYI "ISTR ")
otherwise do Exit(22);
)

When *-nonempty_files_only* is used, Sizup will only consider nonempty files for further processing. A nonempty file exists but has size 0.

The *-parallel_for* keyword can be used to cause Sizup to clone itself a specified number of times and thereby process its data files in parallel, each clone working on roughly the same number of files. For example,

```
Sizup -recls BIG_TABLE HUGE_TABLE -app myapp -parallel_for 4
```

will cause Sizup to identify all of the files making up BIG_TABLE and HUGE_TABLE that need index building, divide them up into 4 roughly equal size groups and process all of the groups in parallel. There will be substantial speedup when this is done on a symmetric multiprocessor.

Of somewhat limited value, for any Sizup run specifying any number of files and record classes, the user can use the *-do_section* keyword to specify a certain fraction of the job to be done. For example,

```
Sizup -recls BIG_TABLE HUGE_TABLE -app myapp -do_section 3/15
```

will cause Sizup to identify all of the files making up BIG_TABLE and HUGE_TABLE that need index building, divide them up into 15 roughly equal size groups and process the third one of those groups. Of course, those files having been processed, running the same command again will choose a different set.

### 3.7.10.6 Sizup: Managing Long Runs

For larger files, say, with hundreds of thousands or millions of records, and depending on the number of indices, Sizup can take a fair amount of time to run. This should be no cause for concern since queries frequently run in seconds on data that Sizup has spent hours on. Now when Sizup’s running time exceeds whatever default CPU time limit has been given to the user, the operating system will terminate the Sizup process, sometimes without even making much effort to properly notify the user. In this case, the user must somehow tell the operating system to leave Sizup alone. On BSD systems using the Korn shell, this is done by using the ksh command "ulimit". For example, a "ulimit -t
100000" command in the user’s .profile tells the operating system to not kill any process for time until 100,000 seconds have passed.

During long Sizup runs, the user may wish to know how far along Sizup is at any given moment. This can be easily estimated by dividing the number of bytes in the associated .siz file by 4 since the .siz file contains 4 bytes for every record in the data file. Predictions as to when Sizup will end should be based on the fact that index building has the $n \log n$ time complexity of sorting where $n$ is the number of records in the file.

### 3.7.10.7 Getting The Most Speed From Sizup

Since horizontal partitioning in practice can easily result in hundreds of files per record class, every second saved of Sizup processing per file is important. The remainder of this section has is concerned with all of the tricks one can use to get the utmost of speed from Sizup. Casual readers may wish to skip over this material; only those greatly concerned with getting the utmost speed from Sizup will be interested.

The first trick is to use `-novalidate` if the user can guarantee that the data will match its rcd description. Also, using `-dmrcd` will eliminate calculating and writing information back to the rcd, which is particularly useful when working with the large rcds associated with some horizontally partitioned record classes (`ar(1)` is not too swift in these cases).

Furthermore, when using the `-fls_via_stdin` option, the following special syntax for each data file:

```
@record_class=complete_data_file_path
```

will enable Sizup to dispense with reading in the rcd in order to determine where the files it is to work with are actually located in the file system. Here is an example:

```
@SUPPLIER=/${HOME}/${DS_DDIR:-d}/SUPP
```

(The command will still work if an incomplete data file path is provided, with the understanding that the rcd will have to be read in.)

When Sizup builds indices, it needs to sort the keys. By default, `sort(1)`, as implemented using path `$DS_SORT`, is used. On installation of Daytona, `$DS_SORTOPTS` can be set to whatever `sort(1)` command line options are desired, such as how much of main memory to use (via the `-y` option for memory in kilobytes) and which directory should contain `sort`'s temporary files (e.g., `-T /home/mytmp`). To override the installation’s definition of `$DS_SORTOPTS` in `DS.Set`, just prefix your invocation of Sizup with the new definition of `$DS_SORTOPTS`, as in:

```
DS_SORTOPTS='--T /tmp/mydir -y 175000' Sizup -m -rec SUPPLIER
```

On large memory machines, the use of the `-y` option can speed Sizup up considerably; for sizing purposes, remember that by default, there will be one sort per index per file being indexed at the same time.

When Sizup uses `sort(1)`, it spawns several child processes per index. This is very good when working on a multiprocessor. However it can also result in the creation of more child processes than a uniprocessor system can handle. In that event, to avoid using `sort(1)`, just invoke Sizup with the `-bt_key_sort` option; `-sort_key_sort` is the default. When `-bt_key_sort` is used, the indices
will not be as compact as they are otherwise. Unfortunately, Sizup running under \texttt{-bt\_key\_sort} does not currently compute index statistics for cluster B-trees. Also, for all B-tree indices, the statistics computed under \texttt{-bt\_key\_sort} are not as complete as those computed under \texttt{-sort\_key\_sort}, the default.

### 3.8 Miscellaneous DS Commands

Daytona metadata like the contents of aar’s and par’s describes the components of a Daytona database. Unfortunately, in various ways, the database can become inconsistent with its description. Fortunately, the \texttt{Checkup} command will exhaustively test the hypothesis that the database (identified by $\texttt{DS\_PROJ}$, $\texttt{DS\_APPS}$, and $\texttt{DS\_PATH}$) is consistent with its description (in terms of the aar’s and par’s). Discrepancies will be reported. Use \texttt{DS Man} to see the Checkup man page for details.

A \texttt{DS\_Clean\_Misc} command will remove from the current directory all of the many temporary files that Daytona can generate.

The \texttt{Census} command computes a variety of statistics for all the data files it can find on the basis of its arguments which specify .\texttt{siz} files.

```bash
# Census +?
Usage: Census [ -c[omm]as ]? [ -i[n]dices ] [ <siz_file_paths_RE> ]*
  if no siz_file_paths arg, then ./*.siz is used
N.B. may wish to quote shell expressions here!
```

If the optional \texttt{-commas} is given, then digits are grouped into threes by commas. Here is some sample output:
Census works as quickly as possible to compute this information from information cached in the designated .siz files. It will also print out the paths to .siz files that do not have GOOD persistent Status codes, thus helping to identify instances of failed Sizups. Census needs to create a temporary file to run. By default that is in the current directory; to change its location, use $DS_TMPDIR as in:

```
$DS_TMPDIR=/tmp Census '*.siz'
```

The `-i` option prints out additional statistics on the indices.

### 3.9 System And User Imports And Definitions

In addition to application and project descriptions, Daytona also needs to read Cymbal imports and definitions for system and user-defined functions, predicates, and procedures (fpps). Imports and definitions for Daytona’s built-in functions are contained in the file `sys.env.cy` in the `$DS_DIR` directory and are attached as an appendix to this manual. The concepts behind these imports are discussed further in the latter portion of Chapter 6 on procedural Cymbal.

Please take a look at this file `sys.env.cy`: it contains the usage characteristics for all of Cymbal’s built-in functions, predicates and procedures (fpps), even ones not discussed in the manual because they are so new. You may want to use these. In fact, these prototypes often tell you all you want to know. Likewise, for those people who know how to read grammars, the Cymbal grammar document is an indispensable reference for defining what it is that makes a Cymbal query syntactically correct.

User definitions for Cymbal fpps as well as the Cymbal imports for their fpps written in C are
kept in a number optional file usr.env.cy in some directory in the path $DS_PATH as well as proj.env.cy for the project proj mentioned in $DS_PROJ and app.env.cy for each application app in $DS_APPS. Further information on these files is also given in Chapter 6.

It is also easy to import functions written in C into Cymbal; see Chapter 6 for details.

3.10 Macro Facility

Daytona uses ds_m4, an extension of m4, to provide a macro facility. All Cymbal and SQL queries are automatically preprocessed by ds_m4 by using $DS_DIR/rename.m and $DS_DIR/sys.macros.m as well as macros in some other files identified later in this section. Note the distinction between using ds_m4 itself, which is upwards compatible with m4, and Daytona using ds_m4 customized with Daytona-specific macros.

Here is the usage for ds_m4:


(The \ above is just escaping the new-line.) Note the absence of a space between the option and its argument. These options are the standard ones from the original m4 -- see man m4(1). One of the more useful ones is -B whereby one specifies the size in bytes of the ds_m4 internal pushback and argument text buffer: error messages announcing exhaustion of this resource can be forestalled by providing a larger amount for this argument than system shell variable $DS_M4 (used to invoke ds_m4) currently provides. Furthermore, Daytona’s user-level shell variable $DS_M4FLAGS can be set and exported to add additional arguments to the $DS_M4 invocation. Also, note the some of the options, like the ones for -D define, -U undef, and -I include, can be interspersed among the input files.

When it comes to finding include, not input, files, first, if the file argument to include contains a dollar sign or begins with a tilde, then it is expanded by echoing it into the (Korn) shell. Then if the possibly expanded file begins with a slash, then that entire path is opened. Otherwise, the current directory is searched for the include file. If not found, then the directories that are searched first are the ones, if any, specified with the -I option, in the order found on the command line. (A -I argument consists of one or more directory paths separated by colons.) Finally, as needed, if the M4PATH environment variable is set, it is expected to contain a colon-separated list of directories, which will be searched in order. -I arguments and the M4PATH value may contain unexpanded shell expressions using dollar; a tilde can only be used if it starts the concatenation of the -I arguments and the M4PATH value. Basically, the only expansions that occur are the ones that would occur if the concatenation of the -I arguments and the M4PATH value were echoed by ksh. Obviously, the tilde is a little finicky here -- safer to use the dollar sign.

ds_m4 also has two additional macros that m4 does not: _m4_file_ expands into a double-quoted string containing the current file name (which string is "-" if stdin) and _m4_line_ expands into a double-quoted string containing the current line number in the file.

When used by Daytona (i.e., with $DS_DIR/rename.m), ds_m4’s built-in macros have had their names changed so that they begin and end with an underscore. When Daytona uses ds_m4 (i.e., with $DS_DIR/sys.macros.m), its quotes are the two digraphs @< and>@ (called “squawks”) and its
comment characters are double quotes. It is important to remember that \texttt{ds\_m4} macro substitution does not occur within \texttt{ds\_m4} comments: consequently, the user should realize that no STRING constant (which are delimited by double-quotes) will be subjected to any \texttt{ds\_m4} macro expansions. There are good reasons for that but if the user would like macro substitutions in string-like constants, then they can use LITERALS instead (which are delimited by back-and-forward single-quotes). Examples of Daytona’s use of \texttt{ds\_m4} are found in \$\texttt{DS\_DIR/EXAMPLES/sys/Archie.cy}.

The macro \texttt{\_Define} is the same as \texttt{\_define} except that when it expands, it does not include the new-line that follows its final parenthesis. This enables \texttt{\_include} lines in Cymbal query files to be counted as one line, even though the include file may contain many \texttt{\_Define} calls.

Here is an example of using \texttt{\_Define} to define a macro in such a way that including a file with this definition in it does not change the line numbers of the contents of the expanded file from what they were before the expansion. First, here is the contents of the file, \texttt{mymacs}, with the macro definition:

\begin{verbatim}
\_dnl_
\_Define_\_(AZZA, @<this is line 1 this is line 2 this is line 3>@)
\_dnl_
\end{verbatim}

(\texttt{\_dnl\_} is a macro that expands by deleting the new-line that follows it. There is no need to use \texttt{\_dnl\_} in this example but if the user would like to separate definitions in the macro file, this is one way to do it without introducing unwanted new-lines into the expansion of calls to \texttt{\_include\_}.) Anyway, here is the contents of the file that uses this macro:

\begin{verbatim}
\_include\_(mymacs)
hello
AZZA
goodbye
\end{verbatim}

Try expanding it.

Indeed, to invoke \texttt{ds\_m4} the way that Daytona does, say:

\begin{verbatim}
DS M4 <qfile>
\end{verbatim}

As a reminder, \texttt{ds\_m4} macros with arguments must be invoked with the opening parenthesis immediately following the macro name. For example, "mac1.(x,4)" will work and "mac1 (x,4)" will not.

As mentioned above, Daytona’s use of \texttt{ds\_m4} uses rename.m and sys.macros.m to process each query. Daytona will also process queries with \texttt{project.env.m} if any for the current project if any and the user may define their own \texttt{ds\_m4} macros in a file named \texttt{usr.env.m}. As long as these files can be found in \$\texttt{DS\_PATH}, Daytona will use them automatically -- and not only for processing user queries but also for user Cymbal environment files. Of course, the user can use \texttt{ds\_m4’s} include mechanism to include their own macros.
3.10.1 Macro Facility: Archaic Deprecated cpp Option

Daytona also supports the user specifying the Cymbal/SQL query preprocessor of their choice. For example, this feature enables the user to replace the default m4 with cpp. Just put a Query_Preprocessor Invocation note under the PROJECT description in your pjd. For example,

```bash
<Query_PreprocessorInvocation, ^/usr/local/bin/cpp -P -I$my_includes^>
```

is a valid specification. Please note that for cpp use, -P is mandatory since without it, #line directives will be emitted and cause the Cymbal parser to reject your query.

Unfortunately, some cpp’s are not entirely compatible with Cymbal and DSQL. The following pitfalls must be avoided:

1. some cpp’s are thrown off by the use of Cymbal LITERALS such as ‘foo’.
2. STR constants containing embedded new-lines may confuse cpp.
3. SQL-style strings containing escaped quotes such as ‘Jack O’’Connors’ may be confusing to cpp.
4. The following SQL may be confusing to cpp: where Name like ‘m\%’ escape ‘\’
5. The following SQL may be confusing to cpp:

```sql
for_each_time .city Is_In $[
  -- this SQL comment is too much for some cpp’s
  select City from SUPPLIER where Number > 490 ]$
  do { do Write_Line(.city); } 
```

And the truth is that cpp has really been taken over and subordinated to the needs of C compilers, which really don’t want anyone using cpp standalone.

3.11 Bug Reporting

We make every effort to ensure that the Daytona software runs the way that the documentation says it does. If you are bitten by a bug, please report it so that it can be fixed in the next release. Daytona is quite robust; most bugs take only a nominal amount of time to fix. So please report them so we can get rid of them.

When reporting a bug, please make sure that it is reproducible and please send us enough information for us to reproduce it. Typically, this will include the error message that you get plus a copy of the query and the aar.

Also, please be sure that it can’t be handled by one of the following procedures:

- If a query executable dies with a bus error or segmentation violation, then you may be accidentally overwriting memory. Please regenerate the executable using the -SOC, -ABC, -ZDC, and -NPC options for Tracy: these options will cause additional string overflow, array bounds, and zero-divide checking to take place during execution (see earlier in this chapter). If this doesn’t work, then you have found a Daytona bug. Please report it.
• If you get a message like:

   end of request file encountered unexpectedly

then one of Daytona’s parsers is saying that something is unbalanced in the query. It could be
braces or parentheses or quotes of some kind.

There are two tools you can use to track this down. One is the vi % matching capability
which will enable you to easily find the matching partner of given parentheses, braces, and
brackets.

The other is binary search, which while tedious, is guaranteed to work. Simply divide
your query into two pieces in a syntactically rational way and run the relevant parser on each
piece. At least one will fail. Divide a failing portion into two and recurse. Here is a typical.parser invocation for the Cymbal parser Reducyr:

   DS Reducyr -r <qfile> >/dev/null

• All Daytona queries are scanned by the ds_m4(1) macro processor. Considerable effort has been
taken to restrain ds_m4 from rewriting your query for you in the wrong way. But ds_m4 can
get confused. So, if you get a message from ds_m4, then you know that you have offended it in
some way. Use DS M4 to help you debug.

• If mysterious query executable runtime errors occur, it may be because the indices are out-of-
date and the executable has been invoked with the +T option which prevents this error
condition from being detected. If so, please run the executable without +T. If that doesn’t
work, then something somewhere may be out of sync. To get an entire application back in sync
again, execute "DS Resync <app>" to cause Daytona to rebuild all of its derived files for the
application <app>.

• If immediately after a new DS release comes out, mysterious compilation and runtime errors
appear, please use DS Resync as described above.

• If the parser produces a SYNTAX ERROR after reading some word, always consider the
possibility that it may be a keyword which is being used incorrectly. For example, a SYNTAX
ERROR will result if "then" is used as a variable name.

• If you get an error message which says system error and has not been remediated by running
Tracy with -ABC, -SOC, -ZDC, -NPC, please report it. It’s definitely a Daytona bug.

3.12 Daytona System Limits

Here are the Daytona system limits.

— The maximum DC data file size is currently 30 billion bytes. (This is not the same as the
maximum table size. When using dynamic horizontal partitioning, there is no maximum table
size. Also, keep in mind that one way to avoid hitting this limit is to use either field-
or record-level compression or both. Otherwise, the data file must be split into smaller pieces for
use in a dynamic horizontal partitioning scheme.)
— The maximum length for a record in a DC data file is 65534 bytes. (This is not the same as the maximum size for records in a table. When using vertical partitioning via views, there is no maximum size for records in a table. Furthermore, since this maximum length is for the record as it appears in the DC data file, if that record is field- and/or record-compressed, then the maximum length for the uncompressed record is even greater than 65534 bytes, perhaps by several times. Note that Daytona’s storage implementation is not designed to work efficiently with enormous data file records. Users who need to store big multimedia data like WORD documents, photos, and videos have the option of storing those objects elsewhere in the filesystems and then storing just the file paths to those objects in Daytona.)

— The default maximum length for a key value is 120 bytes. Keep in mind though that since FIELDS with field-compressed values are stored in their compressed form in key values, the default limit on uncompressed key values is actually larger than 120 bytes. (Furthermore, when using Message_Digest, there is no maximum length for a key value, save that imposed by the maximum length for a record in a DC data file.)
4. **DSQL: SQL For Daytona**

In a nutshell, DSQL, Daytona’s SQL, is an extended version of the query and update portion of ANSI standard SQL. All of the SQL DML (i.e., query and update (as opposed to the data definition)) capability of the ANSI 1989 standard is present in DSQL along with several more modern additions such as CASE, INTERSECT, EXCEPT, and SELECTs in the FROM clause. Daytona’s support of the SQL DDL (data definition language) is realized in the standalone executable Daisy which supports the likes of CREATE/DROP TABLE/INDEX/FILE in addition to supporting interactive DML commands. (A man page for Daisy is attached as an appendix and is also accessible by executing DS Man.)

The existence of several widely published texts on SQL makes it unnecessary for another one to appear here. Particularly recommended are the treatments by Date [1984, 1986, 1987]. The documentation provided by other vendors such as Oracle and DB2 forms yet another source, although, of course, the user must take care to note which features of these vendor’s implementations are specific to their software alone. The full DSQL grammar is given as an appendix to this manual.

DSQL is also a fully integrated subset of Cymbal; in fact, DSQL is processed by first translating it into Cymbal, then by processing that like any other Cymbal. DSQL’s integration into Cymbal takes place on 3 dimensions. The first is that DSQL just automatically inherits certain Cymbal functionality: Cymbal’s positional-argument functions and predicates (whether built-in or user-defined) are available just by using them where appropriate. (See cy.sql.d.Q for examples.) Secondly, access to Daytona LIST/SET-valued fields is accomplished just by using the corresponding column names with the understanding that the effect will be to loop over each element of the set as would occur with the likes of one_of_the_Children in Cymbal. Likewise, the second-order Cymbal statistical functions var and stdev are available by simply treating them like any other SQL aggregate function.

The second dimension of integration is due to adding various Cymbal syntactic constructions to DSQL. These are detailed later in this chapter.

The third dimension of integration has to do with the embedding of SQL in Cymbal. In fact, the main reason why Daytona does not currently provide SQL embedded in C is because it is simply much more pleasant to program at a 4GL level with SQL embedded in Cymbal than at a 3GL level with SQL embedded in C. (See also Chapter 18 on Modes Of Use.) For example, in a moment, examples of this integration will show in part how easy it is to write Cymbal/DSQL queries which interrogate the user at runtime for their required execution parameters.

4.1 **Missing Value v. Null Value**

One significant way that DSQL differs from the ANSI standard query and update portion of SQL is that missing values are handled according to Daytona’s missing value philosophy, not according to that of the standard. The essence of Daytona’s missing value philosophy is that if something isn’t there, then it can’t be worked with or more colloquially, you can’t work with something that isn’t there. In particular, for any given record, if a DSQL SELECT makes reference to a FIELD in a condition (in a way that does not use the IS (NOT) NULL predicate), then that record can only be worked with if it
has a value for that FIELD. Note that consequently, in order to be used, records must have values for every FIELD mentioned in the query in a non-IS-NULL way; this includes FIELDS that are just being printed out. Consequently, a DSQL select * can sometimes surprise users because the absence of a value for some FIELD in every record will prevent anything from being printed out.

Another way to look at this situation is that SQL has null values, which are values that mean the absence of a more useful value for that FIELD type. On the other hand, Cymbal does not have somethings that mean nothing; instead Cymbal just supports the notion that it’s possible for the user simply not to have provided a value for a FIELD in a RECORD, in which case the FIELD is said to have a missing value for that RECORD: this is not a something that means nothing, it is a nothing that means nothing and once again, Daytona’s missing value philosophy is: you can’t do anything with nothing.

This and other behavior is sometimes inconvenient and can be partially worked around by the user defining and maintaining their own designated values indicating missing or unavailable data as might be done, for example, by using Default_Value = "" for STR FIELDS or 1E78 for FLT FIELDS or -1 for INTs. Specifically, having a Default_Value for a FIELD specified in the rcd (see Chapter 3) means that any time there is no value given for that FIELD in some data record, then the Default_Value is used instead. Such a Default_Value in effect becomes the user-specified ‘in-band’ special null value for that FIELD.

Note that both missing values and separately Default_Value as being used here require the user to specify in the rcd that missing values are allowed in data file records. This is achieved by using (0->1) multiplicity prefix in the Type specification for the FIELD as in:

```#
{ FIELD ( Source )
  <Type, (0->1) STR>
  <Default_Value, "">
  <__Position, 2> }
```

To avoid using these user-defined null values in computations, the user must remember to include restrictions like Salary ! 1E78 in their where clauses. This is the kind of logic that Daytona automatically employs when its missing value handling facilities are being used. Indeed, when one disables Daytona’s missing value handling by using a Default_Value, then one has lost the advantage of automatically calculating statistics correctly. For example, suppose one wants to compute the average weight of parts. This is done correctly (and automatically!) using Daytona’s default missing value handling because not only do parts that have no weight recorded contribute nothing to the total weight but furthermore and importantly, they contribute nothing to the count of parts that is the denominator used in computing the average weight. On the other hand, if Daytona’s missing value handling is disabled by using a Default_Value of 0, then while all those parts with weight 0 will contribute nothing to the total weight, they do contribute to the computation to the total number of parts, which simply invalidates the computation of the average. Worse yet is what would happen should the Default_Value be 1E78 and records which contain that value are not explicitly excluded by explicit Cymbal tests in the query: obviously, contributing 1E78 repeatedly to the total salary is the wrong thing to do here.

A query-by-query way to avoid records being rejected because they have missing field values is to use the but_if_absent keyword described in Chapter 13. This keyword enables a there_isa to be modified in such a way that for any designated FIELD, if that FIELD is missing a value, then alternative logic is
to be followed. Clearly, this does not entail modifying the rcd and changing the behavior of all queries against that RECORD
CLASS; instead it works locally one there is a in a query at a time. Details in Chapter 13.

Lastly, if the sole objective is to print out records from a table, and simply take no printing action whatsoever whenever a missing field value is encountered, then that can be achieved by using the Cymbal Describe PROC or its shell-level embodiment, DS Show. Since Describe is confined to just printing what amounts to be a report, it is able to take a different yet consistent approach to handling missing values: instead of skipping over any record that contains them, Describe simply fails to print anything upon encountering a missing FIELD value. This enables Describe to produce printed tables that contain empty areas where a FIELD value might otherwise occur. So, while still printing out the FIELD values that are present and thus processing the record, not skipping it, Describe still refuses to do something/anything in response to encountering nothing. This is still not the SQL approach of printing out lots of nulls but at least, records are not being skipped just because of one missing FIELD value. See Chapter 13 for more on Describe.

4.2 Duplicate Answer Nondeterminism

The other significant way that DSQL differs from the ANSI standard is in the way that it produces duplicate answer tuples, i.e., rows in the answer table that are absolutely identically the same. The semantics of ANSI SQL specify very precisely when duplicate answer tuples will be produced and when they won’t. However, DSQL inherits Daytona’s philosophy with regards to duplicates, which is that while the system guarantees that every distinct answer will be produced at least once, there are no assurances as to how many times it will be produced. As a result, Daytona can optimize queries so as to process them more efficiently (by avoiding the production of duplicates); in fact, as Daytona improves over time, it will become more and more oriented towards automatically eliminating all duplicate answers. In other words, if an SQL answer table contains duplicates, DSQL will produce all of the distinct tuples, but it won’t necessarily replicate each one the number of times ANSI SQL semantics would call for. Remember: both Cymbal and ANSI SQL are processed in ways that can produce duplicate answers: they differ in how many duplicates may be produced.

(These differences arise because DSQL is translated into Cymbal (declarative) assertions and Cymbal is processed as a logic/deductive database language, which simply does not have the same semantics as SQL. In a deductive database, duplicate answers correspond to different derivations of the same fact: that is prima facie redundant and possibly inefficient. (In some cases, getting rid of duplicates requires sorting, which is expensive.) So, Daytona makes no apologies for eliminating duplicates, when it is efficient to do so.)

There are situations however when the user is actually interested in the duplicate answers that ANSI SQL produces, as could only be the case if they wanted to count them: indeed, what is the total information carried by a list of \( n \) duplicate answers? it is the count \( n \) and any one of the duplicates. This kind of count information can be guaranteed to be provided to the Daytona DSQL and Cymbal user by making suitable modifications to the query so as to, in effect, give each duplicate its own unique tag, implying that the modified answer is no longer a duplicate! In other words, if the user wants the information provided by SQL duplicates then Daytona can guarantee that the user will get it if the user’s interest in
the duplicates is indicated in their queries by writing them in such a way that each duplicate is made into a unique answer. In short, regarding duplicates, Daytona used appropriately can get the same information out of the database that ANSI SQL can. Your Daytona representative can provide you with more information on this issue as needed.

As a consequence of all this, various counts will vary. In particular, \( \text{count(*)} \) will cause Daytona to try to generate as many duplicates as possible and then count them, whereas \( \text{count(distinct Field_1)} \) will count what is left after all duplicates are removed. On the other hand, \( \text{count( Field_1)} \) will produce a number in between. For an example, see suppred.3.S. This is related to \( \text{count GroundEnumeration} \) as discussed in Chapter 14 for Cymbal, remembering of course that all DSQL is translated into Cymbal. In that regard, \( \text{count(*)} \) is treated as a GroundEnumeration count unless other aggregates are being requested at the same time: see the documentation on the \( \text{aggregates()} \) function which is how these multiple SQL aggregate function calls are implemented in Cymbal.

Also, regarding duplicates, INTERSECT ALL and EXCEPT ALL will return duplicates but only those that are present in the first operand in the quantity that they are present there. This is not consistent with the standard.

### 4.3 Using DSQL In The Daytona Environment

The easiest way to run DSQL is to do so interactively with Daisy. The second easiest way is to use DS QQ, as described in Chapter 3 and the tutorial "Getting Started With Daytona Step-By-Step" (a separate document). The invocation DS QQ whatever.S will cause the system to look around in the file hierarchy neighborhood ./.Q/./.Q/./.Q/. for the file whatever.S and, finding it or no, will call up the user’s text editor on the contents of that file. The point is that the .S tells Daytona that the contents of the associated file are 100% DSQL (i.e., no Cymbal statements like for_each_time, etc.).

At any rate, consider now the issue of how to write DSQL queries. The key point here is that the ANSI standard portion of Daytona’s implementation of SQL is, by and large, not case sensitive, although of course any added Cymbal syntax does remain case-sensitive. This means that the SQL portion of DSQL queries can be typed consistently using any mixture of upper- and lower-case letters, with the 4 exceptions that, for use in hybrid Cymbal-SQL, the UPLOWS Select, Insert, Delete, and Update are not considered to be SQL keywords (and are so available to be either (Cymbal) PROC or PRED). Important: note the use of the word consistent: the same item must be capitalized consistently throughout the SQL query. For example, if a query sometimes uses PART.Number and sometimes uses part.number, then Daytona may refuse to process the query. Consistency is a virtue here because there is no point in prompting other readers of the query to ask the question, "I thought that these two columns were the same, so how come they are capitalized differently?".

There are two other ways to run DSQL queries: DS Stacy and DS Tracy. DS Stacy is used for running pure DSQL; the only parser that Stacy uses is the DSQL parser Squirrel. DS Tracy is used for processing Cymbal queries, DSQL queries, and hybrid Cymbal-DSQL queries; its parser is the Cymbal parser Reducyr and Reducyr will call Squirrel as needed. DS Stacy is the one to invoke for pure DSQL. DS Tracy will accept any DSQL query enclosed by $[ and ]$ and otherwise, will accept any DSQL query as it is as long as it does not use the SQL -- double-hyphen comments and does not begin with a parenthesis or with the begin SQL keyword. Any query file that ends with the .S suffix will be handled as if DS Stacy was called, even if DS Tracy was.
If the user wishes to write exclusively in DSQL, then by setting the shell variable `DS_SQLONLY` equal to `y`, they can ensure that the Squirrel parser is always called by Daytona shell scripts, instead of the Cymbal parser Reducyr. So, for example, when `DS_SQLONLY` is `y`, then `DS Compile` will process only queries that are written in pure DSQL.

Information from the shell environment may be carried to the SQL parser, Squirrel, by means of the shell variable `DS_SFLAGS`, whenever Squirrel is called by Daytona commands. For example, the keyword `-sql_out_fmt` is used to specify the Display format type for the output. Consequently, the arguments for this keyword are: `_table_`, `_packet_`, `_desc_`, `_xml_`, `_safe_`, and `_data_` with `_data_` being the default. Likewise, `-sql_out_sep` and `-sql_out_com_ch` convey the user’s specifications for output field value separator and output file comment character, respectively, which are only allowed and of value when the table format is `_data_`. Lastly, `-sql_out_no_heading` can be used with `-sql_out_fmt_data_` so as to suppress the heading comments. `DS_SFLAGS` can be set at the entry to some shell or it can be set in an ad hoc manner by using a convenient shell convention, as in:

```bash
DS_SFLAGS=" -sql_out_sep '|' -sql_out_com_ch '#' -sql_out_fmt _table_ "
// or on a cmd by cmd basis
DS_SFLAGS=' -sql_out_fmt _table_ ' DS Compile stpaul.1.S
```

If `-logging` is a substring of `$DS_SFLAGS`, then all SQL updates will be done as logging transactions. Notice that `DS_SFLAGS` exerts its effect at compile-time, not run-time.

Alternatively, if `-bombproof_reads` is a substring of `$DS_SFLAGS`, then all there isa’s generated by Squirrel (in response to an SQL query) will contain the keywords `-ignoring_failed_opens` and `-with_dirty_reads_ok`.

Multiple DSQL statements may be sequenced together with intervening semicolons. In pure DSQL (i.e., no surrounding Cymbal), semicolons are separators, not terminators: a syntax error will result if the last character that Squirrel sees is a semicolon. This practice conforms to the ANSI standard. DSQL users who would like to use semicolons in the same free and easy way they are used in Cymbal should send their DSQL queries through the Cymbal parser. As mentioned above, the 100% failsafe way to do this is to enclose them in `$[ ]$`.

### 4.4 Daytona Extensions To Standard SQL

Hatted Cymbal constants like the DATE `^1999-04-30` and `^12.3.192.2.IP` are welcome. As a convenience, SQL-3 standard timestamp syntax is supported as in `^2004-11-11 11:11:00` (note the absence of an `@`-sign, which one would find in Cymbal DATE_CLOCK, which are also welcome). Incidentally, DSQL does support the SQL TIMESTAMP datatype as in `TIMESTAMP ‘7-24-01 2:33:05’`.

The Cymbal keyword-argument function and predicate call syntax has been added to DSQL so as to provide access to many Cymbal built-in functions and predicates which use this argument syntax. When using such keyword-argument fpps in DSQL, it may be necessary to enclose the keywords in hats. In addition, SQL statements may reference Cymbal array elements as constants in comparisons. Also, as regards both of these dimensions, users are free to add their own positional-argument functions and predicates to SQL by writing them either in Cymbal or C.
DSQL supports using the SELECT list item AS-clause as a true alias and not just as a column label for use in the resulting table. The as-alias can be used in the GROUP-BY, HAVING and ORDER-BY clauses. This convenient extension is most definitely not part of the standard. An example should make this clear (case.1.S):

```sql
select count(*),
case
  when Weight < 2.0 then "light"
  when Weight < 5.0 then "medium"
  else "heavy"
end
as wt_category,
10* sum( rtn(Weight, 1.0) )
from PART
group by wt_category
having wt_category = "light" or wt_category = "medium"
order by wt_category
```

Finally, with a small syntactic modification, Cymbal casting from one type to another can be done in DSQL. The rule is just to use (: and :) instead of parentheses to enclose the Cymbal type expression. So, for example:

```sql
select *
from SUPPLIER
where (:STR:) Number Is_A_Substr_Of Telephone
```

Note the use of the Cymbal predicate Is_A_Substr_Of. Also, as a special convenience, Daytona offers the (:FIELD:) cast. This cast casts its term to whatever type is associated with the FIELD that term is being compared to in some way. These examples should make this clear:
for_each_time .nbr Is_In [ 1 -> 10 ]
{
    update ^ORDER^
    set Quantity = 1.1 * Quantity
    where Number > (:FIELD:) .nbr ;
}
select *
from ^ORDER^
where Supp_Nbr = (:FIELD:)’404’
    and Number not in ( (:FIELD:) "618", (:FIELD:) "684" );

set [ .number, .name, .color, .weight, .intro_date ]
    = read( from _cmd_line_ ) otherwise do Exit(1);

  // can only use (:FIELD:) here if casting a VBL dereference
insert into PART (Number, Name, Color, Weight, Intro_Date )
    values( (:FIELD:) .number, (:FIELD:) .name, (:FIELD:) .color,
    (:FIELD:) .weight, (:FIELD:) .intro_date );

Standard SQL regular expression matching is limited. So, DSQL users have access to Cymbal’s
regular expression matching predicate Matches which recognizes all the standard UNIX regular
expression metacharacters.

Daytona’s INTERVAL syntax has been added so that one can write the likes of:

    select * from SUPPLIER
    where Number in [ ^400^INT(_short_) -> 425 by 5 ]

DSQL supports using the ‘as’ labels on aggregate function select list items in the having and order
by clauses:

    select supp_nbr, avg(Quantity) as Avg_Quantity
    from ^ORDER^
    group by supp_nbr
    having  Avg_Quantity > 3000.0
    order by Avg_Quantity

Even though DSQL does not support SQL null values, DSQL does have a way to support left
outer joins. This is done by relying on what are called dummy values in the Cymbal context. See
Chapter 13 for a full description of how to do left, right, and full outer joins in both DSQL and Cymbal.
In DSQL, they can look like this:
select S.Name,
    sql_if_dummy_then(O.Number,""),
    sql_if_dummy_then(O.Quantity,""
from SUPPLIER as S left join `ORDER` as O
on S.Number = O.Supp_Nbr
where S.Number >= 500 and 0.Quantity > 3000

4.5 Examples Mixing DSQL With Cymbal

The next query (suppavgqty.1.S) illustrates the use of Cymbal functions in formatting the results of SQL queries. This query prints out the total and avg quantities supplied by suppliers whose cities have names beginning with "New".

```sql
-- total and avg qtys supplied by "New.*" suppliers
select supplier.name, supplier.city,
    count(*) as `Count`,
    str_for_dec( avg(quantity), 3 ) as `Avg`
from `order`, supplier
where supplier.number = `order`.supp_nbr
    and supplier.city Matches "New.*"
group by supplier.name, city
having count(*) > 2
order by supplier.name
```

First, syntactically speaking, Cymbal /**/ and // comments and ANSI standard -- double-hyphen comments are welcome in DSQL.

Second, a word about the hats or carets. SQL keywords like "avg" and "count" can be used literally in DSQL queries if they are enclosed in hats, as is the case with `order` as well. The function of the hats is to turn off the special meaning of the words as defined by SQL. This is a very useful convention. In fact, it is used again for Cymbal keyword-argument sequences where in order to distinguish between DSQL column references and Cymbal keywords, it is necessary to enclose the keywords with hats. The keyword-argument Cymbal functions currently supported are whence, index, substi, gsubsti, and translate. Note that standard SQL has a similar "escape" capability called <delimited identifier> which uses double quotes instead of hats. DSQL does not use double quotes for this purpose because it wishes to allow users to use double quotes to delimit strings, as is done in Cymbal. Once again, DSQL uses hats instead.

The as argument gives the user the opportunity of specifying a column label for the tabular output of the query. As per the SQL standard, this label may be a regular SQL column name. DSQL extends this definition by allowing the label to be either a hatted quantity (as illustrated above) or an SQL or Cymbal string constant. These string constants can of course contain any ASCII character. An advantage of using a Cymbal string constant is that, since they are considered to be ds_m4 comments, no ds_m4 macro substitutions can be performed on their contents.

By default, Daytona prints out floating point numbers with maximum precision. Since this can
well result in seeing some 15 digits per number, the Cymbal functions \textit{rtn} or \textit{str\_for\_dec} can be used to round to a more attractive number of decimal places. In fact, all of Daytona’s output formatting functions are available so that the user can use \textit{lb}, \textit{cb}, \textit{rb}, etc. to format their own SQL output.

The next query (union.2.Q) can only be read by Tracy since it involves Cymbal-DSQL hybrids. Cymbal \textit{Write} statements precede each of 2 DSQL queries: the first is a simple \textit{SELECT} and the second is a \textit{UNION} query. The use of the \$[ \]\$ is required for this \textit{UNION} query because it begins with a left parenthesis.

\begin{verbatim}
do Write_Line( "Suppliers whose Names begin with a or A" );
select Name as Supplier
from SUPPLIER
where lower_of(Name) Matches "^a+" ;

do Write_Line( "Another union of part_nbrs in two ranges");
$
 (select Part_Nbr
 from `ORDER``
 where Part_Nbr > 120
 and Part_Nbr < 140
 union all
 select distinct Part_Nbr
 from `ORDER``
 where Part_Nbr > 160
 and Part_Nbr < 180 )$
\end{verbatim}

Actually, the \$[ \]\$ could be extended to encompass both SQL statements.

Here is a sequence of semicolon-separated \textit{Display} calls involving SQL (cysql.IQ):
do Display
  each[ .number, .part ]
  each_time(
    [ .number, .part ] Is_In $[ select number, name from part where number < 105 ]$
  );
when( .target Is_In $[ select self from employee_
    where location = 'Philadelphia' ]$
    and .target != "Steve McCracken"
  ) do {
    skipping 2 do Write_Line( "Found one!" );
  };
  do Display
  with_title_line "Get phone numbers of suppliers from St. Paul"
  with_labels [ "Supplier", "Phone" ]
  each_tuple_of
  $[ select Name , Telephone
    from SUPPLIER
    where City = 'St. Paul'
  ]$

  The first two subqueries show that SQL query results can be used as aggregate arguments to Is_In
  assertions. When used in this setting, the $[ sql-stmt ]$ construct plays the same syntactic role as that of
  a LIST of (query answer) TUPLES. The last example illustrates that SQL can even be used for the bulk
  of a Display call.

  Depending on where the $[ ... ]$ appears in a Cymbal query, it may be considered the equivalent
  of a Display call or a BOX -- or in fact, what amounts to be an OPCOND. While OPCOND is an
  advanced concept, it can be used here behind the scenes to avoid the inefficiencies due to using a BOX.
  In other words, Daytona will create a BOX to store all the answers of an SQL query if Is_In is used. If
  there are a lot of answer TUPLES, then this can take quite a bit of effort in terms of time and memory
  for the system to achieve. The way to avoid this cost is to use Is_Selected_By instead of Is_In. While on
  the one hand, Is_Selected_By does not have the keyword options that Is_In does, Is_Selected_By does
  avoid the cost of making a BOX. Is_Selected_By can only be used if its argument is a LIST, not a SET:
  SQL group-by’s for example produce SETS and so are not compatible with Is_Selected_By. Here is an
  example of its use (cy.sql.3.Q):

  for_each_time [ .x, .y ] is_such_that( [ .x, .y ] Is_Selected_By
    $[ select Number, Name from supplier
      where Number < 405 order by Name ]$
  ) do Write_Words( .x, .y );

  The last two examples (stpaul.sql.IQ and cysql.IQ) show how to use Cymbal to gather runtime
arguments for SQL queries from both the command line and stdin.

```python
locals: STR: .city_pat
set [ .city_pat ] = read( from _cmd_line_ );
select Name as Supplier, City, Telephone as Phone
from SUPPLIER
where City Matches .city_pat ;
do {
local: STR: .city_pattern
skipping 2
do Exclam( "Enter city pattern: " );
set [ .city_pattern ] = read_line();
for_each_time [ .supplier, .city, .phone ] Is_Selected_By
  
  select Name, City, Telephone
  from SUPPLIER
  where City Matches .city_pattern
}
```

Notice that the second query goes further and takes over complete control of both the SQL query’s input handling and output formatting. Specifically, this query prompts the user at runtime for a city pattern and then prints out supplier information in "packet" format by using, in part, Cymbal’s `lb` or left justify string in block function. (Of course, one could also use an SQL-based Display with_format _packet_.)

To get some idea of the size of SQL queries that Daytona can handle, take a look at $DS_DIR/EXAMPLES/usr/orders/Q/having.S.

### 4.6 Parallel DSQL

The way to parallelize a DSQL query is to define the first table in the FROM clause to be a view table that is defined in Cymbal using "easy parallelization" and whose definition is put in a *.env.cy* file for all to use. Then basically any query which puts that view table as the first table in its FROM clause will be parallelized. See Chapter 15 for details on how to do this.
4.7  row_number And Top-K In DSQL

row_number() is a virtual column in SQL that is known as a window function for reasons too arcane to get into here. Suffice it to say that it produces a sequence of integers starting with 1 that are associated with rows in the result table of a SELECT. As will be seen, this basic capability is leveraged to express top-k queries. Even though DSQL adheres to SQL standards and practices for these concepts, they are well worth discussing here.

4.7.1  row_number In DSQL

In its simplest incarnation, row_number() just provides sequence numbers for the result table rows. Consider this query from rownbr.sql.3.Q:

```sql
select row_number() as RowNum, Number, Name, City 
from SUPPLIER
where Number between 450 and 460
```

Here is the corresponding output table:

<table>
<thead>
<tr>
<th>RowNum</th>
<th>Number</th>
<th>Name</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>450</td>
<td>Leda Enterprises</td>
<td>Seattle</td>
</tr>
<tr>
<td>2</td>
<td>451</td>
<td>Glaucus Leasing</td>
<td>London</td>
</tr>
<tr>
<td>3</td>
<td>452</td>
<td>Julius Receiving</td>
<td>St. Paul</td>
</tr>
<tr>
<td>4</td>
<td>453</td>
<td>Caesar Import-Export</td>
<td>Minneapolis</td>
</tr>
<tr>
<td>5</td>
<td>454</td>
<td>Pompey Sales</td>
<td>Indianapolis</td>
</tr>
<tr>
<td>6</td>
<td>455</td>
<td>Antony Sales</td>
<td>Phoenix</td>
</tr>
<tr>
<td>7</td>
<td>456</td>
<td>Mark Leasing</td>
<td>Albuquerque</td>
</tr>
<tr>
<td>8</td>
<td>457</td>
<td>Hannibal Leasing</td>
<td>Wichita</td>
</tr>
<tr>
<td>9</td>
<td>458</td>
<td>Carthage Export</td>
<td>Newark</td>
</tr>
<tr>
<td>10</td>
<td>459</td>
<td>Plymouth Shipping</td>
<td>Fairlawn</td>
</tr>
<tr>
<td>11</td>
<td>460</td>
<td>Ashurbanipal Shipping</td>
<td>New York</td>
</tr>
</tbody>
</table>

Note that the parentheses following row_number() are mandatory. All the queries in this section are from rownbr.sql.3.Q.

While supported and useful, the use of row_number() above is actually not standard SQL but rather a convenient abbreviation. This is because the standard requires an over() expression to accompany row_number() as in:

```sql
select row_number() over() as RowNum, Number, Name, City 
from SUPPLIER
where Number between 450 and 460
```

This query yields the same result as before.
over() becomes more interesting when it contains an order by specification:

```sql
select row_number() over( order by Quantity ) as RowNum,
    "ORDER".Number as Onbr, Name as Supplier,
    SUPPLIER.Number as Snbr, City, Quantity
from "ORDER", SUPPLIER
where "ORDER".Supp_Nbr = SUPPLIER.Number and
    SUPPLIER.Number between 450 and 451
```

which specifies that the row numbers be assigned in the order of increasing Quantity:

<table>
<thead>
<tr>
<th>RowNum</th>
<th>Onbr</th>
<th>Supplier</th>
<th>Snbr</th>
<th>City</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>151</td>
<td>Glaucus Leasing</td>
<td>451</td>
<td>London</td>
<td>896</td>
</tr>
<tr>
<td>2</td>
<td>699</td>
<td>Leda Enterprises</td>
<td>450</td>
<td>Seattle</td>
<td>1203</td>
</tr>
<tr>
<td>3</td>
<td>607</td>
<td>Glaucus Leasing</td>
<td>451</td>
<td>London</td>
<td>1607</td>
</tr>
<tr>
<td>4</td>
<td>621</td>
<td>Glaucus Leasing</td>
<td>451</td>
<td>London</td>
<td>1900</td>
</tr>
<tr>
<td>5</td>
<td>719</td>
<td>Leda Enterprises</td>
<td>450</td>
<td>Seattle</td>
<td>1963</td>
</tr>
<tr>
<td>6</td>
<td>381</td>
<td>Leda Enterprises</td>
<td>450</td>
<td>Seattle</td>
<td>1995</td>
</tr>
<tr>
<td>7</td>
<td>235</td>
<td>Leda Enterprises</td>
<td>450</td>
<td>Seattle</td>
<td>2606</td>
</tr>
<tr>
<td>8</td>
<td>914</td>
<td>Glaucus Leasing</td>
<td>451</td>
<td>London</td>
<td>2808</td>
</tr>
<tr>
<td>9</td>
<td>655</td>
<td>Leda Enterprises</td>
<td>450</td>
<td>Seattle</td>
<td>2988</td>
</tr>
<tr>
<td>10</td>
<td>23</td>
<td>Leda Enterprises</td>
<td>450</td>
<td>Seattle</td>
<td>3383</td>
</tr>
<tr>
<td>11</td>
<td>295</td>
<td>Leda Enterprises</td>
<td>450</td>
<td>Seattle</td>
<td>3655</td>
</tr>
<tr>
<td>12</td>
<td>394</td>
<td>Leda Enterprises</td>
<td>450</td>
<td>Seattle</td>
<td>4619</td>
</tr>
<tr>
<td>13</td>
<td>963</td>
<td>Glaucus Leasing</td>
<td>451</td>
<td>London</td>
<td>4745</td>
</tr>
<tr>
<td>14</td>
<td>965</td>
<td>Leda Enterprises</td>
<td>450</td>
<td>Seattle</td>
<td>4834</td>
</tr>
</tbody>
</table>

If it were desired to order by decreasing quantity, the following specification would be used:

```sql
row_number() over( order by Quantity desc )
```

This order by specification follows the same syntactic rules that hold for the order by clause for a SELECT. In particular, multiple columns can be specified in an order by.

Suppose one wanted to place a constraint on a row_number() column. Since the SQL standard forbids referring to a SELECT column in the body of that SELECT, it is necessary to do that constraining in an enclosing SELECT:
select * from
(select
    row_number() over( order by City desc, Quantity asc ) as RowNum,
    "ORDER".Number as Onbr, Name as Supplier,
    SUPPLIER.Number as Snbr, City, Quantity
from "ORDER", SUPPLIER
where "ORDER".Supp_Nbr = SUPPLIER.Number and
    SUPPLIER.Number between 450 and 451
) as subq // the standard requires a subquery name!!
where RowNum % 2 = 0

Here then are the even-numbered rows as the result:

<table>
<thead>
<tr>
<th>Rownum</th>
<th>Onbr</th>
<th>Supplier</th>
<th>Snbr</th>
<th>City</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>719</td>
<td>Leda Enterprises</td>
<td>450</td>
<td>Seattle</td>
<td>1963</td>
</tr>
<tr>
<td>4</td>
<td>235</td>
<td>Leda Enterprises</td>
<td>450</td>
<td>Seattle</td>
<td>2606</td>
</tr>
<tr>
<td>6</td>
<td>23</td>
<td>Leda Enterprises</td>
<td>450</td>
<td>Seattle</td>
<td>3383</td>
</tr>
<tr>
<td>8</td>
<td>394</td>
<td>Leda Enterprises</td>
<td>450</td>
<td>Seattle</td>
<td>4619</td>
</tr>
<tr>
<td>10</td>
<td>151</td>
<td>Glaucus Leasing</td>
<td>451</td>
<td>London</td>
<td>896</td>
</tr>
<tr>
<td>12</td>
<td>621</td>
<td>Glaucus Leasing</td>
<td>451</td>
<td>London</td>
<td>1900</td>
</tr>
<tr>
<td>14</td>
<td>963</td>
<td>Glaucus Leasing</td>
<td>451</td>
<td>London</td>
<td>4745</td>
</tr>
</tbody>
</table>

`row_number()` can of course be used to provide sequence numbers for the output of group-by queries:

select * from
(select
    row_number() over( order by avg(Quantity) ) as RowNum,
    City,
    rtn(avg(Quantity), .001) as Avg_Qty,
    sum(Quantity) as Tot_Qty
from "ORDER", SUPPLIER
where "ORDER".Supp_Nbr = SUPPLIER.Number and
    SUPPLIER.Number between 450 and 475
group by City
) as subq
where RowNum between 5 and 10

Note that this causes each row/group of the result to get its own row number in order of increasing total quantity.
This is simply not the same kind of thing as separately numbering the individuals within each group, a subject which is covered next.

### 4.7.1.1 Numbering Individuals Within Each Group

SQL uses the `partition by` keywords within the `over` fragment of a `row_number` to specify that individuals within a group are to be numbered separately from all the other groups. Here is an example (`rownbr.sql.4.Q`):

```sql
select * from (select row_number() over (partition by SUPPLIER.Name order by PART.Name ), SUPPLIER.Name, PART.Name, Quantity from SUPPLIER, "ORDER", PART where "ORDER".Supp_Nbr = SUPPLIER.Number and "ORDER".Part_Nbr = PART.Number ) as subq (Row_Nbr, Supplier_Name, Quantity, Part_Name) order by Supplier_Name
```

Here is a fragment of the output:
<table>
<thead>
<tr>
<th>Row_Nbr</th>
<th>Supplier_Name</th>
<th>Quantity</th>
<th>Part_Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Macedonian Limited</td>
<td>belt</td>
<td>3335</td>
</tr>
<tr>
<td>2</td>
<td>Macedonian Limited</td>
<td>fan</td>
<td>891</td>
</tr>
<tr>
<td>3</td>
<td>Macedonian Limited</td>
<td>fastener</td>
<td>2736</td>
</tr>
<tr>
<td>4</td>
<td>Macedonian Limited</td>
<td>nail</td>
<td>4593</td>
</tr>
<tr>
<td>5</td>
<td>Macedonian Limited</td>
<td>pencil</td>
<td>216</td>
</tr>
<tr>
<td>1</td>
<td>Odysseus Import</td>
<td>antenna</td>
<td>2207</td>
</tr>
<tr>
<td>2</td>
<td>Odysseus Import</td>
<td>bolt</td>
<td>2706</td>
</tr>
<tr>
<td>3</td>
<td>Odysseus Import</td>
<td>fan</td>
<td>1474</td>
</tr>
<tr>
<td>4</td>
<td>Odysseus Import</td>
<td>handle</td>
<td>1999</td>
</tr>
<tr>
<td>5</td>
<td>Odysseus Import</td>
<td>jack</td>
<td>362</td>
</tr>
<tr>
<td>6</td>
<td>Odysseus Import</td>
<td>knob</td>
<td>4823</td>
</tr>
<tr>
<td>7</td>
<td>Odysseus Import</td>
<td>pattern</td>
<td>2913</td>
</tr>
<tr>
<td>8</td>
<td>Odysseus Import</td>
<td>sealant</td>
<td>1707</td>
</tr>
</tbody>
</table>

Clearly, each group is numbered separately.

So, one might ask, if there are groups, where is the *group by* clause? Good question. There is no *group by* clause but yet there are groups nonetheless. The *partition by* keywords are followed by a comma-separated sequence of column references, which serve to instruct the system as to how to group the output for the purpose of generating row numbers and for that purpose only. The use of *partition by* is incompatible with simultaneously using *group by* in the same SELECT-FROM-WHERE because when using *group by*, there is only one row output per group whose purpose is to contain/display aggregate group characteristics.

### 4.7.2 Top-K In DSQL

Top-k queries seek to identify individuals that are top performers according to some ordering. They use a restricted form of the syntax used above for the last two *row_number()* queries. What distinguishes them from their brethren is that the restricted form of their syntax causes the system to implement them in a particularly efficient manner where at no point in the execution of the top-k query are more than \( k + 1 \) rows stored. This efficiency is requested simply by specifying that the *row_number()* be bounded from above. Here is such a top-k query (*rownbr.sql.3.Q*):

```sql
select *
from (  
    select row_number() over( order by Weight ),
    Number, Name, Weight
    from PART
    where Name Matches "a"
) as subq (Row_Nbr, PNumber, PName, Weight)
where Row_Nbr <= 10
```

with corresponding output table:
<table>
<thead>
<tr>
<th>Row_Nbr</th>
<th>Pnumber</th>
<th>Pname</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>102</td>
<td>keyboard</td>
<td>1.1</td>
</tr>
<tr>
<td>2</td>
<td>148</td>
<td>cabinet</td>
<td>1.1</td>
</tr>
<tr>
<td>3</td>
<td>136</td>
<td>sandpaper</td>
<td>1.3</td>
</tr>
<tr>
<td>4</td>
<td>158</td>
<td>magnetic tape</td>
<td>1.3</td>
</tr>
<tr>
<td>5</td>
<td>173</td>
<td>antenna</td>
<td>1.3</td>
</tr>
<tr>
<td>6</td>
<td>106</td>
<td>washer</td>
<td>1.5</td>
</tr>
<tr>
<td>7</td>
<td>174</td>
<td>paper clip</td>
<td>1.5</td>
</tr>
<tr>
<td>8</td>
<td>194</td>
<td>disc brakes</td>
<td>1.5</td>
</tr>
<tr>
<td>9</td>
<td>186</td>
<td>NiCad battery</td>
<td>1.7</td>
</tr>
<tr>
<td>10</td>
<td>196</td>
<td>radio</td>
<td>1.7</td>
</tr>
</tbody>
</table>

This would no longer be a top-k query if the constraint was instead \( \text{Row\_Nbr} > 10 \), in which case the system would have to compute the unconstrained result table (which could be arbitrarily large) and then produce all but the first 10 rows in the sort. To qualify the query as a top-k query, the predication that bounds \( \text{row\_number()} \) from above must be either the sole contents of the outer \( \text{SELECT} \)'s \( \text{WHERE} \) clause or else that \( \text{WHERE} \) clause must be a conjunction that includes that predication as a conjunct.

To get the bottom-k, i.e., the top-k for the opposite sort, just sort in the descending direction with the likes of:

\[
\text{select row\_number()} \text{ over( order by Weight desc )},
\]

In that case, the output is this:

<table>
<thead>
<tr>
<th>Row_Nbr</th>
<th>Pnumber</th>
<th>Pname</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>126</td>
<td>plate</td>
<td>9.7</td>
</tr>
<tr>
<td>2</td>
<td>129</td>
<td>nail</td>
<td>9.5</td>
</tr>
<tr>
<td>3</td>
<td>119</td>
<td>can opener</td>
<td>9.3</td>
</tr>
<tr>
<td>4</td>
<td>134</td>
<td>clasp</td>
<td>9.3</td>
</tr>
<tr>
<td>5</td>
<td>155</td>
<td>tape</td>
<td>9.3</td>
</tr>
<tr>
<td>6</td>
<td>156</td>
<td>eraser</td>
<td>9.1</td>
</tr>
<tr>
<td>7</td>
<td>180</td>
<td>sealant</td>
<td>9.1</td>
</tr>
<tr>
<td>8</td>
<td>128</td>
<td>hammer</td>
<td>7.9</td>
</tr>
<tr>
<td>9</td>
<td>184</td>
<td>AA battery</td>
<td>7.9</td>
</tr>
<tr>
<td>10</td>
<td>162</td>
<td>3-way outlet</td>
<td>7.5</td>
</tr>
</tbody>
</table>

Here the relevant PARTs are presented in the order of decreasing Weight. This would put them in the bottom 10 of an ordering by increasing Weight but of course, they are the top-10 in order of decreasing Weight.

Top-k queries are unusually amenable to being parallelized. As a rule, DSQL queries are
parallelized by presenting them as a view of corresponding parallelized Cymbal code. To see an example of how this is done, see the view definition of PARA_TOPK_ORDERA2 in orders.env.cy in the test suite and observe that using it in DSQ is as simple as writing:

```sql
local: INT .max_count = 20
select * from PARA_TOPK_ORDERA2
parallel for 4;
```

If one wanted to use 100% pure DSQ, then the value for \( k \) would have to be a constant, instead of the expression `.max_count` as it is here. Chapter 14 contains a discussion of top-k queries as written in Cymbal.

As an implementation note, the contents of a DSQ top-k query table are kept in a LIST[ TUPLE ] box, not a SET[ TUPLE ] box, although both options are available in the Cymbal alternative.

### 4.7.2.1 Computing The Top-K For Each Group

Given the machinery provided by `row_number` and the previous discussion of how to use that to construct top-k queries, it is very easy to write (and understand!) queries that compute the top-k for each group (top-k.sql.2.Q):

```sql
select *
from (select row_number() over (partition by SUPPLIER.Name order by Quantity desc),
        SUPPLIER.Name, Quantity, PART.Name
    from SUPPLIER, `ORDER`, PART
    where "ORDER".Supp_Nbr = SUPPLIER.Number
    and "ORDER".Part_Nbr = PART.Number
) as subq (Row_Nbr, Supplier_Name, Quantity, Part_Name)
where Row_Nbr <= 4
order by Supplier_Name
```

This query computes the top 4 ORDERs by descending Quantity value for each SUPPLIER and then orders the entire result set alphabetically by Supplier_Name.

### 4.8 Updating In DSQ

Firstly, as a convenience for the user which leads to a more efficient implementation that minimizes the number of transactions, the SQL `insert` statement has been extended to take one or more tuples of values as illustrated by:

```sql
insert into supplier ( Number, Name, city, telephone )
values ( (:INT(_short_):)393, "General Mills", 'Trenton', '609-555-8888' ),
( (:INT(_short_):)391, 'Wilbur Mills', 'Fair Oaks', '609-888-4321'),
( (:INT(_short_):)560, "Harlan Mills", 'Lawn Fair', '324-444-1234' )
```

Daytona treats each use of an SQL `delete`, `insert`, and `update` as a transaction, unless it appears within a `begin ; ... end` transaction unit, in which case, the entire unit is taken to be a...
single transaction. Consequently, the only way to get a `select` to behave as a transaction (i.e., to get Share locks) is to put it in a `begin ; ... end` transaction unit.

Here is a hybrid Cymbal-DSQL multi-statement transaction that is discussed in more detail in Chapter 17 (wtadjust.2.IQU): (The `work` keyword used below is optional.)

```sql
locals:
    FLT: .wt_cutoff = 5.0

set [ .wt_cutoff ] = read( from _cmd_line_ );

$[ begin work;

update `ORDER`
    set Quantity = (:INT(_short_);)( .5 * Quantity )
    where Part_Nbr in
        ( select Number from PART_ where Weight > .wt_cutoff );

select count( * )
from PART_
where Weight > .wt_cutoff ;

update PART_
    set Weight = 1.5 * Weight
    where Weight > .wt_cutoff ;

end work ]$
```

The next example shows how SQL updating commands can be used in the middle of Cymbal transaction tasks (sqltrans.1.IQU):
do Remove_Parted_With_Color_Match( "blue"RE );

global_defs:
define PROC transaction task:
  Remove_Parted_With_Color_Match( RE .color_pat )
  {
    for_each_time .parted_nbr
      is_such_that(
        there_is_a PARTED where(
          Number = .parted_nbr
          and Info_Source = "Zachary Scott"
          and Color Matches .color_pat
        )
      )
    )
    delete from `ORDER` where Part_Nbr = .parted_nbr
  }
  delete from PARTED where Color Matches .color_pat
  and Info_Source = "Zachary Scott"
}

See also the part_loop.IQU interactive updating example given in the "Getting Started With Daytona" tutorial.

### 4.9 Transaction Statistics

Most SQL implementations have the capability of reporting the number of records deleted, inserted, and/or updated that are associated with an SQL transaction, including the ones using `begin; ... end`. There is nothing in the SQL syntax which specifies or asks for this; it is typically information that is volunteered out-of-band by the SQL DBMS.

In Daytona, this information is available for the asking by accessing appropriate global variables as illustrated by:

```sql
delete from SUPPLIER;
do Write_Words( .txn_status.Cur_T_Deletes, "records deleted" );
```

Here is what is available:

```sql
C_external STRUCT{ UINT .Serial_Nbr,
                  UINT .Cur_T_Deletes,
                  UINT .Cur_T_Inserts,
                  UINT .Cur_T_Updates } .txn_status
```

See Chapter 17 for more details on this general mechanism which applies to all transactions, whether SQL-based or not.
5. Cymbal Classes, Variables, Functions, Assertions And Descriptions

Both declarative and procedural Cymbal make use of the same concepts of classes, variables, functions, predicates, descriptions, and assertions. This chapter introduces the user to these fundamental quantities. It doesn’t make for the most exciting reading but it is essential for being able to truly understand what follows. The Cymbal Quick chapter and the "Getting Started With Daytona Step-By-Step" tutorial (a separate document) should provide the necessary motivation for reading all of this one.

Where appropriate, subsequent examples of Cymbal usage will be augmented by the relevant grammar productions taken from the Cymbal grammar appendix; please read the beginning of this appendix for a definition of the grammar symbols employed in this chapter.

5.1 Cymbal Comments

Cymbal and DSQL queries can be commented by using the PL/I and C comment convention whereby a comment begins with a /* and ends with a */. In addition, the C++ // comment-from-here-to-end-of-line is supported: the comment begins with // and ends with the next new-line. Comments may not be nested.

5.2 Cymbal Classes

5.2.1 The Class Concept

In Daytona, a class is a set of objects satisfying some membership criterion that specifies precisely what is required for an object to be a member of the class at any given time. For example, the SUPPLIER record class could be defined, for any given time, to be the set of records with Number, Name, City, and Telephone attribute values corresponding to the hardware suppliers known to exist in the country at that time. The intension of a class is this time-dependent membership criterion. For record classes, it will certainly include a description of the attributes used by the records; some authors consider this latter schema information to be the intension of a record class: for Cymbal, that is just part of it. The extension of a class at any given time is a listing of any and all members at that time. Database updates cause the extension to change. It is possible for class membership criterion to be extensional in nature if it consists of a time-invariant explicit listing of any and all of the objects in the class; it is intensional if it is a natural language or first-order logic assertion that characterizes, for any given time, precisely those objects that are members of the class. For Daytona, views are classes that are defined intensionally by means of declarative Cymbal that refers to information in record classes.

There are times when the meaning of "class" is augmented to include the set of functions, predicates, and procedures that make use of the elements of the class set; this would constitute an object-oriented view of classes. A type consists of a (sometimes partial) description of how to construct elements of one class from elements of another class. For example, consider TUPLE[ ( 5 ) INT ]. As explained in detail later, this is the type consisting of all tuples of integers of length 5.
5.2.2 Common Primitive Classes

Cymbal supports a number of basic or primitive classes from which other classes can be constructed. The most commonly used basic classes are INTEGER, UINT, FLOAT, STRING, RE, CRE, MONEY, BOOLEAN, DATE, CLOCK, DATE CLOCK, and TIME. Cymbal also recognizes shorter synonyms for some of these basic classes, namely, INT, FLT, STR, BOOL, and DC. In fact, it is true in general that any Cymbal class with a lengthy name has a shorter synonym.

The class of all objects is the class of OBJECTs, which of course includes all the basic classes as well as those classes of OBJECTs that are constructed in some way from the members of basic classes. (To be in the spirit of things, we are following the Cymbal convention of completely capitalizing object class names.)

INTEGERs are represented as usual by possibly signed strings of decimal digits. Hexadecimal notation can also be used as exemplified by "0xFF" and "0x1a1". In order to prevent confusion, no INTEGER can start with a '0' and not be in hexadecimal; the C compiler would interpret such a number in Daytona-generated C code to be octal whereas some users may have wanted it to be read in decimal notation. However, there are functions like int_for_octal_str, int_for_binary_str, and int_for_hex_str (and their inverses) that facilitate some manner of working with INTs written in these notations.

As a convenience, Cymbal supports hatted Cymbal INTEGER constants that may contain commas as thousands-separators; here are some examples: "12,345"INT and "123,456,789,000"INT(_huge_) . Note that the full syntax with the hats and the INT type specifier are required in order to use the commas.

The INTEGER class is one of several Cymbal classes that have subclasses. There are five subclasses for INTEGER as specified by _huge_, _long_, _short_, _tiny_, _int_. To fully and explicitly express a subclass in Cymbal, just enclose the appropriate specifier in parentheses and append it to the base class name (with no intervening space) as in INTEGER(_short_). The INTs from _huge_ to _tiny_ are implemented as 64-bit, 32-bit, 16-bit, and 8-bit machine integers, respectively. The _int_ subclass specifier is only infrequently used: it causes corresponding C variables to have type 'int', as opposed to 'long int' or 'short int'. The ability to have simple 'ints' can be useful when interfacing with C code. INTEGER(_tiny_) are stored in one byte.

Each class with subclasses is defined to Daytona with an default subclass specifier. For INTEGERs, the default subclass specifier is _long_, which means that if a variable is said in Cymbal to have a type of INT, then Daytona will take that to be an abbreviation for INT(_long_). The default subclass specifiers used by Daytona are specified in the class definitions included in the system file sys.env.cy.

Subclass specifiers will sometimes be concisely referred to as "sizes" but that is just an historical accident. For the types used in the first releases of Daytona, all the subclass specifiers did refer in some way to the amount of information that could be stored in values of the type, hence the use of the term "size". However, there are now other ways for subclasses to differentiate themselves, not just by size.

The UINT family consists of the unsigned integer correspondents to INTEGERs with subclasses _huge_, _long_, _short_, and _tiny_.

Bitwise operations on INT and UINT including shifts, bitwise and and or, and exclusive or are described in Chapter 7.
Floating point representations either employ digits and decimal points in the usual way or use scientific notation. Hence, 23.4, -1E23, and 1.2e24 are all FLOATs whereas 456 is an INTEGER. FLOATs must contain either a ‘.’ or an ‘e’ or an ‘E’; sequences of digits that don’t are INTEGERS. The subclass specifiers for FLOATs are _long_ and _short_ (corresponding to C doubles and singles, respectively), with the default being _long_.

In languages like C, floating point round-off error may cause two floating point calculations that the user considers to be equivalent to be unequal as far as the equality predicate is concerned. Daytona provides some freedom from this by considering two floating point numbers \(x\) and \(y\) to be the equal if, for \(\varepsilon\) equal to 8 times the machine’s MINFLOAT value, either both \(x\) and \(y\) are within \(\varepsilon\) of 0.0 or if \(|x/y - 1.0| < \varepsilon\).

FLTs also present indexing issues for B-trees, associative arrays, and BOXes. Clearly, if an entry is indexed by 8.33333333333333 and floating computation round-off error produces 8.3333333333334 as a key, then the stored entry will not be found. Furthermore, the user may be thinking in terms of 8 1/3 but this number is represented by 8.3333 as a FLT(_short_) and 8.33333333333333 as a FLT(_long_) and so suppose one mistakenly types in the wrong number of 3s for a key in an indexed search. For example, an entry indexed by 8.33333333333333 has been created but is then searched for by using 8.33333333333333: regrettably, it will not be found. So, Daytona will warn about using FLT in index situations and will recommend that the the FLT used as indices always be rounded to some established precision by using \(\text{round\_to\_nearest()}\) (AKA rtn). This strategy is illustrated in part._loop.IQU and dynara.b.Q. The same considerations hold for MONEY since it is implemented as a FLT.

Division by 0 is undefined in mathematics. Since Daytona FLT arithmetic is implemented by the OS in IEEE floating point, dividing by 0.0 can result in a value that when printed is seen to be inf.0. This is not Daytona’s choice, this is what IEEE floating point does. If the user would like an error condition raised leading to program termination, then that can be done by running Tracy with the -ZDC (i.e., zero divide checks) option, which can be put into the shell variable DS_TFLAGS globally or just locally as in:

```
DS_TFLAGS=-ZDC DS Compile <my_query_fl>
```

In contrast to INTs, where the type of an INT constant is the smallest subclass of INT that will hold it, FLOATs are different in that all FLT constants are assumed to be FLT(_long_). If the user wants to work with FLT(_short_), then the user must be consistently explicit about it, both with the constants (as in ^1.3444^FLT(_short_)) and FUN/VBL definitions/declarations, except that Daytona will allow any FLT constant that is of precision less than 7 and is not explicitly specified by the user to be FLT(_long_) to be assigned to a FLT(_short_) VBL or in fact, to go into any FLT argument slot.

As a convenience, Cymbal supports hatted Cymbal FLOAT constants that may contain commas as thousands-separators; here are some examples: ^12,345.87^FLT and ^123,456,789,000.9^FLT. Note that the full syntax with the hats and the FLT type specifier are required in order to use the commas. Commas as thousands-separators may be used in data stored in RECORD_CLASSES by using Input/Output Filter_Funs to write the FLOATs (or INTs) out using \(\text{str\_for\_dec()}\) with its _1000s_ option and then reading them back in from the DC data into a FLOAT (or INT) by using first getting rid of any commas (by the likes of .x">", " or by using the faster \(\text{wipe\_out\_these\_chars()}\)) and then converting that STR to FLT (or INT). Use the PRED Is_A_Graphic_Str to determine if a STRING has the

Copyright © 2013 AT&T
All Rights Reserved.
September 15, 2013
syntax of a decimal (fixed point) number like 12,345.87 -- whether it is using commas as thousands-separators or not.

MONEY constants in Cymbal look like ’3923.45’MONEY. MONEY values are represented in programs as C doubles, as are FLOAT(_long_) values. Consequently, all arithmetic on them is done using floating point arithmetic. However, when written out using Write or stored in a data table using do Change, they are rounded to two decimal places. In this way, little if any precision is lost during internal calculations and yet, the results are printed and stored automatically in a way that looks like money. Since C doubles have a precision of at least 15 places, the MONEY type can accumulate much larger sums than, say, INT(_long_). The stdev, var, and corr aggregate functions do not yet work on MONEY.

BOOLEAN is the class of boolean values of which _true_ and _false_ are the only members. BOOL is a synonym.

STRINGs are sequences of characters enclosed in double quotes. So, "Tom" and "Helena's last chance: " are both STRINGs. STR is a synonym for STRING. New-lines are acceptable within STRINGs as are the C backslash escape sequences: the octal escapes like \145, and \n, \t, \f, \r, \v, and \\. Also, " in a STRING constant is considered to be a double-quote. Conversely, \ preceding any other character is considered as \ preceding that character -- as opposed to that character alone. Also, a backslash followed by a new-line character is considered to be a hidden new-line and both characters are treated like they weren't there.

As far as ds_m4 is concerned, STRING constants are treated as comments which means that no macro substitution can occur on their contents. There are good reasons for this but if the user would like macro substitutions in string-like constants, then they can use LITERALs instead (which are delimited by a back-quote and a quote).

Subclasses of STRINGs are indicated either by *, =, or by a non-negative integer. If a non-negative integer is used, then the subclass consists of all strings whose lengths are equal to or less than that integer. The (default) * subclass is the class of all strings, regardless of length. Section 6.9 describes the use of the = subclass in working with strings imported from C.

In terms of implementation, C character arrays are used to implement STRINGs with a non-negative integer subclass, as in STR(22). The type char * is used to implement STR(=), which is unlikely to be used in pure Cymbal programs. Use the special constant _null_str_ to set a STR(=) VBL to the null string pointer, i.e., a C NULL.

STR(*) objects are implemented with a special C structure called a Byte_Seq_Box which is automatically expanded as needed so that it can hold strings that are indefinitely long. A Byte_Seq_Box contains a pointer to the corresponding (malloc’d) C character array, the length of that array (which is called maxlen), and the amount of that array which is currently being used to store the STR value, that amount being called len. There are a lot of types in addition to STR(*) that are implemented using Byte_Seq_Box; they can be found by examining the with_c_type value in the CLASS definitions in sys.env.cy. The function allocmaxlen maps instances of such types to their maxlen: the unusual name is used instead of maxlen in order not to conflict with any user’s choice of that name. The reason why it is useful to go into the implementation of STR(*) objects is because they can hold byte sequences that are not C strings. Recall that since every C string is terminated by a null byte, it is impossible for a C string to contain any null bytes (because the first one that appeared would terminate
Certainly, STR()s typically represent C strings. However, since STR() are implemented by using Byte_Seq_Boxes and since a Byte_Seq_Box not only terminates its entry with a null byte but also keeps track of the length of its entry separately, an STR() can be used to store arbitrary ASCII or 8-bit byte sequences, possibly containing many null bytes. There are only rare occasions for wanting to use STR(*) in this way; the only documented one is when reading/writing using stated_sizes. In order to support the general case, some STR(*) operations are implemented differently than corresponding operations for the other STR subclasses. In particular, the relational operators =, >, >=, <, <= are done using Byte_Seq_Box-aware code which gives no special treatment to "interior" null bytes. Also, assigning STR constants (like "abcd") to STR(*) VBLs is done in such a way that any "interior" null bytes are copied over like any others. Note that the behavior of these operations is exactly the same as the corresponding traditional C string operations when the arguments are C strings. Lastly, if it is ever desired to set to some INT value the len attribute of the Byte_Seq_Box behind an STR(*) VBL, then the Set_Len_To PROC declared in sys.env.cy will do it: any extensions to the current value are accomplished with however many null bytes are needed.

For the purposes of influencing type-inference, STR constants can explicitly indicate their intended type simply by suffixing their type to the constant as in "abcede"STR(10).

An RE is a regular expression and a CRE is a compiled regular expression. An RE constant is written in Cymbal enclosed in double quotes with the RE class name immediately following. So, ".*$"RE is one of the standard UNIX regular expressions, The full definition of REs is given in Chapter 7. CREs are not printable and therefore the user cannot work with them as constants. They can of course be the values of variables and functions. CREs are not often needed these days since the system will now pre-compile RE constants behind the scenes whenever possible. So, CREs are deprecated and anachronistic; their use should be replaced with RE use.

However, there is one runtime use of CRE which can be valuable. Any CRE VBL that has received a value (possibly through compile_re() ) can provide the pattern used to compile it and more importantly, the number of RE \(\backslash()\) subexpressions it contains. This is done via using the STRUCT member selectors Pattern and Nbr_Subexpr. Here is an example use: .mycre.Nbr_Subexpr.

Not infrequently, the user desires to embed the values of variables in the midst of STRINGs. This can be done by a simple Write call, a Write call that uses a with_tuple_format keyword argument, or by using the concat FUNCTION. Perhaps the most pleasant way though to accomplish this end is by using the pseudo-class ISTR (as in ‘interpolated string’) as in "hello, .x; .y sends greetings."ISTR. A use of ISTR is very much like using a macro. Indeed, very early on in the Cymbal translation process, ISTR terms are replaced with concat calls. For example, the ISTR expression above is replaced by:

```cymbal
concat( [ "hello, ", .x, "; ", .y, ", sends greetings." ] )
```

As long as the VBLs x and y can be coerced to WRITABLES, this concat call will produce the intended STRING. In general, the concat function takes an optional STR second argument that is used to separate the elements of TUPLE first argument in the return value of the function.

```cymbal
STR FUN( manifest TUPLE[ ( 1-> ) OK_OBJ ], STR .sep = "" ) concat
```

Do not be seduced into thinking that ISTR is a CLASS: it can only appear in Cymbal programs as
part of one of these constants, where it tells the system how to understand the preceding STRING. In particular, ISTR cannot be the type of VBL or a FUN. Or to put it differently, an ISTR is just a STRING that is written in a shorthand that instructs the system to interpolate the WRITABLE form of VARIABLE values at the indicated places in the STRING.

In addition to the usual escape sequences that STRINGs can contain, ISTR supports the escaping by a backslash of what would otherwise be interpreted as a VBL value. For example:

```plaintext
do Write_Line( ".x = .x" );
```

Furthermore, a variable dereference that ends with \& is taken to be the same as the variable dereference itself. This supports expressions like:

```plaintext
do Write_Line( ".x plural = .x\&s" );
```

### 5.2.2.1 Time-related Primitive Classes: DATE, CLOCK, DATE_CLOCK, TIME

Cymbal supports a rich collection of time-related types. DATE, CLOCK, and DATE_CLOCK are the basic types that are used to identify points in the time dimension. TIME is used to express durations of time as defined by the amount of time between two points in the time dimension.

There are seven different subclasses of DATES that correspond to different ways of writing a DATE. The following table defines them.
<table>
<thead>
<tr>
<th>Subclass Specifier</th>
<th>Example</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>yyyymmdd</em></td>
<td>1998-03-02</td>
<td>ISO 8601</td>
</tr>
<tr>
<td><em>yyyymmdd</em></td>
<td>1998-3-2</td>
<td>leading 0 not required</td>
</tr>
<tr>
<td><em>yyyyymmd</em></td>
<td>19980302</td>
<td>using the empty delimiter</td>
</tr>
<tr>
<td><em>yyyyymmd</em></td>
<td>19980302</td>
<td>leading 0s required</td>
</tr>
<tr>
<td><em>mmddyyyy</em></td>
<td>3-2-1998</td>
<td>- is the default delimiter</td>
</tr>
<tr>
<td><em>mmddyyyy</em></td>
<td>03/02/1998</td>
<td>using / as delimiter</td>
</tr>
<tr>
<td><em>mmddyyyy</em></td>
<td>3.2.1998</td>
<td>using . as delimiter</td>
</tr>
<tr>
<td><em>mmddyy</em></td>
<td>3-2-98</td>
<td></td>
</tr>
<tr>
<td><em>mmddyy</em></td>
<td>03-02-98</td>
<td></td>
</tr>
<tr>
<td><em>mmddyy</em></td>
<td>03-02-00</td>
<td>for the year 2000</td>
</tr>
<tr>
<td><em>mmddyy</em></td>
<td>3-2-10</td>
<td>for the year 2010</td>
</tr>
<tr>
<td><em>mmddyy</em></td>
<td>3/2/98</td>
<td></td>
</tr>
<tr>
<td><em>mmddyy</em></td>
<td>3.2.98</td>
<td></td>
</tr>
<tr>
<td><em>ddMMMyyyy</em></td>
<td>22 Jun 1998</td>
<td>AT&amp;T presentation format</td>
</tr>
<tr>
<td><em>ddMMMyyyy</em></td>
<td>02 Jun 1998</td>
<td>MUST have the leading 0</td>
</tr>
<tr>
<td><em>Mddyyyy</em></td>
<td>September 14, 1998</td>
<td></td>
</tr>
<tr>
<td><em>Mddyyyy</em></td>
<td>May 4, 1998</td>
<td></td>
</tr>
<tr>
<td><em>MMMddyyyy</em></td>
<td>Sep 04 1998</td>
<td>UNIX style We allow: Mar 3 1998</td>
</tr>
<tr>
<td><em>MMMddyyyy</em></td>
<td>Mar 2 1998</td>
<td>no leading 0 required on input</td>
</tr>
<tr>
<td><em>DDDMMMddyyyy</em></td>
<td>Fri Sep 04 1998</td>
<td></td>
</tr>
<tr>
<td><em>DDDMMMddyyyy</em></td>
<td>Fri Sep 4 1998</td>
<td>no leading 0 required on input</td>
</tr>
</tbody>
</table>

Note that the subclass specifiers consist of patterns indicating the presence of month, day, and year values as well as their length. Capital letters appearing in these subclass specifiers denote that appropriate letters are used in the corresponding positions in the actual DATES. The subclass specifier used by default (if none is explicitly given) is _yyyymmdd_.

The subclass specifier refers only to the format used to print the associated DATE by procedures such as Write. Otherwise, any variable of any DATE subclass type may be assigned to any other such variable or have its value offered as an argument to a parameter variable of any DATE subclass type. This insensitivity to format also holds for reading in DATES that are in DC tables or are in the rcds.

DATE constants may be written in three possible ways. They can appear within hats as with "1998-03-21" or with a DATE suffix as with "22 Jun 1998 DATE" or with a complete DATE subclass suffix as with "Fri Sep 04 1998 DATE(_DDDMMMddyyyy_)". In the first case, the Cymbal parser determines the exact type of the constant. In the second case, the Cymbal parser also determines the exact subclass for the constant on the basis of the constant alone, in this case ignoring the default subclass specifier. (The default subclass specifier is ignored in this case on the theory that it would be too strange for the user to Write a constant like "22 Jun 1998 DATE" and have it come out looking like 1998-06-22.) In the third case, since the user has gone out of their way to specify the exact type of the constant, the Cymbal parser will determine the type of the constant exactly and generate an error.

Copyright © 2013 AT&T
All Rights Reserved.
September 15, 2013
if it does not match the explicitly given type exactly.

The two-digit years that are used in DATE(_mmddyy_) are just abbreviations for four digit years: if the two-digit year yy is between 50 and 99, then the corresponding four digit year is yy + 1900; else, it is yy + 2000. Of course, if a DATE(_mmddyy_) has a year that is not in this 100 year range, then it will be printed using 4 digits.

For the _mmddyy_ and _mmddyyyy_ DATES, since they can use any of three separators (i.e., [./-]), the question arises as to how to specify which of those separators to use when printing, since the separator is clearly not specified in the type. The solution is subtle but effective. Whenever a _mmddyy_ or _mmddyyyy_ DATE variable is assigned another DATE(_mmddyy_) or DATE(_mmddyyyy_) object, the separator character associated with that variable is changed as needed to be the same as that of the assigned value. So, for example, with this program:

```plaintext
local: DATE(_mmddyy_) .dd = '12/05/04'
set .dd = '2007-06-11'
do Write_Line(.dd);
```

the output is: 6/11/07. (Recall that the default separator is -.)

DATES are stored in memory as 11 byte C structs. Many examples of DATE usage can be found in dates.*Q. Functions that manipulate DATES are described in Chapter 7.

Just as DATES identify points along the calendar dimension, CLOCKs identify time points down to the nanosecond (if needed) along the time dimension associated with a day. CLOCK constants indicate clock values in either 12 or 24 hour time formats. Here are some examples: ‘1am’CLOCK, ‘1:30pm’CLOCK, ‘9:59:12PM’CLOCK, ‘22:10:10.10’CLOCK. Note that if the am/pm designator is left off, then 24-hour CLOCKs are assumed. Also, hours may be supplemented with minutes, seconds, and fractions of a second in turn, going from left-to-right. CLOCK constants satisfy the regular expression:

```
[0-9][0-9]?\:[[0-9][0-9]?\:]?\:([0-9][0-9]?\:([0-9]([0-9]?)?)\)?\((am|AM|pm|PM)\)?
```

These definitions are reflected in the 8 CLOCK subclasses that govern the printing of a CLOCK:
<table>
<thead>
<tr>
<th>Subclass Specifier</th>
<th>Example</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>24_hhmssf</em></td>
<td>17:33:02.561</td>
<td>the f is for fractions of a second</td>
</tr>
<tr>
<td><em>24_hhmssf</em></td>
<td>17:33:03.</td>
<td></td>
</tr>
<tr>
<td><em>24_hhmss</em></td>
<td>17:33:03.</td>
<td>on output, the CLOCK is rounded to seconds</td>
</tr>
<tr>
<td><em>24_hhm</em></td>
<td>17:33</td>
<td>on output, the CLOCK is rounded to minutes</td>
</tr>
<tr>
<td><em>24_hh</em></td>
<td>18</td>
<td>on output, the CLOCK is rounded to hours</td>
</tr>
<tr>
<td><em>12_hhmssf</em></td>
<td>5:33:02.561pm</td>
<td>the f is for fractions of a second</td>
</tr>
<tr>
<td><em>12_hhmssf</em></td>
<td>5:33:03.pm</td>
<td></td>
</tr>
<tr>
<td><em>12_hhmss</em></td>
<td>5:33:03pm</td>
<td>on output, the CLOCK is rounded to seconds</td>
</tr>
<tr>
<td><em>12_hhm</em></td>
<td>5:33pm</td>
<td>on output, the CLOCK is rounded to minutes</td>
</tr>
<tr>
<td><em>12_hh</em></td>
<td>6pm</td>
<td>on output, the CLOCK is rounded to hours</td>
</tr>
</tbody>
</table>

The subclass specifier refers only to the format used to print the associated CLOCK by procedures such as `Write`. Otherwise, any variable of any CLOCK subclass type may be assigned to any other such variable or have its value offered as an argument to a parameter variable of any CLOCK subclass type. This insensitivity to format also holds for reading in CLOCKS that are in DC tables or are in the rcds. Also another implication is that a VARIABLE or FIELD of type CLOCK with subclass `_24_hh_` can have a value which in fact includes minutes, seconds and fractions of a second which will simply not be visible when printed -- unless that value is cast to a type with a format that would support printing that information like `_12_hhmssf_`.

At the C-program level, a CLOCK is stored in memory as two long ints, one for seconds after 12am and the other for nanoseconds. The binary and aggregate `min` and `max` functions can be applied to CLOCKS. The usual equality and inequality predicates work with CLOCKS. Caveat: the colons present in CLOCKS preclude using colons as delimiters in data files that store CLOCKS! Other Functions that manipulate CLOCKS are described in Chapter 7.

A DATE_CLOCK is used to specify a point in the time dimension down to the nanosecond. As the name indicates, a DATE_CLOCK is the combination of a calendar DATE with a CLOCK indicating time of day. For example, `1998-06-22@14:02:34.5558`DATE_CLOCK is a Cymbal constant indicating a DATE_CLOCK. DC is an abbreviation for the DATE_CLOCK name. For readability, the @ can be surrounded by blanks as in `1998-06-22 14:02:34.5558`DC. Even more conveniently, Daytona will automatically classify as DATE_CLOCKs any constants of the appropriate format that appear within hats. Consequently, there is no need to attach a class specifier to the constant, as in `1998-06-22 @ 14:02:34.5558`. As an even further convenience, Daytona will recognize SQL-3 standard timestamps which look like a DATE_CLOCK only that one (or more) blanks appear where the @-sign would be expected (by Daytona), as is the case with `1998-06-22 14:02:34.5558`.

DATE CLOCKS have 56 subtypes corresponding to the 7 different DATE subtypes matched against 8 different CLOCK subtypes. The DC subtypes correspond to the natural concatenation of the DATE and CLOCK subtypes as indicated by `_yyyymmdd_24_hhmssf_`. 

---

Copyright © 2013 AT&T
All Rights Reserved.
September 15, 2013
The subclass specifier refers only to the format used to print the associated DATE_CLOCK by procedures such as \texttt{write}. Otherwise, any variable of any DATE_CLOCK subclass type may be assigned to any other such variable or have its value offered as an argument to a parameter variable of any DATE_CLOCK subclass type. This insensitivity to format also holds for reading in DATE_CLOCKs that are in DC tables or are in the rcds. Also another implication is that a VARIABLE or FIELD of type DATE_CLOCK with subclass \texttt{yyyymmdd\_24 hh} can have a value which in fact includes minutes, seconds and fractions of a second which will simply not be visible when printed -- unless that value is cast to a type with a format that would support printing that information like \texttt{yyyymmdd\_12 hhmmssf}.

A TIME is an amount of time expressed in days, hours, minutes, seconds, and fractions of a second. Sample Cymbal TIME constants are: \texttt{"0s"TIME, "2m22s"TIME, "3h0m15.4567s"TIME, "3d"TIME.} TIME constants satisfy the regular expression:

\[
[(0-9]+d)?([(0-9]+h)?([(0-9]+m)?([(0-9]+s)?\((0-9)\)?s)\]
\]

There are currently two TIME subclasses, \texttt{dhmsf} and \texttt{s}, which are used to specify how TIME values are Written. The default is \texttt{TIME(dhmsf)} which will Write out as days, hours, minutes, seconds, and fractional seconds; for convenience, \texttt{TIME(s)} will round to the nearest second and Write out as the equivalent number of seconds. At the C-program level, a TIME is implemented as two long ints, where one long int stores seconds and the other stores nanoseconds. The usual arithmetic operators have been overloaded so that TIMES can be added, subtracted, multiplied by a FLT, divided by a FLT or a TIME, and mod-ed by a TIME. The binary and aggregate \texttt{min} and \texttt{max} functions can be applied to TIMES; the aggregate functions \texttt{sum} and \texttt{avg} also work with TIMES. The usual equality and inequality predicates work with TIME.

\textbf{5.2.2.2 THINGs And Other Types}

A Cymbal THING is an OBJECT which is identified to the system by a character string name enclosed in hats. So, \texttt{"Henry"} is a THING as is \texttt{"for\_each\_time"}. The idea behind THINGs is that the enclosing hats enable the user to write (between the hats) syntax which would otherwise confuse the Cymbal parser; in this way, the special meaning of Cymbal keywords and other syntax can be turned off and therefore, for example, Cymbal keywords can be worked with and talked about using Cymbal as if they were not part of the language. In fact, it is for the same purpose of syntactic-escape that THINGs are employed in DSQL and the rcd syntax as well.

A DATE, CLOCK, TIME, or MONEY is a THING, which is to say that each is a subclass of THINGs. THINGs are most certainly are \textit{not} STRINGs: it is illegal to do STRING operations (like substring and substitution) on THINGs. Consequently, \texttt{"3-1-49"} is a DATE whereas \texttt{"3-1-49"} is a STR. Notice that the THING syntax enables Daytona to introduce special syntax for the members of an arbitrary number of new THING subclasses without fear of having the parser accidentally misunderstand somebody’s STR constant as one of those THINGs. The backslash escape mechanism employed for STRINGs is \textit{not} used for THINGs. Instead, backslash has precisely two special meanings: it is used to escape the end-of-THING meaning for \texttt{"} so that \texttt{"} can be used as part of a THING and it is used to escape itself so that \texttt{"\\"} is the THING consisting of one backslash; otherwise, backslashes are passed through unchanged with no further implications.

\begin{center}
Copyright © 2013 AT&T
All Rights Reserved.
September 15, 2013
\end{center}
In general, as illustrated by `^2d3h4m22s`\TIME and `^May 5, 1998`\DATE(_Mddyyyy_), by suffixing a THING constant with a type, the user can specify constants of specific subclasses of THINGS. This holds true even for INTs as illustrated by `^5004`\INT(_short_) . DATES and DATE_CLOCKs are the only THINGS that do not need a type suffix for the system to automatically classify them to their smallest containing type. This means, for example, that `^2-17-97` is automatically recognized as a DATE(_mmddyy_) instead of just as a THING.

For historical reasons, Cymbal also supports LITERALS, which are represented by character strings enclosed in matching directed single quotes. `^2+4/6` is a LITERAL. The same backslash escape mechanism employed for STRINGs is available for LITERALS. LITERALS can be used wherever STRINGs can; currently, there is nothing in Cymbal that can be done with LITERALS that can’t be done with STRINGs and so the user may as well work exclusively with STRINGs. (However, any trouble $ds_m4$ macros have with double quotes can probably be avoided by using LITERAL constants instead of STRING constants. The reason is because double quotes are considered to be comments by $ds_m4$ whereas not so for directed single quotes. This implies that $ds_m4$ will require double quotes to appear in pairs -- because it will be looking for the matching double quote to the one that starts a comment. Consequently, $ds_m4$ will do substitutions inside of LITERAL constants but not inside of STRING constants.)

There are two drawbacks to storing large strings in a FILE that is not vertically partitioned, assuming that Daytona’s maximum record length bound of 65534 bytes is being honored. The first is that if the FIELD is not the last one, then it is expensive in time for Daytona to traverse its value looking for the next FIELD value. The second is that such a string is most likely text of some sort which could contain many characters that Daytona is using for delimiters.

The first and second drawbacks are addressed by storing the large unruly sequences of bytes outside of Daytona in UNIX files or archive members and then storing their UNIX file path names as the values of FIELDs, suitably typed, of course. In this regard, a TEXT is (a UNIX filepath to) a text object residing as the contents of a file or archive member and a BLOB is (a UNIX filepath to) a binary (or similarly amorphous) file/archive-member and both are THINGS. `^$HOME/d/lib.a@Monday`\TEXT is the text member named “Monday” in the archive $HOME/d/lib.a, and similarly for `^~john/d/libin@speech.1`\BLOB . Special cases of BLOBS include VIDEO, AUDIO, and PHOTO. These all have various subtypes described in $DS_DIR/sys.env.cy under the class definitions section.

5.2.2.3 Storing The Unstorable: SAFE\_STR, ISO8859, and BASE64

The SAFE\_STR (or SAFE\_STRING) type handles the second drawback by automatically storing ASCII data on disk so that the following character mappings hold:
For a SAFE_STR FIELD, Daytona will automatically substitute the delimiters for the control characters when the FIELD value is read in from disk. Thus, these standard delimiters will appear in memory as desired but will not be present on disk to confuse Daytona. The obvious implication is that any data file using a SAFE_STR FIELD must use delimiters from the group specified above. The case of handling ^A and ^` has a different justification than the policy for handling the printable delimiters. The goal here is to enable SAFE_STR to live up to its name in the sense of being safe: it is simply forbidden (i.e., unsupported) to have ^A in DC data files and since ^` has special meaning to the system as the record-delete indicator, that meaning obviously must also be turned off when not intended. And obviously again, since the control characters used in the SAFE_STR encoding have special meaning to Daytona, the user should not use those control characters in plaintext SAFE_STR FIELD values (and expect them to be left as they are).

SAFE_STR can be used wherever STRING can. This means that Daytona will automatically cast SAFE_STR in situations that require a STR; the converse however is not true in which case an explicit casting of a STR to SAFE_STR is required. There are no length restrictions on SAFE_STR. SAFE_STR can be cast to be a STR and vice-versa: this operation actually does nothing to the object being casted because in memory, SAFE_STR objects are always in their unencoded form; they only become encoded (automatically) when written out to a DC data file. Since SAFE_STR can appear as a STR argument without explicit casting, any necessity of casting is pretty-much confined to casting a STR value to be SAFE_STR when arranging for a value to be written to a DC data file as in:

```
\n do Change so_that( there_is_a WIDGET where( Safety = (SAFE_STR)("abc"+.str1) ) );
```

This SAFE_STR cast does not actually encode the string at all; in fact, it leaves it untouched because this all taking place in memory where SAFE_STR are not encoded. The utility of this cast is to mollify the type-checking system. On the other hand, when it comes to considering what SAFE_STR look like encoded on disk, in the unlikely event that the user should wish to, they can explicitly convert strings to and from the SAFE_STR encoding by using dc_safe_str_for_str() and str_for_dc_safe_str(), resp.

Once again, SAFE_STR is good for making arbitrary ASCII data safe to store (recall that ASCII is a code for the first 7 bits of a byte, with the 8th bit set to 0). The advantage of SAFE_STR is that an encoded string is precisely the same length as its source string.
By means of the ISO8859 type, Daytona supports storing in DC data files text written using any of the ISO 8859 character sets. As described in detail in Wikipedia, ISO 8859 codes are 8-bit codes that coincide with ASCII when only the first 7 bits are used (i.e., when the 8th bit is 0) but continue on to impose meaning in terms of characters on all byte codes with the 8th bit set. There are 16 “parts” of ISO 8859, with Part 12 having been abandoned; each of these parts corresponds to a particular assignment of national language characters to those 8-bit codes with the 8th bit set. For example, Part 7 contains a full complement of Greek characters, whereas Part 1 (AKA Latin-1) and its update Part 15 contain the characters that support the most popular Western European languages such as French, German, Spanish, Italian, and Swedish. In 1992, the IANA registered the character map ISO_8859-1:1987, more commonly known by its preferred MIME name of ISO-8859-1 (note the extra hyphen over ISO 8859-1), a superset of ISO 8859-1, for use on the Internet. Note this stunning display of how not to choose a name: the subtly different name ISO-8859-1 names a character set that is a superset of ISO 8859-1. Further opportunities for confusion were added by Microsoft with its Windows-1252 character encoding of the Latin alphabet: This encoding "is a superset of ISO 8859-1, but differs from the IANA’s ISO-8859-1 by using displayable characters rather than control characters in the 0x80 to 0x9F range". "UTF-8 is recommended instead of windows-1252 for modern applications." More on UTF-8 in a moment.

Anyway, Daytona’s type ISO8859 has subclasses that correspond to each of the above variants of ISO 8859. They go from _1_ to _16_, skipping the abandoned Part 12 and then continue by including _1i_ for ISO-8859-1 and its friend _1_win1252_. Daytona does not treat data typed by any of these subclasses (as in ISO8859(_15_)) in any way different from that typed by any of the other subclasses. The point of having these subclasses is to enable the Daytona user to record the nature of the encoding of their data so that they can easily determine what glyphs will be used when UNIX or whatever displays their data. To this end, to convert from a value in one of these subclasses to another, an explicit cast is needed, even though at runtime, the mapping is the identity mapping, a no-op.

When an rcd specifies that a FIELD is of some ISO8859 type, then Daytona will use a space-efficient encoding consisting of either one or two bytes per ISO8859 character. Conveniently, with the exception of the 9 DC-sensitive delimiter characters, # % : [ \ ] { | }, the remaining 86 printable ASCII characters are encoded as themselves. Those 9 DC-sensitive delimiter characters are encoded as (single-byte) ASCII control characters starting at ^N and ending at ^V. All remaining characters out of the 256 are encoded using 2 bytes. Daytona will automatically convert ISO8859 FIELD values to their 8 bit unencoded forms upon reading them into memory. Once in memory, the 8 bit characters are what they are and when written out using Write, are written out unchanged. ISO8859 can be used wherever STRING can. So, the utility of this type really centers around being able to store such text in DC data files (and in such a way that they can be record-level compressed as desired). In the unlikely event that the user should wish to, they can explicitly convert strings to and from the ISO8859 encoding by using dc_iso8859_for_str() and str_for_dc_iso8859(), resp.

In order to work with any character set known to man, the recommended encoding is UTF-8. UTF-8 is one of the encodings in the Unicode family. UTF-8 uses between 1 to 4 bytes to encode characters. Fortunately, the first 128 "code points" are precisely those of ASCII. Daytona does support UTF-8, but only indirectly by means of the general-purpose BASE64 type, which has the corresponding BASE64(_utf8_) subclass.
The BASE64 type is used to store arbitrary data in the FIELDS of DC data files. This includes any kind of binary data. The implication is that there could well be null (all-0) bytes in the unencoded BASE64 quantities. Consequently, once a BASE64 value has been read into memory and decoded into its natural, possibly binary state, the variable holding that value cannot be considered to be a STR kind of variable because in general it cannot be counted on to end with a null byte. Fortunately, the overloaded length function will return the actual length of the BASE64 object whether it contains null bytes or not. However, note that one of the major advantages of UTF-8 is that its character encodings do not contain null bytes; hence, a BASE64(utf8) read into memory from a RECORD_CLASS can be treated like a STRING.

Currently, BASE64 has six subclasses: plain, utf8, ucs2, utf16, ucs4, and utf32. (The latter 4 encodings can contain null bytes, which means that typical string routines relying on the use of a null byte to terminate a string cannot be used with them.) Just as with ISO8859, this is just a mechanism whereby the user can record the nature of the encoded byte strings. Daytona does not support casting from one of these type subclasses to another. Of course, BASE64 could be used to encode Western European languages instead of ISO8859 but it would take more bytes to do it. BASE64 can be used wherever STRING can. In the unlikely event that the user should wish to, they can explicitly convert strings to and from the BASE64 encoding by using dc_base64_for_str() and str_for_dc_base64(), resp. Nota bene: Daytona’s BASE64 encoding is not identical to anyone else’s, although it is based on the same general principles.

"Sorting of UTF-8 strings using standard byte-oriented sorting routines will produce the same results as sorting them based on Unicode code points (Wikipedia).” However, the BASE64 encoding does not preserve the sorting of the un-encoded bytes.

5.2.3 Compression-oriented Primitive Classes

The classes HEKA, HEKINT, HEKUINT, HEKSTR, HEKDATE, HEKCLOCK, HEKDATE_CLOCK, HEKTIME,.HEKMONEY, DOYDATE, ATTDATE, BITSEQ, IP, and IPORT are stored in DC tables in a special ASCII compressed format. They can be very useful for saving space in very large files. HEKA, HEKINT, HEKUINT, HEKSTR, HEKTIME, HEKCLOCK, and HEKMONEY typically cut the size of their FIELD’s values in half. They are intended for use in storing sequences of digits. HEKDATE, HEKCLOCK, HEKDATE_CLOCK, ATTDATE, and DOYDATE provide a compressed way to store dates and other time information. BITSEQ allows sequences of 0-1 bits to be stored compactly and to be manipulated using conventional operations. IP objects are compressed IP (internet protocol) addresses that sort nicely.

A HEKA value is a compressed sequence of digits, minus signs, or dots (periods). As a concession to errors that have to be tolerated in real data, a HEKA can also contain other characters with the result that that particular HEKA is not compressed at all, even if it has a large proper substring of digits: indeed, such uncompressed HEKAs are stored in DC data files as they are but prepended with an escape character. Such out-of-band but still accepted HEKAS include the empty HEKA. Note that decimal numbers like −123456.456789 can be stored efficiently as HEKA; if there aren’t too many digits, a FLT cast in Cymbal will work as in (FLT) "123.45"HEKA . A HEKA remains in its compressed state when it is read into memory and is only converted into its expanded form when it is printed out using Write(). Read() and read() will convert in the other direction from expanded (human-readable) form to
compressed form.

Since there are occasions when the user wishes to Write a HEKA out as a HEKA (i.e., to avoid the default conversion into a sequence of digits), the function `heka_as_str` can be applied to a HEKA in order to return a STR consisting of precisely the HEKA characters of the argument. Note that this is different from `heka_for_str` and `str_for_heka`, which convert back and forth between the plain text and compressed forms, although the preferred was to do that in-memory is to use a (HEKA) or (STR) cast. Conversely, there are occasions when a sequence of characters is read in from the outside and it is known to be in the HEKA code. The user may instruct Daytona to consider this sequence of characters to be a HEKA (without any transformation being performed) by feeding it as the sole argument to `str_as_heka`. Perhaps the key thing to remember about `heka_as_str` and `str_as_heka` is that they are identity functions: they do not change their arguments in any way. Instead, they just convince Daytona to consider their argument (and hence return value) to be a member of a different type.

At the Cymbal level, HEKA constants appear as the likes of `"11238119208888"HEKA` and `"-23341"HEKA` or even `"abcd4"HEKA` (which will not be compressed in DC files). HEKA have the same Subclass or Size designators that STRINGs and LITERALS do, namely, `*`, `=`, or some nonnegative integer. A nonnegative integer Subclass indicates the maximum number of digits that the HEKA can contain. HEKA FIELD values are stored in indices as HEKAS, thus continuing to save space.

The HEKA encoding uses the great majority of ASCII characters. Consequently, care must be taken so that the DC delimiters specified by the user are not among them. In particular, when using any of the HEKA-based types, all of the characters that format a DC record into fields must be taken from the following set:

```
| : { } [ ] # %
```

The HEKA encoding preserves lexicographic ordering in that one HEKA is lexicographically less than another if and only if the decoding of the first is lexicographically less than the decoding of the second. This lexicographic ordering is the same as that used by the `C strcmp` function. Note that this is not the same ordering that would be obtained by considering decoded HEKA as integers, but rather it is that that occurs when they are considered to be strings of digits. This preservation of ordering only holds for HEKAS that do not use a minus sign. Incidentally, the name HEKA is taken from the Greek `hekaton` which means 100: the essence of the HEKA encoding is to encode two digits into one byte.

A HEKSTR is stored in the HEKA format in DC files but as soon as it is read into memory, it is converted out of the HEKA format into a STRING. Once read in, a HEKSTR can be operated on by various fpps just as if it were a STRING or LITERAL. In fact, they can be freely passed to fpps that are declared to accept STRINGs or LITERALS as arguments. (The converse is not true: if an fpp expects a HEKSTR argument, it will not accept a STRING or LITERAL in its place.) At the Cymbal level, HEKSTR constants appear as the likes of `"1123811"HEKSTR` and even `"abcd4"HEKSTR`, although the latter will not be compressed in the DC file. HEKSTR have the same Subclass or Size designators that STRINGs do, namely, `*`, `=`, or some nonnegative integer. HEKSTR FIELD values are stored in indices as strings of digits.

A HEKINT is stored in the HEKA format in DC files but as soon as it is read into memory, it is converted into an INT. HEKINTs have the same subclass specifiers that INTs do, namely, `_tiny_`, `_short_`, `_long_`, `_huge_`. The default subclassifier is `_long_`. Consequently, there are many sequences of
digits that are just too long to be HEKINTs but which could easily be HEKA(*) or HEKSTR(*). At the Cymbal level, HEKINT constants appear as the likes of `1123811'HEKINT and `-23341'HEKINT(_short). Once read in, a HEKINT can be operated on by various fpfs just as if it were an INT. The converse is also true: if an fpp expects a HEKINT argument, it will in fact accept an INT in its place.

A HEKUINT is stored in the HEKA format in DC files but as soon as it is read into memory, it is converted into a UINT. HEKUINTs have the same subclass specifiers that UINTs do, namely, _tiny_, _short_, _long_, _huge_. The default subclassifier is _long_. Consequently, there are many sequences of digits that are just too long to be HEKUINTs but which could easily be HEKA(*) or HEKSTR(*). At the Cymbal level, HEKUINT constants appear as the likes of `1123811'HEKUINT and `13341'HEKUINT(_short). Once read in, a HEKUINT can be operated on by various fpfs just as if it were a UINT. The converse is also true: if an fpp expects a HEKUINT argument, it will in fact accept an UINT in its place.

A HEKMONEY is stored in the HEKA format in DC files but as soon as it is read into memory, it is converted into a MONEY. At the Cymbal level, HEKMONEY constants appear as the likes of `332.49'HEKMONEY. Once read in, a HEKMONEY can be operated on by various fpfs just as if it were a MONEY. The converse is also true: if an fpp expects a HEKMONEY argument, it will in fact accept a MONEY in its place. A HEKDATE is the same as a DATE with the exception that it is stored using the HEKA encoding in DC data files. At the Cymbal level, HEKDATE constants appear as the likes of `1998-06-06'HEKDATE. Consequently, as soon as it is read into memory, it is converted into a DATE. Once read in, a HEKDATE can be operated on by various fpfs just as if it were a DATE. The converse is also true: if an fpp expects a HEKDATE argument, it will in fact accept a DATE in its place.

A HEKCLOCK is the same as a CLOCK except that it is stored using the HEKA encoding in DC files. Consequently, as soon as it is read into memory, it is converted into a CLOCK. At the Cymbal level, HEKCLOCK constants appear as the likes of `12:34:45'HEKCLOCK. Once read in, a HEKCLOCK can be operated on by various fpfs just as if it were a CLOCK. The converse is also true: if an fpp expects a HEKCLOCK argument, it will in fact accept a CLOCK in its place.

A HEKDATE_CLOCK is the same as a DATE_CLOCK except that it is stored using the HEKA encoding in DC files. Consequently, as soon as it is read into memory, it is converted into a DATE_CLOCK. Once read in, a HEKDATE_CLOCK can be operated on by various fpfs just as if it were a DATE_CLOCK. The converse is also true: if an fpp expects a HEKDATE_CLOCK argument, it will in fact accept a DATE_CLOCK in its place.

A HEKTIME is stored in the HEKA format in DC files but as soon as it is read into memory, it is converted into a TIME. At the Cymbal level, HEKTIME constants appear as the likes of `3m2.49s'HEKTIME. Once read in, a HEKTIME can be operated on by various fpfs just as if it were a TIME. The converse is also true: if an fpp expects a HEKTIME argument, it will in fact accept a TIME in its place.

A DOYDATE is a date expressed as an integer beginning with two or three digits indicating the year since 1900 and followed by the day of that year being referenced. So, `94001'DOYDATE is the same as `1-1-94', and `104001'DOYDATE is the same as `1-1-2004'. When DOYDATES are Read/Write from/to the user-level, they will appear as integers (e.g., 94120). They exist in-memory as ints and they are stored in DC files as HEKA.
An ATTDAT is implemented as a HEKINT that indicates an offset number of days since 1-1-84. For example, 0 is the way ATTDAT would represent 1-1-84 and -1 is the way it would represent 12-31-83. Consequently, it is stored as a HEKA in DC data files, it is read into memory as an INT(_long_), and when printed out to or read from the user-level, it looks like a regular date. At the Cymbal level, ATTDAT constants appear as the likes of “3-4-89”ATTDAT. Once read in, an ATTDAT can be operated on by various fpps just as if it were an INT(_long_). (The converse is not true: if an fpp expects an ATTDAT argument, it will not accept an INT in its place.) ATTDATs are vestigial in character and are not recommended for use. Please use HEKDAT instead.

An ATTDAT_CLOCK is a combination of an ATTDAT with a no-longer supported CLOCK type. ATTDAT_CLOCKs have two subtypes: _mmddyyyy_12_ and _mmddyyyy_24_, depending on whether they use 12 hour or 24 hour clock formats. ATTDAT_CLOCKs are vestigial in character and are not recommended for use. Please use HEKDATE_CLOCKs instead.

5.2.3.1 BITSEQ

A BITSEQ like ‘001110011111’B or ‘11’BITSEQ is a sequence of 0s and 1s that is stored in a compact fashion in Daytona data file records. BITSEQ variables may have the * Size (which indicates that they may take values of any length) or they may have a fixed Size indicated by a positive integer. In contrast to STRINGs, where an integer Size indicates the maximum length of the STRING, for BITSEQ, an integer Size indicates the exact length of the sequence. This is because all of the binary operations on BITSEQ that define their peculiar nature require their arguments to have the same Size: consequently, static type-checking on arguments of type BITSEQ(integer>) will ensure the absence of run-time type errors. On the other hand, if BITSEQ(*) VBLs are used and at runtime, a binary operation is attempted where the two arguments are BITSEQ of different lengths, then there will be a runtime error.

To get a BITSEQ of stated length for a BITSEQ(*) value, the FUN bitseq_of_size_for_bitseq may be used as in:

```c
local: BITSEQ(44) .xxx
set .xxx = bitseq_of_size_for_bitseq( 44, (BITSEQ) "010100011" );
_Show_Exp_To(.xxx)
do Write_Line( (BITSEQ(44)) "010100011" );
```

The result of executing this program is:

```
.xxx = 01010001 10000000 00000000 00000000 00000000 0000
01010001 10000000 00000000 00000000 00000000 0000
```

Notice that any changes, in this case, the addition of 0s, occurs on the right. In particular, this right-orientation phenomenon is illustrated above for casting to a BITSEQ(44). Furthermore, the actual length of any BITSEQ(*) will be returned by using the length function.

By default, when a BITSEQ is printed out, for ease in reading, the bits are printed in groups of 8 bits separated by one blank. An example would be ‘00111001 1111’B, which for that matter, is also valid syntax for a BITSEQ constant. In fact, BITSEQ constants can be written with any number of
blanks interspersed in any way. Fortunately, for output purposes, the \texttt{FUNCTION} \texttt{str\_for\_bitseq} takes an INT second argument, \texttt{blank\_interval}, which specifies the size of the 0-1 groups that are to be separated by one blank. If this argument is 0 (which it is by default), then no blanks at all will appear in the output.

The usual operations on bit sequences are supported, including \texttt{not}, \texttt{and}, \texttt{or}, and \texttt{xor}. These are represented, respectively, by the (overloaded) infix operators (except of course \texttt{not} which is unary):

\begin{verbatim}
- * + \%
\end{verbatim}

Also, as an efficient convenience, the \texttt{Covers} predicate is \texttt{true} if and only if every bit that is 1 in the second argument is 1 in the first. The \texttt{Is\_All\_Zeros} predicate is \texttt{true} if and only if every bit in its argument BITSEQ is 0.

Consider the task of testing whether bits 5,6,7 of a BITSEQ \texttt{x} are equal to 101. The following code will set \texttt{.ans} appropriately by individually setting and unsetting specific bits using \texttt{Set\_Bit\_For} and \texttt{Unset\_Bit\_For}:

\begin{verbatim}
set .tmp = .x;
do Set\_Bit\_For( 5, .tmp );
do Unset\_Bit\_For( 6, .tmp );
do Set\_Bit\_For( 7, .tmp );
set .ans = truth( .x \% .tmp Is\_All\_Zeros );
\end{verbatim}

Note the power of the exclusive-or here in facilitating the identification of arbitrary 0-1 subsequences of BITSEQs. The \texttt{PRED Bit\_Is\_Set\_For} is \texttt{true} if and only if there is a 1 bit at the position specified by the first argument in the second argument BITSEQ.

Also, the \texttt{subbitseq} function will efficiently compute sub-BITSEQ in a way exactly analogous to the way that the \texttt{substr} function works.

The encoding used for BITSEQ FIELD and INDEX values is order-preserving with one exception. \texttt{order-preserving} means that for two strings \texttt{a} and \texttt{b} of 0s and 1s \texttt{a} is less than \texttt{b} lexicographically if and only if \texttt{encoding(a)} is less than \texttt{encoding(b)} \texttt{C-lexicographically}. The exceptional case is that BITSEQ of length 4 mod 6 do not compare correctly with strings of length 5 mod 6 or 0 mod 6; this is an inescapable result of not having enough 7 bit codes available to resolve the situation otherwise.

\subsection{IPv4-related classes}

For Internet applications, Daytona provides the IP and EP datatypes. At the Cymbal level, IP (version 4) constants look like ‘135.205.49.11’IP and ‘135.40’IP. The next subsection discusses IPv6. Here for IPv4, there are currently two implemented subclasses, IP(_heko_) and IP(_uint_), with the former being the default, i.e., IP used alone is taken to mean IP(_heko_). IP(_heko_) values are stored both in memory and on disk in a special compressed format that uses a HEKA-like encoding for each octet. This format has the additional useful property that when these (especially compressed) values (which the user doesn’t ordinarily see) are sorted as character strings, the effect is the same as sorting the corresponding IP addresses lexicographically by integer octet. As an example of IP(_heko_) ordering, ‘192.205.31.226’IP > ‘192.205.31.32’IP . The user may pad the octets with 0s as in ‘012.002.111.001’IP but the system elides leading 0s when asked to print IPs out.
IP(_uint_) turns out to be a much more useful implementation than IP(_heko_): it is more space efficient, it has more fpps, and some of its functions like masked_ip are much faster than they are for IP(_heko_). IP(_uint_) are stored as HEKA UINT(long) in DC data files and as UINT(long) in memory; of course, when printed out by Write, they appear as dotted digit string IP addresses. The UINT(long) in memory is kept in "network byte order" (i.e., big-endian) (regardless of which endian the machine is). This enables the XOR operator %%% to operate on two IP2s returning an IP2, which can be of value in computing longest prefix matches. (One inefficient but effective way to find a longest prefix match is to XOR the candidate IP2 with each of a set of network IP2s and take that IP2 with the smallest XOR value as the answer: see ip.uint.1.Q) As a convenience, IP(_uint_) can be abbreviated as IP2. In contrast to IP(_heko_), while `192.168'IP(_uint_) is valid, it will be considered to be `192.168.0.0'IP(_uint_) and printed out as 192.168.0.0 and so, the recommendation is to write all four octets when writing Cymbal IP(_uint_) constants. Daytona will accept IP2 constants that use leading zeros for the octets but suppresses leading zeros on octets when Writing IP2 objects. Chapter 7 describes IP-related functions. Here are some examples of what can be done with IP(_uint_):

```
^192.168.20.20^IP2 +10
^192.168.20.20^IP2 -10
^133.106.200.160^IP(_uint_) - ^133.106.200.150^IP(_uint_)
max( ^133.106.200.150^IP(_uint_), ^133.106.200.159^IP2 ) // or min()
fet .ip Is_In [ ^133.106.200.150^IP2 -> ^133.106.200.159^IP2 ] {} 
when( ^133.106.200.150^IP(_uint_) < ^133.106.200.159^IP2 ){}
```

To get the storage efficiency due to compression, Daytona stores IP2 as HEKA. There are occasions however when data is provided with the IP addresses stored using dotted digits. Data in such a form can still be treated as IP2 at the Cymbal level by using the appropriate Input/Output Filter_Fun. Here is an example taken from rcd.SMORGAS4:

```c#
#{ FIELD ( Local_Ip_Str_Uint )
    <Type, ( 0->1 ) IP(_uint_)> 
    <Input_Filter_Fun, ip_uint_for_str>
    <Output_Filter_Fun, str_for_ip_uint>
    <Default_Value, 112.60.0.2>
    <Min_Value, 1.1.1.1>
    <Max_Value, 135.206.255.255>
    <__Position, 4> ) #
```

Here is a comparison of IP(_heko_) to IP2:

- the IP2 datatype auto-converts into a network-byte-order UINT upon reading in from the data file whereas the IP(_heko_) datatype stays in its compressed form.
- the IP2 datatype is more storage efficient in the data file ( <= 5 bytes vs. upto 8 bytes for IP).
- IP(_heko_) is order-preserving, whereas the DC/HEKA representation of IP2 is not order-preserving, e.g., while `0.35.40'IP2 < `0.78.32'IP2 on an octet-by-octet lexicographic basis (as occurs in-memory), the HEKA representations of these IP2 are in the reverse order. Nonetheless, IP2
can still be used in range queries because IP2 values are stored in B-trees using a BITSEQ representation, not the HEKA one.

- IP(_heko_) has Starts_With, IP2 does not.
- IP2 has a full complement of arithmetic functions; IP does not.
- IP2 is optimally fast for doing masking since it just uses the C & operator; whereas IP involves string conversions coming and going and is significantly slower.

Both IP(_heko_) and IP(_uint_) store IP Version 4 addresses. Such addresses are recognized by Is_An_Ip_V4_Str.

Cymbal also supports the IPNET(_uint_) or IPNET2 type which is basically an IP2 coupled with a netmask between 1 and 32. Here is an example of its use:

```c
local: IPNET2 .ipn2_copy
set .ipn2 = `10.10.11.12/16'IPNET2;
set .ip2 = .ipn2.Ip;
set .mask = .ipn2.Mask;
set .ipn2_copy = [.ip2.Ip, .ipn2.Mask ]; // only if .ipn2_copy is typed explicitly
set .ipn2_copy = ipnet_for_ip_and_mask( .ip2, .mask );
```

Note the use of the STRUCT member selectors Ip and Mask. One can construct an IPNET2 from an IP2 and a mask by using the (overloaded) function ipnet_for_ip_and_mask. The in-memory and B-tree sorting of IPNET2 is lexicographic first by their Ip components according to the IP2 ordering and then by their Mask components according to the INT ordering.

The applying of integer masks can be done by using the masked_ip function. It is important to keep in mind that Daytona will accept Ip components of an IPNET2 structure that have not been masked with the accompanying mask. In other words, it is not required that:

```
masked_ip( .ipn2.Ip, .ipn2.Mask ) = .ipn2.Ip
```

If it's true, then it's fine as well. It is left to the user to determine what is meant by an IPNET2 when the Ip component is not equal to its masked form.

Since IP(_heko_) addresses may well be considered to be sensitive information, the EP datatype enables them to printed as strings of letters, as with the EP constant `bvtaqppubys'EP. The advantage gained here is that no-one can confuse this with an actual IP address and actually, there is no easy-to-remember way to translate back and forth between an EP and an IP. EP values are stored in memory and on disk in the same format as IP(_heko_) values.

Conceptually, an IPORT is an IP address plus a port number; in that regard, it is identical to a socket. There are four subclasses of IPORT, with the first two for IPv4 and the last two for IPv6: IPORT(_hekop_), IPORT(_uint_), IPORT(_v6_), and IPORT(_v6z_), where the default subclass is the original IPORT(_hekop_). An abbreviation for IPORT(_uint_) is IPORT2. An IPv4 IPORT is an IPv4 IP plus a port number as in `12.12.10.80'IPORT. Ports introduced by dots can also be used as in `192.205.31.226.101'IPORT but for Cymbal Writing purposes, the colon separator is used to honor a not uncommon practice and to be consistent with the hostname-port syntax used for URLs (and hence
IHOST_PORTS). Ports are constrained to be between 0 and 65535, inclusive. IPORT values are stored in memory and on disk in a special compressed format. This format has the additional useful property that when these (especially compressed) values (which the user doesn’t ordinarily see) are sorted as character strings, the effect is the same as sorting the corresponding IPORT addresses lexicographically by integer octet and lastly by integer port number (with no leading 0s). In keeping with the standards, the largest allowable port number is 65535.

The functions ip_of and port_of return their appropriate pieces of an IPORT. The method for referring to the IP and port components of an IPORT2 is by means of STRUCT element selection: .x.Ip and .x.Port. One can construct an IPORT2 from an IP2 and a port number by using the (overloaded) function iport_for_ip_and_port.

Since there are rare occasions when the user wishes to Write a IP out as a IP (i.e., to avoid the default conversion into a sequence of digits and dots), the function ip_as_str can be applied to a IP in order to return a STR consisting of precisely the IP characters of the argument. Note that this is different from ip_heko_for_str and str_for_ip_heko, which convert back and forth between the plain text and compressed forms of IP. Conversely, there are occasions when a sequence of characters is read in from the outside and it is known to be in the IP encoding. The user may instruct Daytona to consider this sequence of characters to be a IP (without any transformation being performed) by feeding it as the sole argument to str_as_ip_heko. Perhaps the key thing to remember about ip_heko_as_str and str_as_ip_heko is that they are identity functions: they do not change their arguments in any way. Instead, they just convince Daytona to consider their argument (and hence return value) to be a member of a different type. Likewise for iport_hekop_as_str and str_as_iport_hekop.

IHOST and IHOST_PORT are the domain-name-based hostname analogs of IP and IPORT. As exemplified by `trntwn.acme.org`IHOST and `www.mrstwn.univ.edu:80`IHOST_PORT, these are the read-friendly forms of IP/IPORT that DNS (Domain Name Service) servers translate into the IP addresses that the machines use to make connections. For consistency with RFC 921 and 952 (but not precise compliance), an IHOST constant must be a string which matches the following regular expression:

```
^[a-zA-Z][-a-zA-Z0-9.]*[a-zA-Z0-9.]*$
```

IHOST_PORT constants must begin with an IHOST and contain a colon with the port portion consisting of a sequence of digits representing an integer between 0 and 65535, inclusive. Functions ihost_for_ip and ip_for_ihost for converting back and forth between IHOST and IP are described in Chapter 7. The functions ip_of and port_of return their appropriate pieces of an IHOST_PORT.

### 5.2.3.3 IPv6-related classes

In addition to its types that support IPv4, Cymbal has additional types that support IPv6. IPv6 internet addresses are 128 bit unsigned integers whose 16 8-bit bytes are stored in memory in a big-endian way, i.e., with the most significant byte having the lowest computer memory address. Clearly then, in memory, an IPv6 address is 4 times the size of an IPv4 address. The first (or high-end, reading from left-to-right) 64 bits are intended to be used to identify the network, with the last 16 bits of those intended for subnetting purposes, and then the second 64 bits are intended to identify the host computer. The following examples illustrate the official syntax for representing IPv6 addresses at the...
human-readable level. They are all based on the fundamental representational form of 8 groups of 4 hex codes separated by colons:

2001:0db8:85a3:08d3:1319:8a2e:0070:7334
2001:0db8:85a3:8D3:1319:8A2E:70:7334
2001:0db8:0000:0000:0000:0000:1428:57ab
2001:0db8:0:0:0:0:1428:57ab
2001:0db8:0::0:1428:57ab
2001:0db0:0000:0000:0000:0000:0428:57ab
1080::8:800:200C:417A
FF01::43
::1
::
23AB:0123:6:0def::192.168.10.10
::13.1.68.3
::FFFF:13.1.68.3

These examples illustrate that the case of the hex letters is immaterial (to Daytona), that leading 0s in hex quartets can be omitted, and that consecutive quartets of hex 0s can be represented by a double-colon ::. Of course, only one double-colon can be used since the existence of one is a request to replace it with as many quartets of 0s as are needed to fill out the address to the necessary 128-bit size. The last three examples show a hybrid IPv6/IPv4 syntax which is deprecated in some circles but which is used by Solaris and which Cymbal supports in any event: basically this syntax consists of an IPv6 address whose last two hex quartets have been replaced with a standard IPv4 dotted representation whose UINT equivalent is equal to the UINT equivalent of the last two hex quartets.

For emphasis, the first two of the next three IPv6 addresses are the same and are different from the third:

FFFF:A::
FFFF:000A::
FFFF:A000::

Cymbal has two IPv6 classes: IP(_v6_) and IP(_v6z_). For shorthand, these are also known to the system as IP6 and IP6Z, respectively. They carry the same information and indeed, in memory, they appear exactly the same, i.e., as an array of four UINT(_long_) stored in network-byte-order, i.e., big-endian. And any fpp that can accept an instance of one of them as an argument can, without further ado, accept an instance of the other -- because in fact, they both have the same representation in memory. The only ways in which they differ are in the syntax they use when Writing them on output to the user or else when Daytona stores them as contents of FIELDS in DC data FILEs. Specifically, when an IP6 is written out to user-level using Write, then it will appear with all 8 hex quartets completely and explicitly printed and separated by colons; on the other hand, when an IP6Z is Written out, it is written out in the shortest possible legal form as achieved by using the double-colon notation and eliminating leading 0s in quartets.

Analogously, for storage in DC data files, IP6Z is written out economically in such a way that
pretty much the longest sequence of zeros is represented by a single code byte. Both are written out using the same field-compression codes as BITSEQ but as was just said, IP6Z takes an extra step and squeezes out the longest possible sequence of 0s before encoding.

Here is an example of casting to IP6Z in order to achieve a short representation for printing:

\[
\% \text{ DS Expr} \\
(IP6Z) ^ \text{2001:0db0:0000:0000:0000:0428:57ab} \text{IP6} ^ \text{D} \\
2001:DB0::428:57AB
\]

The DC data file encoding of the IP6 data is order-preserving, meaning that one encoded IP6 string is less than another in lexicographic C-character order if and only if the first is less than the other as 128-bit integers.

One implication is that IP6Z are good for representing what are called IPv4-compatible IPv6 addresses and additionally, IPv4-mapped IPv6 addresses. Here, respectively, are examples repeated from above:

\[
^::13.1.68.3^\text{IP6Z} \\
^::FFFF:13.1.68.3^\text{IP6Z}
\]

While these are true IPv6 addresses, when treated as IP(v6z_), they take up minimal space when printed out and when stored in a Daytona data file. Specifically, an IPv4-compatible IP6Z address would typically take 7 bytes in a datafile compared to the 22 bytes that a full IP6 would take (and to the 16 bytes that storing four 32-bit ints would take). Contrast this with the at most 5 bytes that an IP2 would take. Note that IPv4-compatible addresses are deprecated by RFC 4291. So, when doing C-style casting from an IP2 address to IP6, the casting will be done so as to produce an IPv4-mapped address (not the "compatible" one). If for some reason, an IPv4-compatible address is desired, then the function \text{ip_v6_for_compat_ip_uint} can be used as opposed to the \text{ip_v6_for_ip_uint} function that is used behind the scenes when doing an IP6 cast at the Cymbal level as in \text{(IP6)\_my_ip2} to get an IPv4-mapped IPv6 address. The following \text{PRED[ IP(v6) ]} serve to identify the indicated kind of IPv6 address: \text{Is_A_Mapped_Ip_V4} and \text{Is_A_Compat_Ip_V4}. Their companion \text{Is_Some_Embedded_Ip_V4} which is true if either one of the two preceding are true.

The \text{PRED Is_An_Ip_V6_Str} identifies putative IP(_v6_) and IP(_v6z_) strings.

\text{INT(_huge_)} can be added and subtracted to/from IP(_v6_) and even more, IP(_v6_) can be added and subtracted from each other, which is of use in doing masking. This also essentially gives Cymbal the ability to work with 128-bit integers.

Cymbal also supports the IPNET(_v6_) or IPNET6 type (respectively IPNET(_v6z_) / IPNET6Z) which is basically an IP6 coupled with a netmask between 1 and 128. In other words, IPNET uses the CIDR notation where the "netmask" number following the slash is the number of bits starting from the left that identify the network. Here is an example of its use:
local: IPNET6 .ipn6_copy
set .ipn6 = ^1111:2222:3333:4444::CC/64^IPNET6;
set .ip6 = .ipn6.Ip;
set .mask = .ipn6.Mask;

Note the use the STRUCT member selectors Ip and Mask. One can construct an IPNET6 from an IP6 and a mask by using the (overloaded) function ipnet_for_ip_and_mask.

As with IP6 and IP6Z, any fpp that takes an IPNET6 argument will accept an IPNET6Z argument.

The applying of integer masks can be done by using the masked_ip function which, thanks to overloading, works with IP6 addresses as well as IP2. It is important to keep in mind that Daytona will accept Ip components of an IPNET6 structure that have not been masked with the accompanying mask. In other words, it is not required that:


If it’s true, then it’s fine as well. It is left to the user to determine what is meant by an IPNET6 when the Ip component is not equal to its masked form.

The in-memory and B-tree sorting of IPNET6 is lexicographic first by their Ip components according to the IP6 ordering and then by their Mask components according to the INT ordering. When stored in DC data files, IPNET6 and IPNET6Z reuse the same field-compression encoding that IP6 and IP6Z use.

In the realm of IPv6, Daytona also has IPORT(v6) and IPORT(v6z), with their abbreviations IPORT6 and IPORT6Z. The syntax of the constants is necessarily different and involves bracketing the IP address as in:

^[2001:1890:1110:3::2000:50]:22^IPORT6Z,
^[::1]:18128^IPORT6Z

The method for referring to the IP6 and port components of an IPORT6 is by means of STRUCT element selection: .x.Ip and .x.Port. One can construct an IPORT6 from an IP6 and a port number by using the (overloaded) function iport_for_ip_and_port.

The in-memory and B-tree sorting of IPORT6 is lexicographic first by their Ip components according to the IP6 ordering and then by their Port components according to the INT ordering. When stored in DC data files, IPORT6 and IPORT6Z reuse the same field-compression encoding that IP6 and IP6Z use.

Chapter 7 contains descriptions of additional IP networking functions.

5.2.4 Class Syntax

In general, any completely capitalized word in Cymbal is taken to be either the name of an object class or the name of a generic member of that class. For example, in a given application, “EMPLOYEES” might be the name of the set of all employee records whereupon “EMPLOYEE”, the singular form, would be considered to signify a generic member of the class of employee records. This
singular-plural convention is suggested and is not enforced by the system. Object class names must begin with a capital letter and then continue with zero or more capital letters, underscores, and digits.

Although not classes in the language per se, UPPER, LOWER, and UPLOW are relevant classes of STRs as far as the Cymbal parser is concerned. For example, all variable names must be LOWERs and all predicate names must be UPLOWS, in the context of the C_external exceptions described below. The members of all 3 classes consist of sequences of letters, digits, and underscores that do not begin with a digit. UPPERS must have at least one letter and all their letters must be upper-case. LOWERS are defined analogously. The first letter in an UPLOW must be upper-case and some subsequent letter must be lower-case and of course, the ULOW cannot be all digits. (See also the discussion of UPLOWS in Chapter 4.) Syntactically, LOWUPS are considered LOWERs.

5.3 Simple Cymbal Variables

Cymbal variables take members of object classes as their values. So, for example, the variable \( x \) could take the value 27 from the class of INTs. All variable names must be LOWERs. (However, no such restriction applies to names for C_external global variables when they are declared or defined prior to use in the Cymbal file that uses them.) To refer to the value of a variable, merely prefix its name by a period or dot. So, if \( x \) had the value 27, then to say in Cymbal:

\[ .x = 27 \]

would be to correctly assert that the value of \( x \) is equal to 27. In short, Cymbal makes a strong and consistent distinction between a variable (like \( x \)) and its value (like \( .x \), read "the value of \( x \)”). Variables should be thought of as named boxes with the value of the variable being the contents of the box. When the user wishes to talk about the contents of the box instead of the box itself, then the variable name must be preceded by a dot.

In addition to the conceptual clarity maintained by this distinction, the dots are consistent with the high-level nature of Cymbal. For example, in C, *, & , and -> are used to accomplish what dots alone accomplish in Cymbal. In particular, Cymbal has no analog of the & address-of operator because machine addresses are far too low level of a concept for a 4GL and as illustrated by Cymbal, they are not needed.

Also, as discussed by the authors of BLISS, a system programming language [Wulf, et al., 1971], an advantage of the dot notation is that the meaning of a variable’s name is context-independent, i.e., it always means the variable. This contrasts with the situation in C where an identifier means one thing on the left-hand side of ‘=’ and another thing on the right-hand side.

The use of the dots also helps the Cymbal parser out quite a bit. In order to support 4GL user-friendliness, Cymbal usually does not require that the user explicitly define scalar variables. Consequently, there must be some syntax to help the parser distinguish among keywords, function calls, and variables, all of which are in the same syntactic class, precisely that of lower-case strings. For the variables, it’s (frequently) the dots. The syntax used for defining and declaring variables of various types is given towards the end of Chapter 6.

As will be seen shortly, the presence of composite classes inspires an extension of variables to components of composite class members.
5.4 Composite Classes

Cymbal also has composite or compound classes whose elements are formed by creating structures or aggregates of elements of other classes like INTs and sometimes even other composite classes. This manual will use the term scalar to denote elements of non-composite classes, i.e., objects which are considered to be atomic or primitive by the system. Several such builtin composite classes are SETS, BUNCHES, LISTs, TUPLEs, INTERVALs, ARRAYs, and CHANs.

5.4.1 SETS, BUNCHes, LISTs, TUPLEs

A SET is a collection of indefinite or unspecified (at compile-time) size of objects of the same type that is not considered to be ordered in any particular way and which does not contain any duplicates. A BUNCH is a SET of objects except that it is of known (at compile-time), specified finite size. A BUNCH constant is denoted syntactically by an explicit sequence of comma-separated objects enclosed with matching curly braces. Consider:

{ "Katie", "Joan", "Harry", .y, new_gma_host(), "Katie", "Bryant", "Willard" }

When Daytona is asked to enumerate the elements of the above BUNCH, each distinct element will only be produced once, even though "Katie" appears twice in the listing and even if .y and new_gma_host() evaluate to some other element of the BUNCH. SETS may also be defined implicitly as would be the case with the SET-former boxes discussed subsequently in Chapter 12.

A LIST is a sequence of objects of the same type which may be added to or deleted from; its components may be selected by ordinal position in the LIST. Note that duplicate occurrences of an object in a LIST are retained so that in an enumeration of a LIST, each of those duplicates will appear in the appropriate order. A TUPLE is a sequence of objects of possibly differing types whose length is fixed and unchangeable. A TUPLE constant is denoted syntactically by an explicit sequence of comma-separated objects enclosed with matching brackets as illustrated by:

[ +1, .x, "abc", 12e5, -1, 1 ]

This same bracket syntax can be used to represent a particular LIST, as long as the components all have the same type. LISTS can also be defined implicitly as would be the case with the LIST-former boxes discussed subsequently in Chapter 12. At the C-level, Daytona implements TUPLEs as C structs; LISTS and SETS are implemented as boxes, which in general are very complicated data structures.

(Caveat: the word ‘TUPLE’ is being used here in the sense it appears in mainstream mathematics, i.e., as an element of a cross-product. Note that this is not the same as the sense often used for relational databases; there, a TUPLE is an unordered collection of field-name-tagged scalars. Such constructs should more properly be called RECORDS or STRUCTS.)

Variables are allowed to assume TUPLE values. Such variables are defined/declared using syntax like:
local: TUPLE[ INT, DATE .exit, MONEY .salary ] .status
set .status = [ 22, ^2001-01-01^, ^234.56^MONEY ];
set .s = .status#3;
set .status#1 += 100;
do Write_Words( .status ); // all TUPLE components are written

Note that the infix tuple-element-selection operator # is used to reference the elements of a TUPLE by ordinal position starting with 1. These ordinal positions must be given by INT constants; they cannot be the results of expressions: this is because Daytona has to be able to determine the type of # expression at compile-time. Keyword tags like exit are NOT used to refer to elements of TUPLES: they are strictly for mnemonic purposes. Note also that a TUPLE-valued variable dereference can be given as an argument to the Write PROC with the effect that all the components of the TUPLE will be written out in order.

It is important to realize that this bit of code would not compile were it not for the presence of the definition for the VBL status. The reason is that while the bracket syntax always accurately conveys the notion of a sequence of objects, there remains the question of how to infer the type of any VBL that is said to be equal to that TUPLE. By default, without any other information available, Daytona will infer the type to be LIST VBL, whose values are indefinitely long LISTs of objects of the same type. Obviously, that endeavour will fail above if the status VBL definition is not present because the TUPLE constant contains both an INT and a DATE. Consequently, the explicit TUPLE VBL definition is necessary.

As a convenience in writing a TUPLE types with many component types, a multiplicity indicator may be used. In that regard, the following two types are in fact the same:

    TUPLE[ (2) STR, DATE, (3) INT ]
    TUPLE[ STR, STR, DATE, INT, INT, INT ]

Cymbal supports nesting TUPLES within TUPLES. TUPLES can be arguments to fpps but the corresponding parameter VBLs must be constant or alias parameters VBLs. They are not yet allowed to be return values from fpps. RECORD_CLASS FIELDs can have TUPLE type.

In addition to SETS and LISTs of scalars, Cymbal supports SETS and LISTs of TUPLES, as seen by:

    { [ 1, 2.2 ], [ 2, 1.1 ], [ 3, 2.2 ],
      [ 2, 3.3 ], [ 3, 1.1 ], [ 1, 3.3 ] }

These can also be indexed: see Chapter 12 for details on what are called boxes.

5.4.2 INTERVALs

INTERVALs are collections of objects that lie between (and including) two boundary objects, according to some ordering criterion. INTERVALs, therefore, are type-homogeneous LISTs that are defined by special arrow syntax that indicates how to construct the elements of the INTERVAL. Here are several:
Note that half-open-on-the-right INTERVALs are supported as illustrated by \([ 2 \rightarrow 78 )\) which does not contain the INTEGER 78. (INTERVALs open on the left are not supported.) These can be convenient for (indexed) range queries. Note that there are occasions when unbounded INTERVALs are permitted, such as in specifying arguments to some functions. If an INT valued variable is asserted to be in an INTERVAL bounded by INTs, then, of course, it is only the integers in the interval that are of any consequence, in which event the INTERVAL is considered to be a sequence of INTs.

By employing an optional \(by\), lattices may be represented as in:

\[
\begin{align*}
[ & 5 \rightarrow 25 \ by \ 1 ] \\
[ & 5.0 \rightarrow 25.0 \ by \ 2.5 ] \\
[ & 5.0 \rightarrow -5.0 \ by \ -2.5 ] \\
[ & "1-1-84" \rightarrow "6-1-84" \ by \ 7 ] \\
[ & "abc" \rightarrow "abcXXXXX" \ by \ "X" ]
\end{align*}
\]

The generation of values for an INTERVAL ceases as soon as the next element generated exceeds the end-point value, meaning being larger if it is an increasing INTERVAL of values and smaller if decreasing.

Please note that there is no INTERVAL type per se, despite the use of a capitalized name. Just think of INTERVALs as being syntactic abbreviations for LISTS of the appropriate type, as \([ 1 \rightarrow 4 ]\) being an abbreviation for \([ 1, 2, 3, 4 ]\), which is a \(\text{LIST}[\ \text{INT} \ ].\)

By using the \(modulo\) keyword, circular lattices can be generated:

\[
\begin{align*}
\text{for} \_ \text{each time} \ .jj \ \text{Is} \_ \text{In} \ [ & 4 \rightarrow 2 \ by \ 1 \ \text{modulo} \ 8 ] \\
& \text{do} \\
& \quad \text{do} \ \text{Write} \_ \text{Line}(\ .jj); \\
& \text{do}
\end{align*}
\]

Here \(jj\) begins by assuming the value 4. Each time it is incremented by 1, it gets that value modulo 8, i.e., it’s as if at the end of each iteration:

\[
\text{set} \ .jj = (\ .jj+1) \ \% \ 8
\]

The loop stops when the final value is reached. For this example, the output is:
Here then is a critical difference: in the modulo situation, since the values may well decrease as they are generated (inasmuch as each newly generated value is the same as that value modulo the modulus), the termination condition has to be that of equalling the “upper” bound — in other words, being greater than or equal will simply not do as it does with traditional INTERVALs. So, some care must be taken. Consider:

```plaintext
fet .clk Is_In [ ^00:00^CLOCK -> ^23:00^CLOCK by ^1h^TIME ]
{
   do Write_Line( .clk );
}
```

This will generate an infinite sequence of values because once ^23:00^CLOCK is reached, ^1h^TIME will be added to it but since there simply are not any CLOCKS that are greater than or equal to ^24:00^CLOCK, the system’s addition will result in ^00:00^CLOCK which is not going to cause termination of the loop. Consequently, since the addition of a TIME to a CLOCK is quietly applying a modulus anyway, in order to get this INTERVAL to behave as expected, it must be modified to officially be a modulo-based INTERVAL:

```plaintext
fet .clk Is_In [ ^00:00^CLOCK -> ^23:00^CLOCK by ^1h^TIME modulo ^24h^TIME ]
{
   do Write_Line( .clk );
}
```

The `by_fun` argument is used to instruct Daytona to generate values for the INTERVAL by applying the FUNCTION to the current INTERVAL value in order to compute the next (interval.2.Q).

```plaintext
for_each_time .x Is_In [ 1 -> 5 by_fun int_succ ]
do {
   do Write_Line( .x );
}
```

```plaintext
define INT FUN( INT .prev_idx ) int_succ
{
   return( .prev_idx +1 );
}
```

Here is the corresponding output:
If we think of an INTERVAL as a function with arguments that returns a sequence of values, then the following import characterizes the possibilities (and is in fact used by the system for type checking):

```
define CLASS INTERVAL_BOUND_TYPE
  = INT|UINT|MODULO|FLT|MONEY|CLOCK|TIME|DATE|ATTDATE|DOYDATE|DATE_CLOCK|STR
  |HEKMONEY|HEKINT|HEKCLOCK|HEKTIME
import:
  INTERVAL FUN(
    ( 0->1 ) by INTERVAL_BOUND_TYPE,
    ( 0->1 ) modulo INTERVAL_BOUND_TYPE,
    ( 0->1 ) by_fun manifest INTERVAL_BOUND_TYPE FUN( INTERVAL_BOUND_TYPE ),
    INTERVAL_BOUND_TYPE,
    ( 0->1 ) INTERVAL_BOUND_TYPE
  ) any_interval
```

The default by argument for a given class can be found in sys.env.cy by determining the value for the keyword `with_default_interval_by_arg` in the definition for the class.

TUPLE/BUNCH and INTERVAL syntax can be merged:

```
[ 1, 2, 5 -> 25 by 5, 50 -> 100 by 10, 1000 ]

{ 1, 2, 5 -> 25 by 5, 50 -> 100 by 10, 1000 }
```

The first expression represents a LIST that begins with 1 and 2, continues from 5 to 25 by 5, etc. The second is the corresponding SET.

### 5.4.3 ARRAYs

Cymbal also supports multi-dimensional arrays of scalar and TUPLE elements of a fixed class. These arrays may be indexed by ranges of INTEGERS or by STRINGs or other THINGs in which case they are associative arrays. The essence of an array is that it is a discrete map that associates a scalar or TUPLE of scalars in its finite domain with a scalar or TUPLE of scalars in its (finite) range. What makes it discrete is that the association is given by explicitly saying what the range element is for each domain element, one by one, as opposed to defining these associations by a generic algorithm. Here are some sample array elements:

```
.x[ 14 ]
.y[ 14, 15 ]
.z[ "abc" ]
.w[ "abc", 14, "12-1-89" ]
```
Notice that INTs, DATES, and STRINGs are being used here to index these arrays. Details on how to define such ARRAYS are given towards the end of Chapter 6 and in Chapter 11, which deals exclusively with associative arrays.

5.4.4 CHANNELs

The CHAN type is for Cymbal I/O channels and is described much more fully in Chapter 8. Channels have such subtypes as _file_ and _string_ and _pipe_. CHANS are not Writable although there are certain members have names, e.g., _stdin_, _stdout_, _stderr_, _cmd_line_, and _popkorn_. In the current release of Daytona, CHANS are scalars. In a future release of Daytona, ‘CHAN’ will be the name of a subclass of STRUCTS.

5.4.5 STRUCTs

User-defined STRUCTs are not yet officially available in Cymbal. However, when they are, they will have syntax like that illustrated in the following import of the any_bundle VBL in sys.env.cy:

```
import: STRUCT{ STR(=) .Name, INT .Index, INT .Tendril_Cnt } VBL any_bundle
```

This system import conveys that any BUNDLE can be thought of as a STRUCT with several members. Suppose bun is a BUNDLE. Then .bun.Tendril_Cnt is the number of TENDRILs associated with .bun (and this is supported by the system now). This shows (the conventional) member dereferencing syntax for STRUCT members. Obviously, the member “tags” like Tendril_Cnt, which must be UPLOW, are mandatory for STRUCTS (which they are not for TUPLEs) because STRUCT members are only known by their tags, there being no notion of position to rely on.

5.4.6 DESCS

A Cymbal DESC or DESCRIPTION is one of these trees of nested attribute-value pairs using #{}# syntax that are used, for example, as record class descriptions (rcds) that are stored in application archives (aars). They do not play much of a role in Cymbal at this time but they will.

5.5 VBL VBLS: Pointers in Cymbal

A variable in Cymbal is conceptualized as a possibly named container for Cymbal objects. Expanding on before, the INT variable x is then a container for an INT and the INT it contains is .x, the value of x. For reasons of efficiency, it turns out that it is very useful to consider these containers, these VBLS, as values that could themselves be stored in other containers, be the values of other VBLS. Thus, the notion of a VBL VBL arises, i.e., a VARIABLE whose value is another VARIABLE. This is Cymbal’s analog of the pointer concept used in other languages. Note that as befitting a high-level language, the Cymbal definition makes no reference to machine addresses. Here is an example of a VBL VBL in use:
local:  INT .x = 5
        INT VBL VBL y
        INT ..z
set .y = x;
do Write_Words( "..y =", ..y );
set .z = .y;
do Write_Words( "..z =", ..z );
set ..z++;
do Write_Words( ".x =", .x );
do Write_Words( "..y =", ..y );
do Write_Words( "..z =", ..z );
// do Write_Words( ".z =", .z );  // .z is not Writable

Here is the output:

..y = 5
..z = 5
.x = 6
..y = 6
..z = 6

Note that \( y \) and \( z \) have the same type. Observe how the first assignment sets the value of \( y \) to be the VBL \( x \) itself, not the value, which would be 5, i.e., \( .x \). The second assignment sets the value of \( z \) to be the VBL that is the value of \( y \), i.e., \( x \). The third assignment increments the VBL that is the value of \( z \), which of course is \( x \), as is indicated by the three Write statements.

A VBL VBL can be set to the null pointer, i.e., a C NULL, by using the \_null_vbl\_ constant.

5.6 Cymbal Variables Extended

Not only does Cymbal support syntax for variables whose values are elements of composite classes, but it also supports syntax for referring to variables that are elements of composite class variables. For example, suppose that \( x \) is an ARRAY variable that maps INTs to STRs. Then \( x[4] \), for example, is a STR VBL whose value is the STR \( .x[4] \). In general, a Cymbal variable has the following syntax:

Variables

\[
\text{Variables} ::= \quad \text{lower} \\
\text{Variable} \quad \text{Tuple} \\
\text{Variable} \quad \# \text{IntPosition} \\
\text{Variable} \quad . \text{StructMbr} \\
\]

\[
\text{ValCall} ::= \quad [ \quad . \quad ]+ \quad \text{Variable} \\
\]

A ValCall is the value of a VBL.

These VBLs that are properly contained within VBLs taking composite class values are particularly
useful when working with VBL VBLs, as illustrated by:

```plaintext
local: TUPLE[ INT, FLT ] .x
       FLT ..y
set .x#2 = 3.4;
set .y = x#2;
set ..y = ..y * 10.0;
do Write_Words(".x =", .x);
do Write_Words("..y =", ..y);
```

The output is:

```
.x = 0 34.0
..y = 34.0
```

The second assignment sets y’s value to be the second container/VBL contained in the TUPLE container for values of x.

### 5.6.1 Composite Type Component Selection Precedences

Consider this arbitrary expression:

```plaintext
set .x = .....y[4].Age("abc")#5[6].Time#7("def");
```

How does Daytona make sense of it by knowing what to components to associate with which? The rules are very simple:

1. Unary dots indicating VBL dereferences are right associative, meaning that they group from right-to-left.
2. Everything else (binary dot, #, [], ()) is left associative, meaning that they group from left-to-right, just like English is read.
3. Unary dots (for VBL dereferencing) bind tighter than the binary selection operators.

All built-in precedences, not only these, can be overridden with parentheses, providing the resulting expression makes sense. Or parentheses can be used to make the implicit grouping explicit as is done for this assignment that is equivalent to its predecessor:

```plaintext
set .x = (((((((((.(.(.(.(.y)))))))[4]).Age("abc")#5)[6]).Time)#7)("def");
```

(FYI, in the example, the substrings ("abc") and ("def") indicate STRUCT/TUPLE component function calls, which are not yet supported for general use.)

### 5.7 Cymbal Functions

Cymbal has a rich variety of built-in functions available and more will be added as there is sufficient user interest. A small sampling of these functions is given here. The rest are discussed in detail in Chapter 7 and Chapter 8. In general, names of FUNCTIONS used in Cymbal are LOWERS. (No such restriction applies to names for C_external Cymbal function tasks or imported C functions.
when they are declared or defined prior to use in the Cymbal file that uses them.)

There are, of course, the usual arithmetic functions +, −, *, /, %, and <*> (aka pow) which obey the customary precedences and associativities. (**x % y is the integer remainder of dividing x by y and pow(x, 4) is x raised to the 4th power). The functions floor, ceil, abs, sqrt, exp, sin, cos, and tan are also available as are two argument min and max.

Cymbal functions are often overloaded which means that they can take arguments of different types and nonetheless figure out the right thing to do. This is certainly true for the arithmetic functions which can take either FLTs or INTs as arguments. What’s more, the + function can even take STR arguments in which event it operates as the concatenation function whereby:

"abc" + "defg" = "abcdefg"

Other string functions include strlen (which returns the length of a string) and substr (where substr("abcde", 2, 3) = "bcd" which is the substring of "abcde" which begins at position 2 and is three characters long). Other STRING functions such as a substitution function are mentioned in $DS_DIR/sys.env.cy. There are DATE functions which, for example, allow users to add 5 days to a date and get a date or to compute the difference in days between two dates.

The preceding example function calls are illustrative of the following grammar productions:

```
FunCall ::= function ( SubjectSeq )
| function ( [ KeywdArg ]* )
| + Subject | − Subject
| Subject infixop Subject
| AggFunCall
;

infixop ::= − | + | * | / | % | **
;

KeywdArg ::= lower
| preposition Subject | preposition Aggregate
;
```

Other than the AggFunCalls which are discussed in Chapter 14, the only syntax described above that hasn’t already been illustrated is the function call syntax which makes use of keyword arguments.

### 5.7.1 Keyword Arguments

Cymbal makes extensive use of keyword arguments, not only here with function calls, but also with satisfaction claims and procedure calls. The primary advantages of keyword arguments are that the arguments carry tags that serve to explicitly identify their role in the call, that the keyword arguments can appear in any order, and that since keywords can have default arguments, the call itself can be shorter by virtue of the user opting to omit a keyword argument pair, knowing that the default will be used instead. The tokens function, for example, uses keyword arguments:
set [ .x, .y, .z ] = tokens( for .some_str matching .some_pattern );
set [ .x, .y, .z ] = tokens( matching .pat for .new_str but_if_absent [ " " ] );

As discussed in Chapter 7, *tokens* is a FUNCTION returning a LIST; *tokens* calls must include STR and RE arguments, respectively, for the keywords *for* and *matching*. A *but_if_absent* keyword argument is optional: if it is not present in the call, then no particular default value is assumed but rather a certain default behavior for *tokens* is assumed.

Since, as shown in sys.env.cy, for the *new_channel* FUNCTION, the default value for the *via* keyword is *_file_* and the default value for *with_mode* is *_read_*, the following two function calls have identical effect:

```
new_channel( via _file_ with_mode _read_ for .chan_name )
new_channel( for .chan_name )
```

Cymbal also supports no-argument keywords which have the effect of being toggles. For example, *with_locking* is a no-argument keyword for *new_channel*:

```
new_channel( for .chan_name with_locking )
```

Since *with_locking* is present, the channel will be opened with locking; if it were absent, then locking would not be used. No-argument keywords provide a more aesthetic way to achieve the same functionality provided by BOOLEAN-valued-argument keywords, as illustrated by:

```
with_notification _false_
```

As described in Chapter 3, the user may add his or her own functions to the system. The convention for naming these functions is that they must begin with a lower-case letter and continue with zero or more underscores, digits, and lower-case letters.

```
In order to distinguish function calls from keyword arguments, it is necessary that there be no space between the function name and the open-parenthesis for its argument list.
```

### 5.8 Cymbal Assertions And Built-In Predicates

#### 5.8.1 Cymbal Satisfaction Claims

Both the procedural and declarative portions of the language make use of logic assertions which will be the topic of the remainder of this chapter.

The simplest Cymbal assertions are satisfaction claims. A *satisfaction claim* (*satclaim* for short) is a sentence consisting of a subject and a predicate, where the claim is that the subject satisfies the predicate. For example, the equality relationship is a predicate. In Cymbal, it goes by the name *Equals* or by the shorter name *Eq* or else by the conventional infix ‘=*’ operator. Here are four equivalent Cymbal equality satisfaction claims:
3 = .x
3 Equals .x
Equal[3,.x]
3 Eq .x

Here the subject is the TUPLE [3,.x] and the predicate is Equals. Note that in general, the subject of a satisfaction claim is a TUPLE of one or more objects. These objects are frequently called terms. The class of terms includes constants, variable dereferences, and function calls. Variables, functions, predicates, and procedures appearing represented by their names (like x) are considered to be constants, hence terms. Other terms include ones denoted by the syntax for TUPLE, STRUCT, LIST, SET, ARRAY, and DESC objects (such as (2)) as well as terms constructed by denoting ARRAY elements (such as .a[1]) or selecting components of TUPLES and STRUCTS (as illustrated by .tu#2).

Obviously, most people tend to use the 3 = .x variety for Equals; the point though is that in general, all satisfaction claims can be written in these 3 generic ways, which we illustrate now with the Is_In predicate:

- The classical symbolic logic way with predicate first:
  Is_In[3,[0,5,10]]

- The infix way with embedded predicate:
  3 Is_In [0,5,10]
  [47,24] Is_In .bbb in_lexico_order sorted_by_spec[2,-1]
  Note the use of of keywords like sorted by spec which will be discussed further below.

- The infix way with special-character operators like ‘=’:
  not this time! too bad: Is_In does not have any operator syntax

The equals sign in Cymbal means the Equals predicate and does not mean assignment as it sometimes does in procedural languages. This is true even in the procedural portion of Cymbal as will be seen in the next chapter. Consequently, Daytona considers the following two satisfaction claims to be equivalent:

\[
x = 23 + \exp\left(\frac{y}{2.0}\right)
\]
\[
23 + \exp\left(\frac{y}{2.0}\right) = x
\]

Each one asserts that the value of \(x\) is equal to the other expression. While it is true that the Daytona implementation sometimes generates a low-level assignment in response to seeing an equals sign in a declarative request, Cymbal queries are most easily understood when = is always read as meaning Equals and never as meaning assignment.

Here are some more satisfaction claims that illustrate Cymbal’s support for the inequality predicates and the fact that all manner of terms can appear in satisfaction claims:

45 > 36
1 + 2 = 3
.str1 = "abc" + .str2

The predicate indicated by the > infix-operator-shorthand is officially known by its name Gt. The
third claim is asserting that the value of the variable \textit{str}1 is the concatenation of "abc" with the value of variable \textit{str}2. In addition to the \textit{=} predicate, Cymbal supports the usual inequality predicates \textit{<}, \textit{<=}, \textit{>}, and \textit{>=} which have the customary meaning for numerical subjects, which refer to lexicographic ordering for STRING and HEKA subjects, and which refer to chronological ordering for DATES.

Predicate names which do not consist of special characters like ‘\textit{=}’ must be UPLOWS. (No such restriction applies to names for C\_external Cymbal predicate tasks or imported C predicates when they are declared or defined prior to use in the Cymbal file that uses them.)

As another example of a predicate, the \textit{Matches} predicate is used to assert that a string matches a regular expression pattern string. The regular expression syntax used by Matches is that employed by egrep(1); see Chapter 7 for details. Using Matches and its abbreviated name \textit{\textasciitilde>} as follows, all of the following satisfaction claims are true:

\begin{verbatim}
"Smithson" Matches "\textasciitilde Smith"
"Smithson" Matches "\textasciitilde Smith"RE
\textasciitilde Smith \textasciitilde \textasciitilde \textasciitilde \textasciitilde [a-z]*\textasciitilde
\end{verbatim}

As indicated here, if an STR is given as the second argument to \textit{\textasciitilde>}, then the type system automatically promotes or converts that to being an RE.

The preceding sample satisfaction claims are illustrative of the following grammar productions:

\begin{verbatim}
SatClaim ::= SomeSubjects CompoundPred
     | SomeSubjects OpCond
     | TokensAsn
     | BoxAsn
     | TupleAggsAsn
     | _true_ | _false_

CompoundPred ::= SimplePred Subject ? KeywdArg *

SimplePred ::= uplow | Is_In | Equals
     | < | <= | > | >= | \textasciitilde>

Equals ::= = | Eq | Equals

KeywdArg ::= lower | preposition Subject | preposition Aggregate

SomeSubjects ::= Subject | [ SubjectSeq ]
\end{verbatim}

As the grammar indicates, the general form of a satisfaction claim uses keyword arguments. This can be illustrated by briefly discussing the \textit{Is\_In} predicate which asserts that objects are members of SETS, LISTS, and INTERVALS, e.g.,
.zz Is_In [ 2, 5, 7 ]
.zz Is_In [ 2, 5, 7 ]
.zz Is_In [ 101 -> 234 ]

*Is_In* is even more overloaded than this:

.xxx Is_In 
.sql $[ select Year from BILL where Origin = "New York" ]$
.xxx Is_In [ .yy : there_is_a SUPPLIER where( City = .yy ) ]
.xxx Is_In { .yy : there_is_a SUPPLIER where( City = .yy ) }

The first example shows how SQL queries can be integrated into Cymbal whereas the latter two illustrate the BOX concept to be discussed later in Chapter 12. As remarked above, *Is_In* is a PREDICATE which also uses keyword arguments:

.y Is_In .box_1 in_reverse_lexico_order with_sort_index si

*in_reverse_lexico_order* is a no-argument keyword and *with_sort_index* is a keyword that takes an INT-valued VARIABLE as an argument (more detail on this in Chapter 12).

Some PREDICATES take positional arguments and are invoked as illustrated by:

```cymbal
Contains[ "abcde", "bcd" ]
Is_A_Date[ "1-1-84" ]
```

Note the use of brackets, not parentheses as function calls would use. In the case of 1 or 2 positional arguments, equivalent infix notation is available:

```cymbal
"abcde" Contains "bcd"
"1-1-84" Is_A_Date
```

The constants _true_ and _false_ can appear wherever a satisfaction claim does: when they do, they are considered to be abbreviations for _true_ = _true_ and _false_ = _true_, respectively. The utility of this will be apparent when the conditionals of Chapter 6 are discussed.

Many more built-in Cymbal PREDICATES are discussed at length in Chapter 7. Also, users may add their own Cymbal and C predicates to the system in exactly the same way that user-defined functions are added (see Chapter 3).

### 5.8.2 Logical Connectives

Cymbal’s declarative statements, i.e., assertions, are built up from the satisfaction claims by using the usual logical connectives and quantifiers. So, in general, for Cymbal assertions A, B, and C,
are all Cymbal assertions where the first three are considered the same, as are the last two.

Conjunction and disjunction both group from the left. The logical operators are mentioned above in order of decreasing precedence so that negation has precedence over conjunction which has precedence over disjunction and so on. Parentheses may be used to effect different groupings. Quantifiers are explained later in this chapter.

Regarding negation, Daytona creates the appropriate negation each time it discovers "Not_" or "Does_Not_" in a predicate name so that

\[
\text{"abcde" Is_Not_A_Substr_Of "abd"
}
\]
is equivalent to

\[
! ( \text{"abcde" Is_A_Substr_Of "abd" } )
\]

5.8.3 The truth Function

Some systems consider assertions to evaluate to either _true_ or _false_. Daytona does not. An assertion is simply not the same thing as its truth value just as an object is not the same thing as its name. Cymbal users must use the truth function to map assertions to their truth values. For example, here is a false assertion illustrating the use of truth:

\[
\text{truth( 1 = 2 ) = _true_}
\]

5.8.4 Extended Predicates

Cymbal supports assertion abbreviations like:

\[
.x > 3 \& < 5
\]

This is an abbreviation for:

\[
.x > 3 \text{ and } .x < 5
\]

In effect, this abbreviation factors out a common subject from a conjunction of satisfaction claims. What the factoring leaves behind is called an extended predicate:

\[
> 3 \& < 5
\]
Here are further examples of `ExtendedPreds`:

\[
\begin{align*}
&> 2 \\
&> 2 \land < 6 \\
&= 3 \lor = .x \\
&!= .y \lor \geq .x \land \leq .z
\end{align*}
\]

These examples show how simple extended predicates like `> 2` can be negated, and’ed, and or’ed together. The reason why "\&" and "\|" are used instead of `and` and `or` is that the latter can only be used for complete assertions that don’t have their subjects missing. Regarding precedence, negation binds tighter than conjunction (\&) which binds tighter than disjunction (\|).

Here are the productions from the Cymbal grammar that characterize `ExtendedPreds`:

```plaintext
Assertion ::= SomeSubjects ExtendedPred

ExtendedPred ::= PredPhrase | ConjPredPhrase | DisjPredPhrase

DisjPredPhrase ::= ConjOr1PredPhrase \[ "|" ConjOr1PredPhrase \]+

ConjOr1PredPhrase ::= PredPhrase \[ \& \] PredPhrase \]+

ConjPredPhrase ::= PredPhrase \[ \& \] PredPhrase \]+

PredPhrase ::= CompoundPred | not CompoundPred
```

Here is a last, somewhat involved example of an extended predicate in use:

"cd" + .x != "abc" \& ! Matches "..c" \lor != "def" \& Matches "..f"

A useful feature of extended predicates is that when a `FUNCALL` or `ARRAY` element subject is factored out, it is computed only once, whereas if it were to have been left factored in, then Daytona would compute it once per occurrence.

### 5.8.5 Quantifiers

Cymbal assertions may also employ the existential and universal quantification associated with first-order symbolic logic languages. Here is a conjunction of 3 existential quantifications:

```plaintext
there_exists .x such_that( .x Is_In [1->5] and .x < 2 > 20 )
and there_does_not_exist INT .y such_that( .y = 2 and .y > 4 )
and there_exists [ .x, INT .z ] such_that( .x Is_In [1->5] and .z Is_In [1->.x] and 2 * .z = .x )
```

The rules Daytona uses in order to determine the truth values of quantified assertions (when feasible) are described in Chapter 9. For the moment, just consider these quantified assertions as being read as English statements which, in the above example, are all true. Notice that type information can
optionally be included; it is only rarely necessary since Daytona’s type inference mechanism is usually able to deduce the appropriate types for variables.

Note that in general, for assertion $A$,

$$\text{there}\_\text{exists} \ [ .x1, .x2, \ldots, .xn ] \text{ such}\_\text{that} ( A )$$

is an abbreviation for

$$\text{there}\_\text{exists} \ .x1 \text{ such}\_\text{that} ( \text{there}\_\text{exists} \ .x2 \text{ such}\_\text{that} ( \ldots ( \text{there}\_\text{exists} \ .xn \text{ such}\_\text{that} ( A ) \ldots ) ) )$$

Incidentally, the square brackets must be used if there is more than one variable but otherwise they are optional. (Also, thexi abbreviates there_exists, sthat abbreviates such_that, and ist abbreviates both is_such_that and such_that.)

By way of terminology, recall from symbolic logic that the quantified assertion

$$\text{there}\_\text{exists} \ .x1 \text{ such}\_\text{that} ( A )$$

has quantifier there_exists .x1 and scope or matrix $A$. The appearance of .x1 in the quantifier is the scoping occurrence of x1. The variables in the quantifier are only known or defined within the associated quantifier scope. For example, in the conjunction of existential quantifications that began this section, Daytona considers the x appearing in the first quantified assertion to be one variable named ‘x’ which is distinct from another variable named ‘x’ that appears solely in the third quantified assertion.

More precisely, the grammar states that the following is an existentially quantified assertion:

$$\text{existential} \ \text{SomeVblSpecs} \ \text{such}\_\text{that} \ \text{BoundedAsn}$$

where

$$\text{existential} \ ::= \ \text{there}\_\text{exists} \mid \text{there}\_\text{does}\_\text{not}\_\text{exist} \mid \text{there}\_\text{exists}\_\text{no}$$

$$\text{SomeVblSpecs} \ ::= \ \text{VblSpec} \mid [ \text{VblSpecSeq} ]$$

$$\text{VblSpec} \ ::= \ [ \text{Type} \ \text{lP} \ [ \ \text{lP} \ \text{lower} ] ]$$

$$\text{BoundedAsn} \ ::= \ ( \text{Assertion} ) \mid \text{Desc}$$

Universally quantified assertions have one of the two following and equivalent syntactic forms:

$$\text{for}\_\text{each} \ \text{SomeVblSpecs} \ \text{if}\_\text{ever} ( \text{Assertion} \ \text{then} \ \text{Assertion} )$$
$$\text{for}\_\text{each} \ \text{SomeVblSpecs} \ [ \ \text{such}\_\text{that} \ \text{BoundedAsn} \ \text{lP} \ \text{conclude} \ \text{BoundedAsn} \ )$$

Three examples are:

$$\text{for}\_\text{each} \ .y \ \text{if}\_\text{ever} ( \ .y \ \text{Is}\_\text{In} [3\rightarrow15 \ \text{by} \ 2] \ \text{then} \ .y \ \% \ 2 = 1 )$$
$$\text{for}\_\text{each} \ .y \ \text{such}\_\text{that} ( \ .y \ \text{Is}\_\text{In} [3\rightarrow15 \ \text{by} \ 2] ) \ \text{conclude} ( .y \ \% \ 2 = 1 )$$
$$\text{for}\_\text{each} \ .y \ \text{conclude} ( \text{if} \ .y \ \text{Is}\_\text{In} [3\rightarrow15 \ \text{by} \ 2] \ \text{then} \ .y \ \% \ 2 = 1 )$$

Copyright © 2013 AT&T
All Rights Reserved.
September 15, 2013
Choose whichever form seems the most natural; what they share that is most important is the notion of clearly identifying the (necessarily-finite) domain of the quantifier, as illustrated by the such that argument and the if clause in the examples above.

5.8.5.1 VblSpecs

A VblSpec is a Cymbal construction used for declaring or defining a variable and its type. The .x in

\[ \text{there_exists .x such_that} ( A ) \]

is actually a pun for

\[ \text{there_exists VBL x such_that} ( A ) \]

which states that the symbol \( x \) is the name of a VARIABLE of otherwise unspecified type. Please note that the \( .x \) in the quantifier is not thought of as being a variable dereference: quite the opposite indeed: it is the VARIABLE, not its value, that is being talked about in the quantifier. Here is a more extensive definition of the variable:

\[ \text{there_exists INT VBL x such_that} ( A ) \]

The occurrence of \( x \) in the quantifier is said to be a scoping occurrence. Any occurrences in the matrix are said to be matrix occurrences or scoped occurrences.

5.8.5.2 Free And Bound Variables And Occurrences Thereof

Quantifiers give rise to the very important concept of free and bound variables. A non-scoping occurrence of a variable is bound in a logic formula \( F \) if the occurrence appears in the scope/matrix of some quantification in \( F \) whose quantifier contains that variable. Otherwise the occurrence is free. A variable is free in \( A \) if it has a free occurrence; a variable is bound in \( A \) if it has a bound occurrence. Clearly, if a variable has a bound occurrence in a given scope, then all of its occurrences in that scope are bound.

Consider the following formula:

\[
\text{for_each [ INT VBL x, .y ]}
\]

\[
\text{if ever} ( .x = 1 \text{ and } .y = .x \text{ then } .y > .z )
\]

All occurrences of \( x \) and \( y \) are bound whereas the occurrence of \( z \) is free.

In the following assertion, all \( x \) and \( y \) occurrences are bound and all \( w \) and \( z \) occurrences are free.

\[
\text{for_each .y if ever} ( .y = .w \text{ then}
\]

\[
\text{there_exists .x such_that} ( .x = .y \text{ and } .w > .z )
\]

Above, in the existential quantification, \( y \) is also free.

An assertion is open if it contains some free variable occurrences; otherwise, it is closed. The definition of closed is extended to include those assertions whose only free variable occurrences are for outside procedural variables since those are considered to be constants in so far as the assertion is concerned.
5.8.5.3 Variable Scoping

One way or another, all of the variables in a valid Cymbal program must be scoped, meaning that each must have a single definition as part of some syntactic construct that encompasses all occurrences of that variable. Distinct variables with mutually exclusive scopes may have the same name as illustrated below by two different variables both named x:

\[
\begin{align*}
\text{there}_\text{exists} \ \cdot x \ such\_that( \cdot x \ \text{Is}_\text{In} \ [1\rightarrow 5] \ \text{and} \ .x > 0 ) \\
\text{and there}_\text{exists} \ .x \ such\_that( .x \ \text{Is}_\text{In} \ [-1.0 \rightarrow -5.0 \ \text{by} \ -1.0] \ \text{and} \ .x < 0.0 )
\end{align*}
\]

Note how the first x has type INT and the second has type FLT; this permissible because they are different VBLs! It is a theorem from logic that bound variables can always be renamed, leaving an equivalent assertion, so long as the new names had not already appeared in the assertion (the idea being to prevent coincidental binding of other variable occurrences that previously had the ‘new’ name). So, an equivalent form of the above assertion is:

\[
\begin{align*}
\text{there}_\text{exists} \ .x_1 \ such\_that( .x_1 \ \text{Is}_\text{In} \ [1\rightarrow 5] \ \text{and} \ .x_1 > 0 ) \\
\text{and there}_\text{exists} \ .x_2 \ such\_that( .x_2 \ \text{Is}_\text{In} \ [-1.0 \rightarrow -5.0 \ \text{by} \ -1.0] \ \text{and} \ .x_2 < 0.0 )
\end{align*}
\]

In fact, when Daytona processes assertions, it renames variables so that distinct variables have distinct names. It does this by appending ‘_sCPn’ to the base variable name for integer n, as in x_sCP2. User variables which have the same name but are considered by the system to be distinct variables with distinct and separate scopes are called homonyms. The system’s disambiguation of homonyms by appending _sCP suffixes to distinguish them may not be what the user expected. When the -VHC (variable-homonym-checks) option is given to Tracy, Tracy will print out for each task, all of the homonyms it has identified.

When nested scopes make use of variables with the same names, the inner one takes precedence over the outer one when it can, as for example,

\[
\begin{align*}
\text{there}_\text{exists} \ \text{VBL} \ x \ such\_that( .x \ \text{Is}_\text{In} \ [1\rightarrow4] \\
\text{and there}_\text{exists} \ .x \ such\_that( .x \ \text{Is}_\text{In} \ [1\rightarrow4] \ \text{and} \ .x > 1 ) \ \text{and} \ .x > -1)
\end{align*}
\]

being equivalent to

\[
\begin{align*}
\text{there}_\text{exists} \ \text{VBL} \ x \ such\_that( .x \ \text{Is}_\text{In} \ [1\rightarrow4] \\
\text{and there}_\text{exists} \ .x_{sCP2} \ such\_that( .x_{sCP2} \ \text{Is}_\text{In} \ [1\rightarrow4] \ \text{and} \ .x_{sCP2} > 1 ) \ \text{and} \ .x > -1)
\end{align*}
\]

A variable occurrence is explicitly scoped if one can identify a scoping occurrence of the VBL whose scope contains the occurrence. For explicitly scoped VBLs, Daytona uses the following procedure to determine which VBLs are homonyms and what their corresponding scopes are. For each non-scoping occurrence of a VBL name (i.e., a use) in an assertion, identify the smallest explicit scope containing it, if any. (If there is no such explicit scope, see the next section on implicit quantification.) The resulting scopes are the scopes of the homonyms and the corresponding VBLs and their occurrences within their scopes are then given uniquely identifying names via _sCP suffixes. In the literature, this imposing of unique names for VBLs in logic formulas, i.e., the elimination of homonyms,
is called *rectification*.

### 5.8.6 Implicit Quantification

Each variable appearing in a valid Cymbal program must have a scope, i.e., a definite region of the program in which it is defined and meaningful. Existential and universal quantification provide two ways among several to scope variables. (The others, which exist in the procedural dialect, consist of the *for_each_time* VBLs, imports/exports/locals sections for fpps, and likewise for certain do-groups.) In many cases, there are so many variables in a Cymbal query that it is a real nuisance for the Cymbal user to have to give explicit scopes (and types) to all of them. Fortunately, Cymbal users may employ both type inference and certain quantifier abbreviations to make a good deal of scoping implicit. The two quantifier abbreviations are *somehow* and *if ever* with syntax:

\[
\text{Assertion} \mid \text{somehow } \text{BoundedAsn} \\
\text{if ever}( \text{Assertion} \text{ then } \text{Assertion} )
\]

For any given assertion, any non-scoping variable occurrence which is not explicitly scoped becomes existentially or universally quantified according to the following procedure: Consider all the occurrences of a VBL name that are implicitly scoped. For each one, find the smallest implicit scope (i.e., *somehow* or *if ever*) or explicit scope that contains it. Then, for this set of smallest scopes, repeatedly take the union of any intersecting pair, until all remaining such pairs have no intersection. (When Cymbal scopes intersect, one is contained in the other, i.e., one is nested in the other.) The scopes remaining are the scopes of the homonyms. After rectification, the scope for any occurrence of any of the VBL homonyms is now known. For example,

\[
\text{somehow( .x = 5 )}
\]

is equivalent to

\[
\text{there exists .x such that( .x = 5 )}
\]

and

\[
\text{if ever( .y = .w then } \\
\text{somehow( .x = .y and .w > .z )})
\]

is equivalent to

\[
\text{for each [ .y, .w ] if ever( .y = .w then } \\
\text{there exists [ .x, .z ] such that( } \\
\text{.x = .y and .w > .z )})
\]

w is an *outside* VBL relative to the existential quantification because, by way of definition, a variable v is an *outside variable* in a quantification Q if v occurs free in Q and if its scoping occurrence is outside of Q; in other words, that scoping occurrence does not identify it as one of Q’s scoped VBLs. In the context of assertions, this means that v is scoped in a quantifier or OPCOND VBL LIST (see Chapter 9) that is outside of Q; furthermore, any procedural VBL appearing in Q is clearly outside with respect to Q because its scope necessarily includes Q. Consider:
there_exists [ .w, .y, .z ] such_that( .y = 4 and
there_exists [ .x ] such_that(
    .x = .y+1 and .w = .x+2 and .z = 2 * .w )
and .z > .y )

Both y and z are outside VBLs for the inner quantification as is w even though all of w’s uses occur within that inner quantification.

Daytona ignores empty somehows, i.e., ones that don’t catch any variables, and empty if_evers are considered to be if-thens. For example,

there_exists [ .x, .y ] such_that(
    .x Is_In [1->3] and .y Is_In [.x->.x+2]
    and
    somehow(
        if_ever( .x > .y**2 then .x > .y )
    )
)

is equivalent to

there_exists [ .x, .y ] such_that(
    .x Is_In [1->3] and .y Is_In [.x->.x+2]
    and
    if( .x > .y**2) then (.x > .y )
)

Daytona silently inserts somehows (if an existential quantifier is not already there) around various assertions in the Cymbal queries it processes. (Since somehows are implicit quantifiers, this implicit insertion of somehows amounts to implicit implicit quantification!) For completeness’ sake, the following is a list of all such assertions, even though many of these constructs will be introduced later in this manual.

- the each_time assertion for Display
- the is_such_that assertion for for_each_time
- the each_time assertion for aggregate functions
- the assertion for intensional box formers
- any assertion being negated
- any disjunct
- the antecedent assertion in an if-then or if-then-else assertion
- the consequent component of an if-then assertion
• the consequent and else-consequent component of an *if-then-else* assertion
• the assertion components to an *if-and-only-if* assertion
• the consequent assertion in a *for_each* or *if-ever* assertion
• the assertion argument to the *truth* function
• the assertion argument to an OPCOND or OPCOFUN
• the assertion identified by a *parallelizing* keyword
• any assertion argument to a macro or path PREDICATE definition
• the assertion argument to *Change* command
• the assertion for *when* commands
• the assertion for *while* and *until* commands

Note that any explicit *there_exists* for its given VBLs is ready, willing, and able to become the scope for other implicitly scoped VBLs in the same manner that *somehow* does.

Consider this illustrative query (somehow.2.Q):

```cymbal
do Display each[ .a ]
each_time(  
  .d = 2  
  and .a = 1 | 2*.d  
  and  
  ( (.b = 4 and .b > .a)  
  or (.b = 5 and .b > 2*.a)  
  or (.a = 6 and .c = 7 and .c > .a)  
  or (.a < 3)  
  )  
  );
```

By applying the rules and conventions above, one determines that it is equivalent to:

```cymbal
do Display each[ .a ]
each_time( there_exists .d such_that(  
  .d = 2  
  and (.a = 1 or .a = 2*.d)  
  and  
  ( (there_exists .b such_that( .b = 4 and .b > .a))  
  or (there_exists .b such_that( .b = 5 and .b > 2*.a))  
  or (there_exists .c such_that( .a = 6 and .c = 7 and .c > .a ))  
  or (somehow(.a < 3)) // empty somehow keyword will just disappear!  
  )  
  );
```

Copyright © 2013 AT&T
All Rights Reserved.
September 15, 2013
Note that there are two distinct VBLs with the name ‘b’. In processing this query, Daytona literally rewrites the second quantification on \( b \) by substituting \( b_{sCP2} \) for all occurrences of \( b \).

### 5.8.7 Cymbal Descriptions

Generally speaking, Cymbal descriptions are special kinds of assertions that Cymbal uses to talk about the object records in a database. The underlying context here is that the Daytona user has an interest in one or more classes of objects where each class is characterized by some membership criterion. For each given object class, there is an associated set of attributes (i.e., FIELDS) where each attribute maps each object in the class either to no value or to some value consisting of some typed object (such as an INT or a BUNCH of DATES). So, for example, the Age attribute for EMPLOYEE objects is a mapping from the set of EMPLOYEES to the set of INTEGER ages that could be a partial function in that for some EMPLOYEES, there may not exist a corresponding Age value (possibly because the person refused to give it). (In mathematical terms, an attribute is a partial function that maps an object class into another set of objects (called the value set or the range of the attribute.).

Recall from Chapter 3 that the object record for such an object is the corresponding set of attribute-value pairs (for those pairs that exist). All of the object records for a given object class are grouped together into the corresponding record class. As discussed in Chapter 3, a record class is implemented by storing the (DC) data file records corresponding to its object records into one or more (UNIX) files, the latter occurring when horizontal partitioning is used. Consequently, a record class is the Daytona-equivalent of a relational table.

A Cymbal description is a primitive assertion, in its simplest form similar to a satisfaction claim claiming that a TUPLE of values satisfies a predicate, that describes one or more object records that have been implemented by stored user data file records. The latter qualification is important: Cymbal descriptions are used to describe information stored in some persistent way, as would occur for information stored on disk or on tape; descriptions are not used to describe transient information (except when they are descriptions defined by a view!). Here is one of the simplest possible Cymbal descriptions:

```plaintext
there_is_an ORDER where(
    Number = 787
    and Supp_Nbr = 321
    and Date_Placed = 4-1-34
    and Quantity = 439
)
```

This merely asserts that there is an object record in the ORDER record class which has a Number attribute with value 787 and a Supp_Nbr attribute with value 321 and . . . and a Quantity attribute with value 439. Here, ORDER is considered to be the record class consisting of all order records in the database. Incidentally, it is useful to note that `there_is_a`, `there_is_an`, `there_isa` and `tisa` are considered equivalent and may be used interchangeably, thus enabling an appropriate choice to be made depending on whether the record class name begins with a vowel or not. In order to improve readability, attribute (i.e., field) names must be UPLOWS.

Each description has exactly one purpose which is to make an assertion that describes one or more
object records of the specified record class. In doing this, of course, it makes reference to the values of attributes in those records and frequently, places restrictions on the values of certain of those attributes.

The syntax for the simple ORDER description above is specified by the following simplified/basic grammar:

\[
\begin{align*}
\text{Desc} & ::= \text{somedesc} \quad \text{class}\_\text{name} \quad \text{where} ( \text{Note} \quad [ \quad \text{and} \quad \text{Note} \quad ]^* ) \\
\text{somedesc} & ::= \text{there\_is\_a} \quad | \quad \text{there\_is\_an} \quad | \quad \text{there\_isa} \quad | \quad \text{tisa} \\
\text{Note} & ::= \text{Attribute} \quad \text{ExtendedPred} \\
\end{align*}
\]

"there\_is\_a" is a member of the somedesc syntactic class; "ORDER", since it is the UPPER name of a generic member of an object class, is a class\_name; and Number = 787 is an example of a Note.

Semantically speaking, in its simplest form, a description of an object begins by stating what its class is and then continues by stating what the values of various attributes are for that object.

The Cymbal grammar also gives a number of syntactic variants and extensions to descriptions, some of which will be discussed later in this chapter. But the point at the moment is that the syntax is highly restrictive: for example, the user cannot just toss an or into the middle of a description or an if...then or a quantifier. Descriptions are fundamentally just (quasi-)existentially quantified conjunctive assertions about FIELDS. This means that after the there\_isa portion, the body of the description looks like this:

\[
\text{where( Field}_1 \quad ... \quad \text{and Field}_2 \quad ... \quad \text{and ... and Field}_k \quad ... )
\]

Having said this, it should also be said that the restricted syntactic form for descriptions in no way reduces the range of assertions that can be made about object records. In the rare circumstance when the full description syntax is not sufficient, it is always sufficient to just associate all desired attribute values with variable values in the description and then after that description, add as a conjunct any desired assertion about those variable values. Consider, for example,

\[
\text{there\_is\_a SUPPLIER where(}
\text{Name} = .\text{supplier}
\text{and City} = .\text{location} )
\text{and (.location} \neq \text"Seattle" \text{or substr( .location, 1, 3 )} = \text"New" )
\]

Here the description is asserting the existence of a SUPPLIER record where the Name attribute value is the value of the variable supplier and the City attribute value is the value of the variable location. This is conjoined with a disjunctive constraint on the value of the location variable.

To some it may seem like:
there is an ORDER where ( 
Number = 787 
and Supp_Nbr = 321 
and Date_Placed = 4-1-34 
and Quantity = 439 
)

is an object record. But, while it may seem to be isomorphic to an object record (observe though that it might not be mentioning all of the attributes), it is nonetheless not an object record. In general, a description describes one or more object records in a way analogous to the way testimony at a trial describes one or more people. Clearly, trial testimony is not the same as the people it describes. Anyway, here is a description that describes several ORDERS (note the multiple occurrences of Quantity):

there is an ORDER where ( 
Number Is_In [ 1 -> 10 ] 
and Quantity > 1000 
and Quantity < 5000 
and Quantity = .qty )

and .qty % 10 = 0

As will be seen shortly, this somewhat wordy basic description can be equivalently written using more sophisticated syntax as:

there is an ORDER where ( 
Number Is_In [ 1 -> 10 ] 
and Quantity = .qty which_is > 1000 & < 5000 )

and .qty % 10 = 0

The description syntax provides a much more flexible way of expressing basic assertions about objects than the fixed-arity, positional syntax used in those logic database languages based on Prolog. The description above of the ORDER whose Number is 787 would be represented in such a language by:

ORDER[ 787, 321, 4-1-34, 439 ]

Note that unless the attribute names are somehow coded into the table name, there is no local syntactic clue to say what the meaning is of the arguments to the predicate ORDER. Furthermore, all of the attribute positions must be occupied with something and the attribute values must appear in a certain order. And, of course, Cymbal description syntax goes far beyond just describing one record.

As a convenience, there_is_no may be used to assert that there are no objects that match a given description. Hence, the following two assertions are equivalent:
there_is_no ORDER where(
    Number > 787
    and Supp_Nbr = 321
)

/***************************/

its_not_so_that(
    there_is_an ORDER where(
        Number > 787
        and Supp_Nbr = 321
    )
)

5.8.7.1 Special Kinds Of Simple Notes

The following example illustrates how Cymbal descriptions can express assertions about the values of LIST/SET-valued attributes as well as about the presence or absence of attribute values.

there_is_a PERSON where(
    Phone_Nbr Is _absent_
    and the Street_Address = "1234 Maple Ave"
    and the Salary > 50000 & < 60000
    and one_of_the Children = "Tom"
    and Children = .kids_set
    and Autos = { "Grand Am", "Corvette" }
)

The relevant portions of the grammar are:

$\text{Note} ::= \begin{bmatrix}
\text{noteprep} & | & \text{Attribute} & \text{ExtendedPred} \\
\text{noteprep} & | & \text{Attribute} & \text{Is} & \begin{bmatrix}
\text{absent}_- & | & \text{present}_- \\
\text{uplow} & | & \text{one_of_the} \\
\end{bmatrix} \\
\end{bmatrix} \begin{bmatrix}
\text{noteprep} & | & \text{Attribute} & \text{Is} & \begin{bmatrix}
\text{absent}_- & | & \text{present}_- \\
\text{uplow} & | & \text{one_of_the} \\
\end{bmatrix} \\
\end{bmatrix}$

The special constructs \text{Is _absent}_- and \text{Is _present}_- are used to assert the absence or presence of a value for the corresponding attribute. They provide the most basic of missing value control; how to achieve total control over missing values will be discussed later in Chapter 13.

If no \text{noteprep} is used, then it defaults to \text{the} . \text{one_of_the} is used only with LIST/SET-valued attributes to make an assertion about some element of the corresponding attribute’s set. In order to make an assertion about the set itself, one would just leave out the \text{one_of_the} keyword (or use \text{the}). In this latter case, the system currently only supports saying that the (LIST/SET) value of the FIELD is equal to the value of a non-ground VBL, i.e., a defining or generating occurrence for the VBL. In other words, the system does not yet support saying that the value of the FIELD is equal to a LIST or SET,
i.e., a test that two BOXES are the same. In fact, in general, the system does not yet support such BOX equality tests written as an equality satisfaction claim of two aggregate objects. Fortunately, the ability to define a new VBL to have the value of a LIST/SET FIELD is quite useful, especially when updating RECORDS with LIST/SET-valued FIELDS. And there are scalar-based ways of testing to see that two boxes are the same.

An object record is well-designed if it gives the values (if any) for all attributes of interest and does not contain descriptive information about any other objects. This implies that the object being described by an object record does indeed have a special role in that it is the raison d’etre and focal point for that object record. If an object has a Name attribute then Cymbal has some special, although optional, description syntax that the user can use in order to improve the readability of their queries, as in:

```cymbal
there_is_a PERSON named "Tom Arnold" where(
    Age = 31
    and Salary = 45000
)
```

This description is interchangeable with:

```cymbal
there_is_a PERSON where(
    Name = "Tom Arnold"
    and Age = 31
    and Salary = 45000
)
```

Any Name attribute must have either STR or LIT values. The readability of Cymbal code is increased if the reader can easily determine the name, if any, of objects being described by some description. In other words, the Name of an object is such a distinguished and useful attribute that it is well worth allowing it to be referenced in a syntactically distinguished way.

Here is the relevant syntax:

```cymbal
Desc ::= some(desc class name IdNote P where( Note and Note ) P ;

IdNote ::= named Subject | meaning Subject ;
```

Analogous to the use of named for the Name attribute, Cymbal allows the use of meaning for the Self attribute, if any. The value of the Self attribute is the object itself that is being described by the description, not its name. Self values are frequently THINGs. Distinguishing between objects and their names may seem like philosophical hair-splitting to some but it has its uses, nonetheless. The named abbreviation will probably be more heavily used by the Daytona user community for the time being than the meaning abbreviation but the user should be aware of the meaning abbreviation because Daytona itself uses it on various infrequent occasions.
5.8.7.2 Further Note Extensions

One of the advantages of descriptions is that they allow the user to syntactically gather together in one place everything they want to say about one or more objects from a given record class. Cymbal provides two extensions to Notes that enable the user to syntactically gather together in one place everything they want to say about one attribute for an object, as illustrated by:

```cymbal
there_is_an ORDER where(
    Quantity = .qty where( .std_qty = 100 * .qty )
    and Date_Placed = .dp which_is > "1-1-84" & < "1-9-86"
    and Date_Recd > .dr which_is > "1-1-84"
)
```

The Quantity note serves to define a new variable `std_qty`. The Date_Placed note serves to get the Date_Placed attribute value into the `dp` variable under the condition that it lie in a certain date range. (This all assumes that `std_qty` and `dp` make their first substantive appearance in the query here, i.e., they are not defined-on-first-use previously (see Chapter 9).) The Date_Recd example assumes that `dr` has already been defined-on-first-use previously and it specifies that `dr`'s value lies between the Date_Recd attribute value and `"1-1-84"`. **Please note that which_is refers to .dr, not Date_Recd!** The corresponding syntax is:

```
Note ::= [ noteprep ? ] AttributeExpr ExtendedPred [ where BoundedAsn ]
where ::= where | for_which
that ::= that | that_is
       | which | which_is
       | who | who_is
       | one_which | one_which_is
AttributeExpr ::= Attribute
                | AttributeExpr Tuple
                | AttributeExpr # IntPosition
                | AttributeExpr . StructMbr
```

Cymbal can be written without these abbreviations by using a lot more ands typically. The benefit of the abbreviations is that they add syntactic variety and readability in addition to enabling related statements to be grouped together. The discussion of descriptions continues in Chapter 13.
6. Procedural Cymbal

Procedural Cymbal is a fairly conventional programming language which is enhanced by its synergy with declarative Cymbal and SQL and which is further augmented by a variety of powerful system built-in functions, predicates, and procedures. This chapter discusses in turn Cymbal’s assignments, conditionals, loops, branches, procedures, tasks, declarations/definitions, and user-defined extension capabilities. Where appropriate, subsequent examples of Cymbal usage will be augmented by the relevant grammar productions taken from the Cymbal grammar appendix; please read the beginning of this appendix for a definition of the grammar symbols employed in this chapter. Modest declarative constructs will be used in this chapter to illustrate the depth of integration between procedural and declarative Cymbal; these constructs were introduced in Chapter 2 and will be discussed in detail in Chapter 9.

6.1 Assignments

The simplest procedural command is the set command which specifies that a variable gets a new value. Here are some examples:

```cymbal
set .x = 1
set .x = .x + 1
set .x++
set .x = .x - 1
set .x--
set .x -= 1
set .ss = "abc"
set .ss += "def"
```

The first assignment above should be read:

set the value of x equal to 1

Please note that "=" always means equality in Cymbal; it never means "assignment": set means "assignment".

The ++ form and -= form are clearly derived from C; for those not familiar with C, the "++" assignment has exactly the same effect as the one preceding it. The 3 subtraction assignments also have exactly the same effect as each other. Any of the arithmetic operators +, *, and / can also be used instead of – to form analogs to the -= form above. The last assignment involving ss illustrates the overloading of the + function. In particular, when used with a VBL whose type is one of STR(*)|LIT(*)|RE(*)|HEKSTR(*)|CMD|SHELLP|SAFE_STR, the RHS is appended to the contents of the LHS VBL in a particularly efficient way that avoids the garbage collection system.

Relative to the composite classes, Cymbal supports:
By using manifest TUPLES, multiple-assignments-in-one may be realized.

\[
\begin{align*}
\text{set} & \ [ \ .x, \ .y, \ .z ] = [ 2, .q, .q**3 ]; \\
\text{set} & \ [ \ .x, \ .y, \ .z ] = [0]; \\
\text{set} & \ [ \ .x1, \ .x2, \ .x3, \ .x4, \ .x5, \ .x6 ] = [ 1, 2, 1000 ];
\end{align*}
\]

The first assignment above assigns values to the variables component-wise. The effect of the second assignment is to give all 3 variables the value of 0. As illustrated by the second and third assignments, when the RHS TUPLE contains fewer elements than the LHS target TUPLE, the last element of the RHS TUPLE is used for the remaining assignments.

Here are three special LIST assignments that are described in complete detail in succeeding chapters.

\[
\begin{align*}
\text{set} & \ [ \ .x, \ .y ] = \text{tokens( for "abcXXXdef" matching "\(\ldots\)XXX\(\ldots\)" )); \\
\text{set} & \ [ \ .a, .b, .c ] = \text{read( from _cmd_line_ )); \\
\text{set} & \ [ \ .nbr, .mean, .sig ]
\quad = \text{aggregates(}
\qquad \text{of [ count(), avg( over .i ), stdev( over .i ) ]}
\qquad \text{each_time ( .i Is_In [ 1 -> 50 ] ))};
\end{align*}
\]

The tokens assignment capability enables the user to do regular expression lexical analysis on strings and to return the desired tokens; specifically, in this case, \(x\) will get the value "abc" and \(y\) will get the value "def" or in other words, after the assignment is executed, the following assertion will be true:

\(.x = \text{"abc" and .y = \text{"def"} \}

The read assignment shows how to read 3 values from the UNIX command line and assign them to different variables in turn.

The aggregates assignment capability enables the computation of tuples of aggregate functions defined declaratively over all variable values which make some assertion true; in this case, \( .nbr \) will become 50, \( .mean \) will become 25.5, which is the average of the first 50 positive integers, and \( .sig \) will become 14.577, which is their standard deviation.

Note that conventional ARRAYS (i.e., not associative) whose sizes are known at compile time may be assigned to from TUPLES and vice-versa:
{ local: INT ARRAY[ 5, 2 ] .bara
  set .term = -9;
  set .bara = [ 1, 2, 3, .term-90 ];
  do Write_Words( .bara );
}

Note that the last element of the TUPLE is considered to be replicated for assignment to the trailing elements of the ARRAY. Here is the converse case:

local: INT ARRAY[ 4 ] .ara1
set .ara1 = stokens( for "9:8:7:6" upto ":" );
set [ .a, .b, .c, .d ] = .ara1;

6.1.1 The Assignment Function set

Cymbal also offers by means of the set function the ability to assign a value to a variable in the middle of an expression and to use that value in the expression. In the scenario described here, the set function takes two arguments, the first being a scalar VBL and the second a scalar value and returns its second argument after assigning the second argument to be the value of the first argument. Here is an example:

set .x = 1;
loop { do Write_Line( .x ); }
while( set(x,.x+1) < 10 );

A similar capability exists in C. The second scenario for using the set function happens when it is used to check for the existence of a dynara element (see Chapter 11 and 21). Nota bene: since the use of this set() function in an assertion violates the declarative nature of assertions, its use is severely restricted: either the satclaim using it comprises the entire assertion or the assertion consists entirely of a conjunction whose every satclaim conjunct is using the set() function.

6.1.2 Substitution of LHS in RHS

As a convenience, Cymbal offers the $ substitution facility whereby any appearance of $ on the right-hand-side of an assignment is taken to be an appearance of the left-hand-side. This is a generalization of the ++ and += shorthands. It can be very helpful when updating TUPLES, as illustrated by:

local: TUPLE[ INT, BITSEQ, DATE, STR(*) ] .tu
  = [ 74, ^111100^B, ^8/30/00^, "IBM"
set .tu = [ $#1+1, ?, $#3+1, $#4+"X" ];
do Write_Words(.tu);

The third line is equivalent to the simple rewriting:

set .tu = [ .tu#1+1, ?, .tu#3+1, .tu#4+"X" ];

The # is used to select the indicate component or element of the TUPLE, as in .tu#1 being the first component of the TUPLE .tu. The ? in the second position of the RHS TUPLE indicates that nothing
is to be done with the second component of .tu.

The $ substitution is always efficient and in some cases, such as when updating array elements, it is more efficient than anything the user can write without using VBL VBLS (i.e., pointers).

Also, as used below, it not only saves keystrokes but it is also clever enough to evaluate the myfun function call exactly once, instead of the apparent four times.

\[
\text{set .ara[ myfun(.z) ] = [ $#1 - 1, $#2 + 2, 0, $#3 + 1 ];}
\]

The assignments in a TUPLE $-substitution happen from left to right as seen by:

\[
\text{set .a[1] = [ .a[1]#1 + .a[2]#2, .a[1]#1 + .a[2]#2 ];}
\]

being implemented by:

\[
\begin{align*}
\text{set .qq = a[1];} \\
\text{set ..qq#1 = ..qq#1 + ..qq#2;} \\
\text{set ..qq#2 = ..qq#1 + ..qq#2;}
\end{align*}
\]

This means that the two appearances of .a[1]#1 on the right-hand-side of the original assignment have in effect different values.

6.1.3 Otherwise: Assignments That Can Fail

As it happens, there are some assignments that can fail. These are discussed here in the same section as the simple assignments even though this discussion assumes information from later in this chapter and elsewhere. The problem is that there are just some (procedural) functions that can fail to produce a useful value to return -- and may in fact fail in manifold ways. For example, the function might need to query some other computer in order to calculate its answer: if that computer is down, then the answer cannot be computed -- and the calling program would like to know that and know the reason why. In general, while there are some errors that can only lead to termination, there may indeed be several others that could in principle be constructively handled by user program logic. Furthermore, it may be inconvenient to code up the return values so that they may sometimes represent normal values and at other times, error exit statuses. (For example, suppose a function returns the last DATE of customer interaction: even if there is just one way for this function to fail, it is still a hack to code that failure mode in a DATE.)

Daytona has several functions of this kind, including: new_channel, read, shell_exec, next_io_ready_tendril, next_waited_for_tendril, get_tickets. What they have in common is that their calculations all depend on circumstances outside the address space of the program, thus exposing them to a number of error situations the program might like to handle with its own logic.

In Daytona’s case, these functions communicate their completion status to the caller by means of assigning an INT completion code to a global variable which is named by appending ‘_call_status’ to the function name. An example would be the VBL read_call_status. Various call statuses are represented by symbolic constants such as the one they all have in common, namely, _worked_. So, a typical use of a function of this kind would be:
loop {
    set [ .x ] = read_line( from .file_chan );
    when( .read_call_status != _worked_ ) {
        when( .read_call_status = _instant_eoc_ ) break;
        else { do Exclaim_Line( "error: unexpected error in reading next line" );
            do Exit( 3 );
        }
    }
}

This case-by-case handling gets to be pretty cumbersome to write pretty quickly. For that reason, Cymbal offers the abbreviation provided by the otherwise construct. In its simplest form, it would be used as in:

set .mychan = new_channel( for "myfile" ) otherwise do Exit( 3 );

This is essentially an abbreviation for:

set .mychan = new_channel( for "myfile" );
when( .new_channel_call_status != _worked_ ) {
    do Exit( 3 );
}

However, Daytona does even more for the user in these situations because when it notices that an Exit call has been placed within an otherwise assertion, it arranges for the Exit call to print out additional information identifying the function that failed, the call status it has set, and the location of the function call in the program. Here is what is produced when the above otherwise assignment for new_channel fails:

error: new_channel call status '_fopen_failed_' at Cymbal line 1 for fpp Begin.

error: R: program with pid 12111 terminating at Tue Sep 26 07:23:16 EDT 2000 with status 3

error: the program was invoked using:
    'R +D'

Even better, Exit can take an optional with_msg keyword argument:

set .mychan = new_channel( for "myfile" )
    otherwise { with_msg "failed to get a new CHAN for myfile" do Exit( 3 ); }

The consequence is to include that message in the output to _stderr_: 
error: new_channel call status `_fopen_failed_' at Cymbal line 2 for fpp Begin.

failed to get a new CHAN for myfile

error: R: program with pid 12403 terminating at Tue Sep 26 07:28:14 EDT 2000 with status 3

error: the program was invoked using:
  `R +D'

The otherwise_switch construct makes it easier to give special attention to one or more call statuses. Here's how to rewrite the read loop from above.

```cymbal
loop {
    set [ .x ] = read_line( from .file_chan )
    otherwise_switch {
        case( = _instant_eoc_ ){ break; }
        else { with_msg "error: unexpected error in reading next line"
            do Exit( 3 ); }
    }
}
```

This is regarded as being an abbreviation for:

```cymbal
loop {
    set [ .x ] = read_line( from .file_chan );
    when( .read_call_status != _worked_ )
    {
        switch( .read_call_status ){
            case( = _instant_eoc_ ){ break; }
            else { with_msg "error: unexpected error in reading next line"
                do Exit( 3 ); }
        }
    }
}
```

In general, the cases of the otherwise_switch should be thought of as cases whose subject term is the value of the appropriate call_status VBL.

The otherwise constructs are supported for precisely those functions in sys.env.cy whose imports include the keyword otherwise_ok.

6.1.4 Assignment Grammar

For the record, here are the grammar productions associated with assignment:
The last 4 Assignment productions are described in detail elsewhere in this manual.

The use of a skolem or ? as the target of an assignment provides a convenient syntax for executing a function for its side-effects while discarding its return value (for whatever reason). For example,

```
set ? = shell_exec( .my_cmd ) otherwise do Exit(3);
```

6.2 Semicolons

Semicolons are used in procedural Cymbal either to terminate commands or to separate them: the parser allows either or both conventions to be used, even within the same program. When semicolons are used as separators, the last command in a sequence of commands does not end with a semicolon. The philosophy here is that a separator semicolon is an operator that joins two actions together in the sequence they will be executed. In C, semicolons are used almost exclusively as terminators except in "for" statements and after action-brace-groups (although semicolons do appear after structure-definition-brace-groups). The most important message here is that the C semicolon convention is also honored in Cymbal. So also is the separator convention where semicolons may or may not follow Cymbal brace-groups (described next). Just pick a convention and the Cymbal parser will tend to be forgiving of any oversights.
6.3 Grouping

A *do-group* consists of a sequence of commands that is enclosed by braces and optionally preceded by a *do*. Such a group is then considered to have the same status as any single command and can therefore be used wherever a single command can be used. Here are two examples:

```cymbal
do {
    local: INT .x
    set .x = 3;
    {
        imports: INT VBL y
        set .y = 4;
        set .z = 5;
    }
}
```

Here is the relevant grammar production:

<table>
<thead>
<tr>
<th>Do</th>
<th>::=</th>
<th>I do</th>
<th>L?</th>
<th>BraceExpo</th>
</tr>
</thead>
<tbody>
<tr>
<td>SqlStmt</td>
<td></td>
<td></td>
<td>$[ SqlProgram ]$</td>
<td></td>
</tr>
<tr>
<td>;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A *BraceExpo* is a sequence of commands enclosed in braces. (It may also begin with some imports and local variable definitions as explained later.)

6.4 Conditionals

There are 2 types of conditionals: the *when-else* conditional and the *switch*. Here is a do-group containing a *when-else* conditional:

```cymbal
do {
    set .x = 3;
    when( .x < 0 ) do { set .y = -1 }
    else when( .x = 0 ) set .y = 0;
    else set .y = 1
}
```

The Cymbal *when-else* command has the same function as the C *if* command in that when the *when* assertion is true, then its command is executed, else the *else* command is executed.

Note that one of the *when* actions is a do-group (the *do* not being required) and the other one is just a simple assignment. Note also the varied use of semicolons.

Here is the grammar production for the *when-else* command:
When executing a *when-else* command if the first *Assertion* is true, then its associated *Command* is executed, else the next *when-else* command (if any) is executed; if all *Assertions* are false and a default *Command* is present, then it is executed.

Absolutely any declarative Cymbal assertion, no matter how sophisticated in its use of logic, can be placed within parentheses after a *when* as long as Daytona can compute its truth value. (Note, as a convention, parentheses are used to group assertions and braces are used to group actions.)

The reader may be asking, "Why *when*? Why not *if*?" Since Cymbal is both a programming language and a logic language, care needs to be taken to distinguish between dynamic, time-related procedural constructs and static, timeless declarative constructs. In symbolic logic,

\[ \text{if } \text{Assertion}_A \text{ then } \text{Assertion}_B \]

is precisely the same as

\[ \text{not } \text{Assertion}_A \text{ or } \text{Assertion}_B \]

This in turn is quite a bit different from the procedural "if" which is essentially

\[ \text{when( Assertion}_A \text{ ) do (Commands )} \]

So, "if" is in fact an overloaded word and in order to keep both meanings distinctly named, *when* with its time and state connotations is used in the procedural setting. (As a concession to conventional usage, the Cymbal parser will accept *if* when *when* is appropriate; such usage may make it more difficult for the user to keep these two concepts distinct their minds which is important because they really are distinct.)

Note that the *Commands* in a *when-else* command cannot be *when-else* commands themselves. This implies that there is no *if-else* ambiguity to resolve like there is in other programming languages because, for example, the following is illegal syntax in Cymbal:

\[ \text{when}( \text{.j >= 3) when( .j = 3) do { set .i=3 } else set .i=2; } \]

The association of the *else* with which *when* must be done explicitly through brace-groups into one of the following:

\[ \text{when}( \text{.j >= 3) { when( .j = 3) do { set .i=3 } else set .i=2; } \}

\[ \text{when( .j >= 3) { when( .j = 3) do { set .i=3 } ) else set .i=2; } \]

As a special convenience, since the Cymbal parser considers a BOOLEAN appearing where an assertion is expected to be an abbreviation for a satisfaction claim asserting that the BOOLEAN is equal to _true_, such C-like conditionals as:
local: BOOL .x
when( .x ) set .y = 8;

are fine for BOOLEAN .x since in this case it is taken to mean the same as:

local: BOOL .x
when( .x = _true_ ) set .y = 8;

For those who know about somehows (see Chapters 5 and 9), every when assertion begins with an implicit somehow if it does not already begin with an explicit existential quantifier. The implication is that the only free variables which a when assertion can effectively contain are procedural variables defined explicitly outside of the assertion since if there were no outside variable corresponding to a variable used in the assertion, then that variable would be considered quantified by the nearest enclosing existential quantifier in the when assertion. To illustrate the implications of this, consider what would happen if the system were given:

when( .x ) set .y = 8;

where x does not appear anywhere else in the program. Then the system considers this to be equivalent to:

when( there_exists .x such_that( .x = _true_ ) ) set .y = 8;

which has exactly the same effect as:

set .y = 8;

Here’s a switch:

switch( substr( .y, 3, 1 ) )
do {
  case ( = "a" ) do { set .x = 5 }
  case ( = "b" | = "c" ) { set .x = 6; }
  case( Is_An_Upper & > "M" ) { set .x = 7 }
  case( Matches "[def]" ) do( set .x = 8; )
  else do {
    set .x = 9
  }
}

The associated grammar productions are:
In contrast to C switches, Cymbal enables the user to switch on terms (not just variable dereferences) of arbitrary type, in this case, STRs. Typically, the term switched on will just be a variable dereference such as .z but in this example, it is the substring of length 1 of .y beginning at character 3. If the term is a function call, then it is computed only once.

The cases are considered in sequence and when the first case is found whose assertion is true, then its command is executed and the switch is exited. Note that the command must be a do-group, not some other kind of statement like an assignment.

The assertion for a case is generated by taking the switched-on term and the extended predicate contained in the case and creating a valid sentence. For example, the assertions for the cases above are:

\[
\text{substr}(.y, 3, 1) = "a"
\text{substr}(.y, 3, 1) = "b" \text{ or } \text{substr}(.y, 3, 1) = "c"
\text{substr}(.y, 3, 1) \text{ Is_An_Upper and } \text{substr}(.y, 3, 1) > "M"
\text{substr}(.y, 3, 1) \text{ Matches } "[def]"
\]

There are no case fall-throughs like there are in C and hence there is no need for breaks. To achieve the effect of a fall-through, just use a | extended predicate.

### 6.5 Loops

#### 6.5.1 Conventional Loops

Cymbal has a full complement of conventional looping constructs. In Cymbal, the *while* and *until* assertions can go either before or after the loop body, at which point, they mean what they mean relative to that position. So, for example, if an *until* assertion appears after the loop body, then the understanding is that after the body has been executed, if the *until* assertion is not true, then the body will be re-executed. A *loop* with no *while* or *until* assertion will loop forever unless broken out of in some way. Here are a couple of simple loops (loop1.Q):
set .total_sq = 0;
set .idx = 0;
while( .idx <= 5 )
do {
    set .total_sq += .idx <* 2;
    set .idx++;
}
do Write_Line( "Total_Sq = ", .total_sq );
set .idx = 0;
loop {
    set .total_sq -= .idx <* 2;
    set .idx++;
} until( .idx > 5 );
do Write_Line( "Total_Sq = ", .total_sq );

Here is the relevant portion of the grammar:

```
Loop ::= loop BraceProg
      | loopstart BraceProg while Assertion until Assertion LoopModifier *
      | loopstart BraceProg until Assertion
      ;

LoopModifier ::= renewing_with BraceProg
              | before_doing_the_first BraceProg
              | after_doing_the_last BraceProg
              ;

loopstart ::= loop do
              ;
```

Here are some more intriguing examples (loop.2.Q):
set .x = 100;
while( there_is_a PART where( Number = .x ) )
    renewing_with {
        set .x++;
        when( .x = 150 ) do Exclaim_Line( "got to number 150" )
    }
do {
    when( .x Is_In [ 140 -> 160 ] ) do Write_Line( "x = ", .x )
} else {
    do Exclaim_Line( "There is no PART Number 100" );
}
loop {
    set .x--;
} until( .x = 0 or there_is_no ORDER where( Part_Nbr = .x ) )
do Write_Line( "x = ", .x )

The while loop above shows the Cymbal analog of the C for statement. In contrast to C, the optional renewing_with braced commands may be an arbitrary sequence of commands; these commands are executed at the end of the loop body prior to doing the test for "renewal", i.e., re-execution of the loop. The renewing_with construct enables all loop definition information to be collected together in one place at the start of possibly large loops. It also supports the use of continue which will cause the execution of the renewing_with code before looping around. The else clause illustrates an enhancement to the usual for loop: if the while Assertion is false (and hence the loop body is not even entered), then the else clause will be executed instead.

Here is another example (loop.4.Q)

// test query showing all supported loop keywords
set .y = 0; // initialize y and enable the first loop test to pass
while( .y <= 50 )
    before_doing_the_first { set .y = 10; }
renewing_with { set .y += 10 }
    after_doing_the_last { _Say_Eq(.y,60) }
    { _Show_Exp_To(.y) }
else { do Write_Line( "error!" ); }

The keywords before_doing_the_first and after_doing_the_last enable the user to specify code to be executed at the indicated places. For example, before_doing_the_first instructs Daytona to execute the associated do-group before executing the body of the loop for the first time -- assuming the loop while condition passes that first time. Just to be clear, the do-group specified by before_doing_the_first is done immediately before doing the first execution of the main do-group of the while loop, implying that it is not executed before the evaluating the while condition for the first time -- so this is not the analog of a C for loop specification. (_Say_Eq is a ds_m4 system macro that expands into a do Write_Line that says that the two arguments are equal. _Show_Exp_To is a ds_m4 system macro that expands into a do Write_Line that says that the expression argument is equal to its value.)
As indicated in the examples, as with all assertions used in procedural Cymbal, any declarative Cymbal assertion that constitutes a test is allowed; the one in the example is testing if .x is a PART Number. Notice that this assertion is a test since the values for all its variables are determined outside of it; in contrast, for_each_time loops generate values for variables from declarative assertions.

### 6.5.2 For_Each_Time Loops

for_each_time commands form the bridge between the declarative and procedural worlds: they enable values to be taken from declarative settings for use in procedural settings. The idea is that "for each time Daytona can find values for a tuple of variables that make some assertion true, Daytona will do something with those values". (Even though the for_each_time command is strictly speaking just a special kind of loop (and is therefore described here along with the other loops), in order to truly understand how they work, one must also be familiar with the declarative portion of Cymbal. Consequently, the reader may wish to browse this section now and to return to it later after having read Chapter 9 on declarative Cymbal.) Here are several examples (foretime.2.Q):

```plaintext
/** 1 **/
for_each_time [ .x, .y ]
is_such_that(  
    .x Is_In [ 1-> 5 ] and .y Is_In [ 1 -> .x ]
) do {
    do Write_Line( "The sum is ", .x + .y );
}

/** 2 **/
for_each_time [ .x ]
is_such_that( .x Is_In [ 1-> 5 ] )
do {
    do Write_Line( ".x = ", .x );
}

/** 3 **/
let INT(_short_) .x
ist(   
    there_is_a SUPPLIER where(  
        Number = .x which_is < 405 )
) {  
do Write_Line( .x )
} else {  
do Write_Line( "Note: can’t find suitable SUPPLIER" );
}
/** .x not accessible here
 * do Write_Line( .x );
**/
```

The first for_each_time above generates all pairs of values for the variables x and y which
satisfy the for_each_time assertion, while printing those pairs out as it does so.

The command that is executed 'each time' a tuple of answers is produced must be a do-group; the parser will not accept anything else, not even a standalone assignment. The command has to be a do-group, with or without the do.

As indicated by the first and second examples, the most general for_each_time generates values for a TUPLE of variables; as indicated by the third example, if there is only one variable, then the LIST notation can be omitted. The third example illustrates that for_each_time can be abbreviated by fet and that is_such_that can be abbreviated by ist. The third example also illustrates that an optional else do-group can attached to a for_each_time: this do-group will be executed if there is no way to satisfy the for_each_time assertion.

The first and second examples use Cymbal interval constructs to define the values that the variables will take. The .x upper bound in the interval range statement for .y illustrates that, as is generally the case, a fully general term can appear any place where a constant appears.

In rare instances, the user will need to specify some type information for the variables used in a for_each_time variable TUPLE. This is illustrated by the INT(_short_) used in the third example above. However, for emphasis, as frequently occurs, such typing is not needed in this case because Daytona’s type inference mechanism is able to automatically infer the type of the variable x.

The associated grammar production is:

\[
\text{ForEachTimeLoop} ::= \text{for\_each\_time} \\{ \\text{SomeVblSpecs} \\text{is\_such\_that} \\text{P} \\\text{BoundedAsn} \\text{Do} \\text{else} \\text{Do} \\text{P} \\\n\text{SomeVblSpecs} ::= \text{VblSpec} | [ \text{VblSpecSeq} ]
\]

A VblSpec is one of these possibly typed variable specifications exemplified above. They will be discussed in more detail later in the section on defining variables.

For those who know about somehows (see Chapters 5 and 9), every for_each_time assertion begins with an implicit somehow if it does not already begin with an explicit existential quantifier.

Here is the first and most important rule about for_each_time usage: the for_each_time variables are known precisely and only within their for_each_time command. For example, the x referred to in the commented-out Write after the third for_each_time is considered by Daytona to be a different variable from the x used in that for_each_time. In fact, over the entire for_each_time program text above, x is taken in turn to be one of 3 different, unrelated variables. In Daytona error messages, each of these variables has a different name based on "x": these names are all like x_sCP2 which is the unique name of the user variable named "x" as it appears in its second scope.

In fact, if that Write is uncommented, then the query cannot be processed by Daytona. The reason is that the variable named x in the Write statement (which, as just remarked, is different from the other variables named x in the query) has not been given a type by a definition (or import) nor has it been assigned a value previously. Consequently, Daytona cannot determine any useful type for it and will therefore not process the query. In any event, that Write cannot achieve the apparent

Copyright © 2013 AT&T
All Rights Reserved.
September 15, 2013
objective of the query writer which is to access the last value of the for_each_time loop index. To get that value known on the outside of the loop, it must be assigned in the inside do-group to be the value of some variable whose scope strictly includes the for_each_time loop.

(The reason for this restricted for_each_time variable scoping rule is that Daytona optimization reserves the right to reorder conjuncts in the for_each_time assertion thus making it infeasible, in general, for the user to be able to predict what the last satisfying TUPLE of answers will be. Therefore, since that last satisfying TUPLE has no predictable meaning, it’s important to ensure that it is not available on loop exit. Once again, the user is free to, say, collect any answer TUPLE in an external box for subsequent use after the loop exits. This is called don’t know nondeterminism in logic programming because, while you know you want all the answers, you don’t know what order they will be produced in.)

A second important rule is that no outside variable for the for_each_time assertion can be modified in the do-group. There are two ways to get around this stricture as described in the For_The_First_Time section immediately following this one as well as in Chapter 12 on boxes. Observe that it is permissible to change the values of for_each_time TUPLE VBLS in the do-group.

Another very important rule is, do not call transactions in a for_each_time do-group that change some table that is being looped over in the for_each_time assertion. Finally, it is currently illegal to execute a return statement from within the body of a for_each_time assertion.

fet is a convenient abbreviation for for_each_time and ist is a convenient abbreviation for is_such_that.

Daytona provides convenient abbreviations for iterating through the members of aggregates (foretime.3.Q):
/** 1 **/
for_each_time .x Is_In [ 1-> 5 ]
{
    do Write_Line( .x );
}
 /** 2 **/
for_each_time
    .color Is_Something_Where( 
        there_is_a PART where( Color = .color ) ) in_lexico_order
     do {
        do Write_Line( upper_of( .color ) );
    }
 /** 3 **/
for_each_time .x Is_In
    $[ select Number from SUPPLIER where Number < 405 ]$
    in_lexico_order
     do Write_Line( .x );
 /** 4 **/
for_each_time [INT .x] = tokens( for "1:2:3" upto ":" )
     do Write_Line( .x );

The associated grammar productions are:

$$
\text{ForEachTimeLoop} ::= 
    \text{for_each_time \ SomeVblSpecs} \ \text{BoxFormerPred} \ \text{BoundedAsn} \ \text{HybridBoxKeywdArg} \ \text{LoopModifier} \ \text{Do} \ \text{else} \ \text{Do} \ \text{Is_In} \ \text{Aggregate} \ \text{UseBoxKeywdArg} \ \text{CompoundPred} \ ; \\
\text{for_each_time \ SomeVblSpecs} \ \text{LoopModifier} \ \text{Do} \ \text{else} \ \text{Do} \ \text{Is_In} \ \text{Aggregate} \ \text{UseBoxKeywdArg} \ \text{CompoundPred} \ ; \\
\text{for_each_time \ SomeVblSpecs} \ \text{Do} \ \text{else} \ \text{Do} \ \text{Is_In} \ \text{Aggregate} \ \text{UseBoxKeywdArg} \ \text{CompoundPred} \ ; \\
\text{for_each_time \ SomeVblSpecs} \ \text{LoopModifier} \ \text{Do} \ \text{else} \ \text{Do} \ \text{Is_In} \ \text{Aggregate} \ \text{UseBoxKeywdArg} \ \text{CompoundPred} ;
$$

The first example illustrates how the elements of an INT lattice can be iterated over whereas the second illustrates how to iterate over the members of a Cymbal set. (See Chapter 12 for more on Cymbal set-formers.) The third example shows one way to make the values of an SQL query available for procedural Cymbal programming.
6.5.3 For_The_First_Time, For_The_Last_Time Loops

Since the for_each_time assertion can in general be satisfied in more than one way, it is sometimes useful to be able to insist that the loop be executed at most once. This is accomplished using the for_the_first_time variant of for_each_time:

```cybal
for_the_first_time .xx is_such_that(
   .xx Is_In [ 1 -> 6 ]
) do {
   do Write_Line( .xx );
}
set .nn = 6;
for_the_first_time .xx Is_In [ 1 -> .nn ]
do {
   do Write_Line( .xx );
   set .nn = .xx;
}
```

As will be seen when discussing box deletes in Chapter 12, there are occasions when it is desired to change one of the outside variables in the for_each_time assertion in the body of the loop. This is forbidden because, in the general case, it greatly complicates the generation of the loop indices since it causes the for_each_time assertion to possibly change with every iteration of the loop. However, if the loop body is guaranteed to only be executed at most once (as it is with for_the_first_time), then the loop body is allowed to change outside variables for the for_each_time assertion since they will not have an opportunity to interfere with subsequent iterations of the loop, there not being any. Chapter 12 discusses other strategies as well.

Correspondingly, Cymbal supports for_the_last_time.
set .nn = 7;
for_the_last_time .xx Is_In [ 1 -> .nn ] do {
    do Write_Line( .xx );
    set .nn = .xx;
} _Say_Eq(.nn,7)
for_the_first_time .pno is_such_that( there_is_an ORDER where( Part_Nbr = .pno and Quantity > 1500) ) {
    for_the_last_time .supp is_such_that(
        there_is_an ORDER where( Supp_Nbr = .sno and Part_Nbr = .pno )
        and there_isa SUPPLIER where( Name = .supp and Number = .sno )
    ){
        do Write_Line( "The winning supplier is .supp"ISTR );
    }
}

6.5.4 Control Break Programming With For_Each_Time

By using the LoopModifiers renewing_with, before_doing_the_first, and after_doing_the_last, the user can nicely express what is called control-break programming. This is where a sorted list of tuples is being processed and is thought of as being a sequence of groups defined by common values on the sort fields for adjacent records. Every time one of those sort fields changes from one record to another, a new group is being entered. Here is a query that counts the number of Suppliers for each City (loop.4.Q):

for_each_time .city Is_The_Next_Where( there_is_a SUPPLIER where( City = .city ) )
    in_lexico_order
    before_doing_the_first { set .cnt = 0; set .prev = .city; }
    renewing_with {
        when( .city = .prev ){" ", .prev);" ", .prev);
        set .cnt = 1;
        set .prev = .city;
    }
    after_doing_the_last { do Write_Line(.cnt, " ", .prev); }
})

Of course, this pedagogical example is just a simple group-by query that could be done even more concisely with:
select City, count(*) from SUPPLIER group by City order by City;

Note that the first set of values for the _for_each_time_ variables are available for use in the _before_doing_the_first_ do-group which by definition is in no way considered to appear lexically prior to the _for_each_time_ nor could it in general because it is allowed to refer to _for_each_time_ variables.

The real value of these _LoopModifiers_ for _for_each_time_ queries can be seen with _after_doing_the_last_. In general, the user cannot know at compile time how many TUPLES a _for_each_time_ will be able to find, let alone what the last one will be. But using _after_doing_the_last_, the user can access and use that last TUPLE of values expressed using the _for_each_time_ variables before they go out of scope just after the _for_each_time_ loop syntax in the code. This enables the user to obtain the effect of a _for_the_last_time_ loop (which is also supported directly as is) as contrasted with a _for_the_first_time_ loop.

Note that, true to their names, neither the _before_doing_the_first_ do-group nor the _after_doing_the_last_ do-group will be executed if the _for_each_time_ assertion cannot be satisfied (which would then imply that its do-group is not even executed once).

Here is a more involved query that computes the same kind of quantities that a report writer would do. A defining characteristic of a report writer is that it allows the user to specify printing out a table interspersed with subtotals or other aggregates computed over preceding groups of records. These sub-aggregates may themselves be aggregated up to cover more inclusive groups in a hierarchical fashion. Here is an example showing how to print out detail records interspersed with three levels of hierarchical aggregates (loop.5.Q).
for_each_time [ .color, .supplier, .qty, .wt ]
Is_The_Next_Where(
    there_is_an ORDER where( Supp_Nbr = .sno and Part_Nbr = .pno
        and Quantity = .qty )
    and there_is_a PART where( Number = .pno
        and Color = .color and Weight = .wt )
    and there_is_a SUPPLIER where( Name = .supplier and Number = .sno )
) in_lexico_order
before_doing_the_first {
    set .prev_color = .color; set .prev_supplier = .supplier;
    set [ .cnt, .c_cnt, .cs_cnt ] = [ 0 ];
    set [ .tot_wt, .c_tot_wt, .cs_tot_wt ] = [ 0.0 ];
}
renewing_with {
    when( .color = .prev_color and .supplier = .prev_supplier ){
        set .cs_cnt++;
        set .cs_tot_wt += .qty*.wt;
    } else {
        do Write_Line( 
            "= .prev_color .prev_supplier: .cs_cnt .cs_tot_wt"ISTR );
        set .prev_supplier = .supplier;
        set .cs_cnt = 1;
        set .cs_tot_wt = .qty*.wt;
    }
    when( .color = .prev_color ){
        set .c_cnt++;
        set .c_tot_wt += .qty*.wt;
    } else {
        do Write_Line( 
            "== .prev_color: .c_cnt .c_tot_wt"ISTR );
        set .prev_color = .color;
        set .c_cnt = 1;
        set .c_tot_wt = .qty*.wt;
    }
    set .cnt ++; set .tot_wt += .qty*.wt;
    do Write_Words( 
        ".color .supplier: .qty * .wt ="ISTR, .qty * .wt );
}
after_doing_the_last {
    do Write_Line( 
        "= .color .supplier: .cs_cnt .cs_tot_wt"ISTR );
    do Write_Line( "== .color: .c_cnt .c_tot_wt"ISTR );
    do Write_Line( "===: .cnt .tot_wt"ISTR );
}

The data has been sorted by Supplier within Color. For each collection of records with the same Color
and Supplier, the records are printed out first and then the total number of records for that group
together with the associated total of the weight for all parts ordered for that Color and Supplier. The next level of aggregates aggregates over all Suppliers within a Color and the last level aggregates over all Colors. While an actual report-writer would provide a more concise way of expressing this query, at least it can be done clearly and effectively in Cymbal.

6.6 Branches

Cymbal contains slightly augmented breaks, continues, gotos, and returns (break.3.Q).

```cymbal
for_each_time .idx Is_In [ 2, 4, 8 -> 10 ]
d0 {  
  set .jdx = 1;  
  until( .jdx = 4 )  
    renewing_with{ set .jdx++ }
  loop{  
    begin_loop2:  
      when( .idx = 9 ) break( 2 );  
      when( .jdx = 3 ) continue;  
      do Write_Words( .idx, .jdx, .idx + .jdx );  
      when( .jdx = 4 ) goto end_loop1;  
      when( .jdx = 2 )  
      {  
        set .jdx++; go begin_loop2;  
      }  
    }  
  }  
end_loop1: ;
}
```

The corresponding syntax:

- **Break** ::= 
  - break [ ( integer ) ]
  - leave [ ( integer ) ]

- **Continue** ::= 
  - continue [ ( integer ) ]
  - loop_again [ ( integer ) ]

- **Goto** ::= 
  - goto lower  
  - go lower

- **Return** ::= 
  - return [ ( Subject ) ]

There are several comments to be made here. First, while a single command can only have one label, it is easy to get the effect of multiple labels by using null or empty commands as was done with the label `end_loop1` above.

Also, as illustrated above, breaks and continues may take an optional integer argument to inform
Daytona how many loops to break or continue out of; the default is 1. This is one rare instance where the argument must be a constant and not a general term. *leave is break* and *loop again is continue*. *breaks* can be used to break out of *for_each_time* loops, possibly in nests containing regular while loops, as long as the last loop broken out of is a *for_each_time* loop. There are no restrictions on *continues* in nests of regular while loops; however, any *continue* whose smallest containing loop is a *for_each_time* loop can only have an implicit or explicit argument of 1.

*returns* are conventional and are discussed along with functions and procedures.

### 6.7 Procedure Calls

The simplest, most powerful way to do something in procedural Cymbal is to invoke one of Daytona’s built-in procedures with a procedure call. Cymbal procedures may take keyword arguments or positional arguments or both. The *Write* procedure is one that uses both:

```
  skipping 2
to .out_chan
do Write_Line( "x = ", .x )
```

Notice, as must be the case with all PROCEDURES, *Write Line* has a name which is an UPLOW, which means that its first letter is upper-case and some subsequent letter is lower-case. (No such restriction applies to names for C_external procedure tasks or imported C procedures when they are declared or defined prior to use in the Cymbal file that uses them.) The positional arguments appear comma-separated and in order within parentheses. Not surprisingly, this *Write Line* will be writing out the string "x = " followed by the string representation of whatever value `x` has. This means that *Write Line* is an overloaded procedure; it is happy to write out the string representations of values of many types without needing any instructions on how to do so, although, of course, formatting functions are provided for customized output.

This particular call to *Write Line* has two keyword arguments. "skipping 2" has the keyword *skipping* and the argument 2 and serves to specify that *Write Line* must skip 2 lines before beginning to write characters. The value of the *out_chan* variable is the I/O channel to which the output is to be sent. Keyword arguments may appear in any order in a Cymbal procedure call. *Write Line* and other I/O functionality are discussed further in Chapter 8.

Here is the corresponding syntax:

```
        | [ [Keyword Arg] ] do procedure ( Subject [ , Subject ] )

Keyword Arg ::= lower | preposition Subject
               | preposition Aggregate
```

The *Write Line* call above corresponds to the third alternative for a *Procall*. The remaining alternatives are illustrated in the next set of examples (procall.1.Q). Notice that the Display call makes
use of keywords that have no arguments.

/** 1 **/  
with_title_lines [ "suppliers from St. Paul",  
  "-- just a test query --" ]  
with_no_heading  
with_no_closing  
do Display each[ .supplier ]  
each_time( there_is_a SUPPLIER named .supplier where(  
    City = "St. Paul" )  
);  
/** 2 **/  
select Name as Supplier from SUPPLIER where City = 'St. Paul';  
/** 3 **/  
$[  
  begin;  
  ( select Date_Recd from `ORDER` where Date_Recd > `12.6.86`  
    union  
    select Date_Recd from `ORDER` where Date_Recd < `1.15.85` );  
  end  
]$  

Example 1 reveals that Display is nothing more than a Cymbal procedure taking keyword arguments. The each keyword takes a TUPLE argument and the each_time keyword takes an assertion argument. Also, given the absence of positional arguments, the keyword arguments can appear in any order before or after (both or neither) the do procedure phrase. When positional arguments are used, all keyword arguments present must appear before the do invocation. Cymbal provides syntax for specifying what arguments are allowed in procedure calls; this is addressed at the end of this chapter.

Examples 2 and 3 illustrate that as far as Cymbal is concerned, SQL queries are just procedure calls; in fact, they are processed by translating them into Display calls. The Cymbal parser does need help with any SQL query that begins with a parenthesis or with begin; such help is provided by bracketing the SQL with $[ and ]$, which can be done in any event.

6.8 Program Structure

The simplest and most common way to write procedural Cymbal is to string together a bunch of commands with semicolons and then to ask Tracy to translate them into C for compilation and execution. Here is a sample program:

start: set .x = 4;  
set .z = 6 * .x;  
do Write_Line( .z );
In general, the syntax for such programs is:

\[
\text{Program} \quad ::= \quad \text{CmdSeq} \\
\text{CmdSeq} \quad ::= \quad \text{ProgAtom} \quad ; \quad \text{ProgAtom} \quad [\quad \text{CmdSeq} \quad ]^* \\
\text{ProgAtom} \quad ::= \quad \{ \quad \text{lower} : \quad \text{I}^* \quad \text{Command} \quad \} \quad | \quad \{ \quad \text{lower} : \quad \text{I}^* \quad ; \quad \text{Command} \quad \} \\
\text{Command} \quad ::= \quad \text{Assignment} \quad | \quad \text{When} \quad | \quad \text{Switch} \\
| \quad \text{Do} \quad | \quad \text{Loop} \quad | \quad \text{ForEachTimeLoop} \\
| \quad \text{Break} \quad | \quad \text{Continue} \quad | \quad \text{Return} \quad | \quad \text{Goto} \\
| \quad \text{ProCall} \\
\]

These simple programs are actually a special case of a more general architecture. In the most general setting, a Cymbal program is a sequence of global definitions defining procedural tasks, declarative functions and predicates, and object classes. For stand-alone programs, at least one task definition is for the Begin task which (like main() in C) is called to initiate the processing of the program. Whenever the user asks Tracy to process a sequence of commands, Tracy quietly collects these commands into a Begin task definition and a call to that Begin task and processes that instead. So, the little program above is considered to be the same as:

\[
\begin{align*}
\text{do} & \text{ Begin;} \\
\text{global_defs:} & \\
\text{define} & \text{PROCEDURE} \text{ task}: \text{Begin()} \\
\{ & \\
\text{start:} & \text{set} \quad .x \quad = \quad 4; \\
& \text{set} \quad .z \quad = \quad 6 \quad * \quad .x; \\
& \text{do} \quad \text{Write}_\text{Line}( \quad .z \quad ); \\
\} \\
\end{align*}
\]

A task is a function/predicate/procedure (i.e., fpp) which has at most one level of helping fpps nested within it. Semantically, the idea is that there is some goal that a task is intended to achieve and the code that it has for doing that is free to call its own private helper fpps as well as other tasks. To this end, the nested fpps are automatically known (without need of explicit declaration) throughout their task and not at all outside of it. Furthermore, any variable or fpp known in the parent task is automatically known in its nested, helper fpps; helper fpps are not allowed to export any of their variables. So, a task is like a procedure in PL/I which may contain other PL/I procedures nested to depth one within it. Task definitions appear at the end of a Cymbal program in the global_defs section. It is illegal for the user to provide an explicit import of a helper since the system automatically does the import in any event; this way, if the user sees any fpp imports in a Cymbal task, they know that those fpp imports are for external-to-the-task Cymbal task or C routines.

Tasks plus their helper fpps form an encapsulation package that provides the same basic modularization capability provided by C program files. Some of the benefits are that the declaring of
shared information is done automatically by Daytona, that there no notion of something being known
or unknown due to its location in the file, and that there is no need to introduce to a high level
language like Cymbal a notion as primitive and as divorced from language considerations as that of a
file.

Also, all system built-in fpps and all user C and Cymbal add-on fpps (imported and defined in
usr.env.cy; see the end of this chapter) are considered globally scoped and are known in all query tasks.

So, more generally, if the user prefixes or suffixes a sequence of commands with helper fpp
definitions and submits that to Tracy, then Tracy quietly collects this whole bundle into a Begin task.

The most general syntax for a Cymbal program is:

\[
\begin{align*}
\text{Program} & ::= \text{EnvStmt} \mid \text{CmdSeq} \mid \text{EnvStmt} \mid \text{GlobalDefs} \mid \\
\text{EnvStmt} & ::= \text{ClassDefs} \mid \text{HelperFppDefs} \mid \text{ExportedVblDefs} \\
& \mid \text{LocalVblDefs} \mid \text{Imports} \\
\text{GlobalDefs} & ::= \text{global } \text{defs} \mid \text{gbldj} \mid \text{global } \text{defs} \\
\text{gbldj} & ::= \text{with } \text{version} \mid \text{Subject} \\
\text{GlobalDef} & ::= \text{TaskDef} \mid \text{DeclFppDef} \mid \text{ClassDef} \\
\end{align*}
\]

A Cymbal program consisting of task definitions alone (which will be explained shortly) and no
commands is considered to be a request by the user to generate the code for those tasks and then to
quit without executing any of them. The various \text{EnvStmt} (environment statements) will be discussed
next.

6.8.1 Version Identification For Code

By attaching a \text{with } \text{version} keyword argument to a \text{global } \text{defs} statement, the user can
indicate that all tasks in a given query (including the Begin task, if any) are all from the indicated
version:

\[
\text{global } \text{defs with } \text{version "1.0 from 2/2/90" :}
\]

The version argument must be a constant (i.e., not a variable dereference or function call) and it may be
an element from the following types: STRINGS, INTS, FLTTS, DATES, THINGS.

The advantage of using \text{with } \text{version} is that Daytona will put an SCCS ID string containing that
version identifier in every .c file that it generates from the associated query file. In this case, the string
"@(#) 1.0 from 2/2/90" would appear in each .c file. The \text{what}(1) command prints out such
strings when applied to executables.

See also the section in Chapter 23 on schema evolution where versioning is used to distinguish
between different versions of apds and rcds.

6.9 Defining And Importing Variables

Daytona goes to moderate lengths to infer the types of variables. In some cases however, such as when variables get their values by means of read() or Read() calls, Daytona really has no sound basis for determining the type of those variables (although it does assume STR by default). In other cases, the user may wish to override Daytona’s type inference procedure with their own preferences, as is the case when STR variables with a specified maximum length are desired. In still other cases, as with composite classes, the type is so complex that it is valuable (and in fact, necessary) just to say explicitly what it is. In any event, it is sometimes necessary to have the syntax available to specify user choices for the types of variables and that syntax is described in this section.

There can be at most one definition for a variable in a Cymbal program. A variable’s definition causes the system to allocate space for holding the variable’s values; an import (or declaration) for a variable merely allows another fpp to (intelligently) access the values of the variable by means of providing it with the variable’s name and type information.

Cymbal provides the syntax for making local variable definitions as well as for specifying the importing/exporting of variables in and out of fpps. Here are some examples:

imports:
   STR .township
   BOOL VBL aok
   INT : .x1, .x2
   C_external DATE ARRAY[ [0->], 12 ]: .uu1, .uu2, .uu3
   INT .truck_nbr[ "Chevy", "Ford" ]
exports:
   INT .car_nbr[ "Saab", "Pinto", "Falcon", "Corvair" ] = [ 1, 2, 3, 4 ]
local:
   constant DATE: .now = today() +1 ;
   constant DATE: .tomorrow = .now +1;
   static INT .nbr_times_called
   CHANNEL(_file_) VBL in_chan = _stdin_

These statements about variables are grouped into imports, exports, and locals. The order of the groups is not important: a group begins with a keyword (like imports) and continues until something else begins like the next group or a procedural command. There can even be multiple groups of the same type.

The first two imports illustrate two different ways of understanding Cymbal variable definition/declarations. The first one is C-like in that it asserts that township is something whose value is of type STR: consequently township must be a variable, and in fact is an STR VBL. The second one just comes out and says directly that aok is an object which has type BOOLEAN VBL.

The third import illustrates how the type of variables can be factored out in a
definition/declaration so as to enable terse type specification for several variables at the same time. One can even write equivalently:

\[
\text{INT VBL: } x1, x2
\]

Notice that these variable definition/declarations may end with a semicolon -- or with nothing at all, it doesn’t matter.

If a subclass specifier is not mentioned in a variable definition, then a default is assumed according to the following table:

<table>
<thead>
<tr>
<th>Default Subclass Specifier</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>INT <em>long</em></td>
<td></td>
</tr>
<tr>
<td>FLT <em>long</em></td>
<td></td>
</tr>
<tr>
<td>STR *</td>
<td></td>
</tr>
<tr>
<td>RE *</td>
<td></td>
</tr>
<tr>
<td>LITERAL *</td>
<td></td>
</tr>
<tr>
<td>THING *</td>
<td></td>
</tr>
<tr>
<td>BITSEQ *</td>
<td></td>
</tr>
<tr>
<td>CHAN <em>file</em></td>
<td></td>
</tr>
</tbody>
</table>

The system records this information on default values in `sys.env.cy`.

Use the type `STR(=)` to import variables into Cymbal programs that are defined to be `char *` C variables in some application C program being linked in with Daytona-generated C code. `STR(=)` VBLS should be used only for this purpose and even then, they must be used with care. In particular, it is very important never to assign `STR(*)` function calls or dynamic variable values to `STR(=)` VBLS because such results are typically temporary values that will be subsequently collected as garbage. In such cases, the safest thing to do is to use a fresh copy of the string in question by using `copy_of_str`. (However, just to be clear, any string created and returned by `copy_of_str` is not part of the garbage collection system and so there is no mechanism in place to free it.) Also, please remember that an `STR(=)` VBL that is set to the C-language NULL should never be used in a Cymbal expression.

When the keyword `local` is used in a VBL definition, then, if the variable is proper to a task, it is known only within that task and its helpers whereas if it is defined in a helper then it is known only within that helper. When the keyword `exports` or `export` is used for the definition of a variable proper to a task, the variable is not only known locally but is also made available to be known by any task which explicitly `imports` it with an `import` declaration. An important caveat here is that explicit imports are necessary in order for one task to be able to use another task’s explicitly exported variables. How to place `import/export/local` VBL statements in `fpp` definitions will be described in the next section. Procedural variables that are not explicitly scoped with `import/export/local` VBL statements are scoped by the type inference mechanism to the smallest helper or task which includes all of their occurrences.

Meanwhile, observe that these constructions all exemplify the following grammar productions:
As illustrated in the examples and grammar, such `vbladjs` as static and dynamic can be used to further define the properties of VBLS. A static VBL is one whose value persists from one invocation of its containing fpp to the next. Consequently, any variable exported or imported is assumed static by default. A dynamic VBL is the opposite (and is the default), i.e., a VBL whose value storage space is reallocated each time its containing fpp is called. A constant VBL is one which always maintains its initial value: no assignment is ever allowed to change it. alias and copy are used in defining parameter VBLS, as will be discussed shortly.

A C_const VBL is one which is not only constant at the Cymbal level but also is declared at the C level using const. This can be useful when importing C functions that insist upon using const arguments.

A C_external VBL is one whose C 3GL name is required to be the same as its Cymbal 4GL name. Ordinarily, Daytona decorates Cymbal VBL names with additional information which serves to minimize name conflicts with other programs in the UNIX and Daytona environment as well as to support the separate compilation of Daytona tasks. This name augmentation can be forbidden by using the C_external keyword so as to facilitate the sharing of VBLS among Daytona-generated C code and user-provided C code.

The manifest adjective only has meaning in the context of ArgSlotDef.

All procedural Cymbal variables have initial or default values. If not specified, they default to the following (based on type):
### Default Initial Values

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT</td>
<td>0</td>
</tr>
<tr>
<td>UINT</td>
<td>0</td>
</tr>
<tr>
<td>FLT</td>
<td>0.0</td>
</tr>
<tr>
<td>MONEY</td>
<td>&quot;0.0&quot;MONEY</td>
</tr>
<tr>
<td>STR</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>RE</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>LIT</td>
<td>``</td>
</tr>
<tr>
<td>THING</td>
<td>``</td>
</tr>
<tr>
<td>SHELLP</td>
<td>``</td>
</tr>
<tr>
<td>CMD</td>
<td>``</td>
</tr>
<tr>
<td>DATE</td>
<td>&quot;12/31/9999&quot;</td>
</tr>
<tr>
<td>TIME</td>
<td>&quot;0s&quot;TIME</td>
</tr>
<tr>
<td>BOOL</td>
<td><em>false</em></td>
</tr>
<tr>
<td>BITSEQ</td>
<td>&quot;0&quot;B</td>
</tr>
<tr>
<td>CHAN</td>
<td><em>nullChan</em></td>
</tr>
<tr>
<td>DOYDATE</td>
<td>&quot;84001&quot;DOYDATE</td>
</tr>
<tr>
<td>ATTDATE</td>
<td>&quot;1/1/1984&quot;ATTDATE</td>
</tr>
<tr>
<td>CLOCK</td>
<td>&quot;00:00:00&quot;CLOCK</td>
</tr>
<tr>
<td>DATE_CLOCK</td>
<td>&quot;12/31/9999@00:00:00&quot;DATE_CLOCK</td>
</tr>
<tr>
<td>IP</td>
<td>&quot;0.0.0.0&quot;IP</td>
</tr>
<tr>
<td>IPORT</td>
<td>&quot;0.0.0.0:0&quot;IPORT</td>
</tr>
</tbody>
</table>

There are more actually for additional types. The system records this information on initial values in `sys.env.cy` as the values of `with_cy_init_val` in CLASS definitions.

As illustrated in part by the local definition of `now` and `tomorrow`, Cymbal variable initializers may contain function calls and references to other variables (as long as they were defined in previous definitions).

### 6.9.1 Defining, Importing And Initializing Array Variables

A variable is an array variable if its definition/declaration includes a `DimSpecTuple`. Each of the possibly several dimensions of the array may be of one of three types:

- **INTEGER-lattice dimension**
  where the dimension indices consist of a sequence of evenly-spaced INTS specified either by an INTEGER INTERVAL or by an actual INT constant `ic` (like 47) in which case the INTERVAL is understood to be `[ 1 -> ic ]`.

- **static associative dimension**
  where a TUPLE of constants is given to enumerate precisely the indices for the dimension. These constants may be of any scalar type except FLTS. (FLTS are ruled out due to the unpredictable impact of round-off error on computing index values.)
• **dynamic associative dimension**

which is the subject of the associative array chapter. The current constraint is that either all dimensions are dynamic associative or none of them are.

These dimension types are illustrated by the following examples, reproduced here from an earlier display:

**imports:**

```plaintext
C_external STR( 23 ): .ss[ 5, [10->20] ], .tt[ 0-> by 2 ]
C_external DATE ARRAY[ [0->], 12 ]: .uu1, .uu2, .uu3
INT .truck_nbr[ [ "Chevy", "Ford" ] ]
```

**exports:**

```plaintext
INT .car_nbr[ [ "Saab", "Pinto", "Falcon", "Corvair" ] ] = [ 1, 2, 3, 4 ]
```

Note that TUPLE-valued ARRAYS are supported for all types of ARRAYS.

When importing ARRAY VBLS or passing them as parameters, Cymbal allows the array dimensions to be only partially specified by saying where they start but leaving unspecified what the last value is. This is the case with `tt` in the above example where `tt` is said to be a one-dimensional VBL whose indices are some subset of the non-negative, even INTS. Outside of the dynamic associative array case, ARRAY VBL imports and fpp parameter definitions can have at most their first dimension be of unspecified size as is the case with `uu1` above. Otherwise, of course, the number of elements of each dimension must be specified. Note that any hope of array bounds checking and index validation is lost when imports like these are used; the user assumes full responsibility for any subsequent mistakes leading to core-dumps.

Notice how the ARRAY type specifier is used in the type-factored import of `uu1`, `uu2`, and `uu3`. This import can be factored even more as illustrated by the equivalent:

```plaintext
C_external DATE ARRAY[ [0->], 12 ] VBL: uu1, uu2, uu3
```

Daytona’s array variable initialization process is somewhat different from C’s. For example, in contrast to C, if there are fewer initializers for an array than the array has elements, then the last initializer is used for all of the remaining elements. (C just uses 0, meaning a suitably long sequence of 0 bits). Array elements are assigned initializers in the order generated by having the indices for later dimensions vary more rapidly. Also, initializations may contain arbitrary expressions, meaning function calls and variables and what not: but, of course, the initialization expressions can only make use of variables that have already been initialized, i.e., appear earlier in the initialization sequence.

For one-dimensional arrays, if the dimension specification is given by an INTERVAL that has no explicitly given upper bound, then the size of the ARRAY is determined by the number of initializers. For example, the `ara` ARRAY below has 4 elements in it:

```plaintext
```

Array initializations may contain nested TUPLE brackets in order to improve readability. Daytona
ignores them. Example:

\[
\text{INT: } .x[ 2, 3 ] = [ [1, 2, 3], [4, 5, 6] ]
\]

Here \( .x[2,1] = 4 \).

Those who use ds_m4 macros with Daytona should be aware of the fact that ds_m4 misbehaves when a Cymbal multi-dimensional array element reference is an argument to a ds_m4 macro. The reason is that the brackets do not serve to hide the comma. In this case, just quote the argument with squawks:

\[
\_\_\_\text{Say\_Eq(}@<.x[ 2, 3 ]@, 4)\]

6.9.2 Variable Specifications

Various constructs within Cymbal such as for_each_time loops and aggregate functions enable the user to specify in situ new declarative variables and (optionally) their types. These are done by means of VblSpecs which are almost the same as the procedural variable definitions just described. Here are some examples:

\[
.x
\]

VBL x

INT .x

FLT(_short_) VBL x

The associated grammar productions are:

\[
\text{VblSpec} ::= \_\_\_vbladj \_\_\_Type \_\_\_ . \_\_\_ lower \_\_\_ DimSpecTuple \_\_\_ \\
\text{VblSpecSeq} ::= \_\_\_VblSpec \_\_\_, \_\_\_VblSpec \_\_\_ \\
\text{SomeVblSpecs} ::= \_\_\_VblSpec \_\_\_|[\_\_\_VblSpecSeq]\_\_\_ \\
\]

6.10 Importing And Defining Functions, Predicates And Procedures

The Cymbal syntax for defining and importing fpps is more expressive than that for languages like C since Cymbal supports additional functionality such as argument defaults, keyword arguments, keywords with no arguments, and varying multiplicities of arguments.

6.10.1 Importing Fpps

Fpp imports are statements that are used to declare to the system what an fpp is called and how it is invoked. Here is a sample Cymbal fpp import.
imports:
STR FUN substr( STR, INT, INT = _all_ )

/* sample invocations */
set .s1 = substr( "abcdef", 2, 3 );
set .s2 = substr( .s1, 2, _all_ );
set .s2 = substr( .s1, 2 );

This import says that substr is a FUNCTION which returns a STR when invoked with a STR and two INTS. If the third argument is not supplied in the call, then _all_ is assumed to be that argument. This import is written in the C-style where substr is placed in a position (i.e., in front of the parameter specifications) analogous to the way that it is positioned in an actual call. Daytona also supports factoring out the type completely and asserting that substr has that type:

STR FUN( STR, INT, INT = _all_ ) substr

This style of import is the most conducive to type-factoring:

INT( _long_ ) FUN( INT, INT) :
  plus_int, sub_int,
  mult_int, div_int,
  mod_int,
  min_int, max_int

The return type alone can also be factored in the C-style syntax but the result is more verbose:

INT( _short_ ) FUN: min_sh(INT, INT), max_sh(INT, INT)

Here is an (abbreviated) import for the Display PROCEDURE:

PROC Display(
  ( 0->1 ) with_title_line STR,
  ( 0->1 ) with_title_lines TUPLE[ ( 0-> ) STR ],
  ( 0->1 ) with_no_heading ,
  ( 0->1 ) with_no_closing ,
  ( 0->1 ) each TUPLE,
  ( 0->1 ) each_time ASSERTION
)

This import says that Display is a PROC which has 6 keyword arguments. The with_title_line keyword takes a STR argument and may or may not appear in a Display call. The with_title_lines keyword takes a TUPLE of 0 or more STR values as its argument and may or may not appear in a Display call. The with_no_heading keyword takes no argument and may or may not appear in Display calls.

In general, the syntax for fpp imports (emphasizing those for FUNCTIONS) is:
Initializer is the keyword argument speciﬁcation used to specify all relevant information about an argument. Its NbrConstraint is used to specify how many times a keyword may appear in a call; if the NbrConstraint is omitted, then it is assumed to be 1 if no initializer is given and otherwise, 0->1. Following the NbrConstraint is the preposition (i.e., keyword), if any. If there is no keyword, then the argument is a positional argument whose position increases by 1 with each positional argument from the left. It is possible for a keyword to take no argument. However, assuming that there is a keyword argument is a positional argument whose position increases by 1 with each positional argument from the left. It is possible for a keyword to take no argument. However, assuming that there is a keyword argument speciﬁcation must appear after the positional ones. However for functions, either all the arguments are keyword or they are all positional.

The pivotal construct here is the ArgSlotDef which is used to specify all relevant information about an argument. Its NbrConstraint is used to specify how many times a keyword may appear in a call; if the NbrConstraint is omitted, then it is assumed to be 1 if no initializer is given and otherwise, 0->1. Following the NbrConstraint is the preposition (i.e., keyword), if any. If there is no keyword, then the argument is a positional argument whose position increases by 1 with each positional argument from the left. It is possible for a keyword to take no argument. However, assuming that there is a keyword or positional argument, the vblad and Type of the argument follows. Next, as needed or desired, the parameter variable that will receive the argument is identiﬁed. Finally, the initial or default value must be speciﬁed for those cases where a keyword or positional argument can be missing in the call. The default values must be constant expressions, i.e., while they can contain function calls, they cannot contain variables. Observe though that positional arguments can be omitted in the call only if they appear at the end of the argument TUPLE (and if the fpp deﬁnition allows for their absence). Any keyword argument speciﬁcations must appear after the positional ones. However for functions, either all the arguments are keyword or they are all positional.
The manifest `vbladj` is used to specify that the Type of the argument must be readily apparent from the call. For example, if the `ArgSlotDef` specifies a manifest INT, then the argument appearing in the call must be an INT constant like 1234 and not some expression like a function call. Likewise, if a manifest TUPLE is specified, then the argument appearing in the call must be a TUPLE pattern like `[ .x, 67, .yy-8 ]` and not some TUPLE-valued expression like `read( from _cmd_line_ )`.

Daytona currently supports `NbrConstraints` 1 and 0→1 only. Daytona does not currently support users defining Cymbal fpps that have no-argument keywords.

Analogous to variables, if a Cymbal fpp is said to be `C_external`, then Daytona will not modify its name when referencing it in the generated C code.

### 6.10.1.1 Kinds Of Parameter Variables

There are 3 kinds of parameter VBLS in Cymbal: constant, alias, and copy. The default is constant, which means that the fpp body is not allowed to change the value of the parameter VBL. On the other hand, both alias and copy parameter VBLS can be modified in the fpp body. A copy parameter VBL is one which contains a copy of the argument. An alias parameter VBL is one which is just a renamed version of the VBL that was passed in as an argument: alias parameter VBLS must always be passed pure VBLS as arguments: this typically involves omitting the dereferencing dot found so frequently when working with Cymbal VBLS. Here is a sample fpp import and call illustrating these possibilities:

```c
calls:
INT some_fun(
    STR(23) .w,
    constant STR .x,
    alias DATE .y,
    copy INT(_short_) .z )
local: DATE .d = `1-1-84`

set .ww = some_fun( .ss, "abc", d, 5 );
```

Both alias and copy parameter VBLS require that the type of the argument match that of the parameter exactly. Otherwise, regular assignment coercions are performed when needed.

alias can be used within Cymbal programs for any Cymbal type, including, for example, INTS, DATES, BOXES, and ARRAYS. However, for Cymbal fpps that will be called from C, alias ARRAY parameter VBLS can be STR(=) or STR(k) but not STR(*).

TUPLES and STRUCTS can be arguments to fpps but the corresponding parameter VBLS must be constant or alias parameters VBLS.

Cymbal also supports passing fpps as arguments to constant fpp parameter VBLS (fppvbl.?Q):
define INT FUN( INT .x ) handler
{
    return(.x);
}

define INT FUN( INT .y, INT FUN( INT .xx ) .z ) doit
{
    local: INT FUN( INT .xx ) .tmp_z

    set .tmp_z = .z;

    do Write_Line( .tmp_z(.y) );
    return( .z(.y) );
}

do Write_Line( doit( 25, handler ) );

Note that the second argument to doit is a FUNCTION, not a VBL or a TUPLE or an INT, but a FUNCTION, which is then applied in the return statement of doit.

6.10.1.2 More Fpp Import Examples

Here is an entire sequence of imports taken from the system environment file $DS_DIR/EXAMPLES/sys/sys.env.cy. Observe that collections of imports are inaugurated with an imports: or import: header.

imports:
    FLT( _long_ ) FUN
covar(
    over TUPLE[ VBLSPEC, VBLSPEC ] ,
    each_time ASN
)
PROC Write(
    to CHAN = _stdout_ ,
    ( 0->1 ) skipping INT,
    ( 0->1 ) with_sep STR,
    ( 0-> ) WRITABLE )

PREDICATE Syntax_Error[ CHAN = _prev_ ]
BOOLEAN FUN truth( ASN )
PRED Is_A_Date[ ALPHA .cand_date ]

A VBLSPEC is one of the VblSpecs discussed previously in the section on variable definitions. ASN is the same as ASSERTION. A WRITABLE is defined as the union of a number of ‘printable’ types; in fact, its definition is given in sys.env.cy:
define CLASS ALPHA = CLASS STR|LIT|RE|DATE|THING|_3GL_TEXT|TIME|CLOCK

define CLASS WRITABLE = CLASS ALPHA|INT|FLT|BOOL|TEXT

The import of Write is interesting because it illustrates how to declare a procedure that has both keyword and positional arguments. The only thing special about imports for predicates is that they use brackets instead of parentheses. Observe how the parameter variable is gratuitously included in the definition of Is_A_Date.

6.10.2 Defining Fpps

An fpp definition is just a define followed by an fpp import followed by a Do. Here is the definition of the Print_Date task taken out of validsupp.Q. It is not supposed to do anything particularly useful but it does illustrate the nesting of the definitions of the three helper fpps month(), Validate[], and Check_Date(). And one of its helpers does have a return command.

```plaintext
define PROCEDURE task : Print_Date( DATE .ddate )
do {
   local: DATE: .date2, .date3
   define STR FUN : month( DATE .date )
   {
      local: STR( 2 ): .ans
      do Check_Date( );
      set [ .ans ] =
         tokens( for (STR).date
               matching "[^-/.]*[-/.]\([^-/.]*\)" );
      return( .ans )
   }
   set .date2 = .ddate;
do Check_Date;
do Write_Line( "Date = ", .date2 );
do Write_Line( "Month = ", month(.ddate) );
when( !Validate[.ddate] ) do Write( "\ntragedy: no date\n" );
def PROC : Check_Date()
{
   when( .date2 Is_Not_A_Date ) do Write( "\ntragedy: no date\n" )
}
def PRED : Validate[ DATE .date ]
do {
   when( .date Is_A_Date ) return( _true_ );
   else return( _false_ )
}
```

Copyright © 2013 AT&T
All Rights Reserved.
September 15, 2013
6.10.2.1 Fpp Task Definitions

Tasks have an extra bit of syntax consisting of optional task adjectives followed by the task keyword, as illustrated in the following syntax for FUN tasks:

\[
\begin{array}{rcl}
\text{FunTaskDef} & ::= & \text{define} \ [ \ fppadj \ ] \ \text{Type} \ \text{FUN} \ [ \ taskadj \ ] \ \text{task} \ [ \ : \ ] \\
\text{TaskDo} & ::= & \text{do} \ [ \ ] \\
\text{begin} & | & \text{transaction} \\
\text{flush_on_return} & | & \text{close_on_return} \\
\text{on_abort_return} & | & \text{Subject} \\
\text{with_logging} & | & \text{with_no_logging} \ | \ \text{with_logging_optional} \\
\text{with_logging_flag} & | & \text{Subject} \\
\text{free_on_begin_when} & | & \text{Subject} \ | \ \text{free_on_return_when} & | & \text{Subject}
\end{array}
\]

The begin task adjective is used to identify a particular task as being the ‘Begin’ task, i.e., the analog of the C and PL/I main() routines. The transaction keyword causes the associated task to become a transaction; see Chapter 17 for more details on transactions. The argument to the on_abort_return keyword is the value that a FUNCTION or PREDICATE transaction task will return when the Abort PROC is called, as occurs when a signal is received. The logging keywords are also discussed in Chapter 17.

free_on_begin_when and free_on_return_when take BOOLEAN arguments which, if they evaluate to true, cause all exported and local dynamic storage associated with a task to be freed at the beginning or end of task invocations, respectively. These keywords enable the user to cause Daytona to reduce the size of its aggregate process memory usage by freeing up the space used by boxes, dynara, transaction Do_Queue, and string garbage either at the end of task execution or at the beginning. Note that imported boxes and dynara will be left alone. Also note that while the storage of dynamic atomically typed VBLs is always freed on task exit (because they are implemented solely on the stack), such is not the case for any boxes and dynara that the task defines, whether exported, static or local. The only way for that storage to be freed is to use these keywords. Furthermore, these keywords also support the removal of any transaction Do_Queue and other miscellaneous storage allocated during task execution.

For example, if a program had 100 database update tasks, all of which were eventually called, then the process image would eventually grow to include the total memory over each of the largest-over-time of the 100 Do_Queue. On the other hand, if free_on_return_when cleared out Do_Queue for each task as it ran, then the process image would never include more than the largest single Do_Queue for one of the tasks (as opposed to the union of all of them). The trade-off here is between memory usage and the time it takes to free and re-allocate dynamic storage on a per-task-invocation basis. The BOOLEAN arguments for these keywords enable the user to control which invocations free up the space: this enables the user to reuse cached box contents across several task invocations until finally they are of no further value and are released.
A useful paradigm for using this functionality is to create a free_on_return_when task which creates/exports a collection of boxes and dynara for export to several other tasks that it calls that work with these data structures. Then when the first one, the creator, exits, all of the associated exported (and local) storage is freed.

Note that this is all ‘freeing’ in the malloc sense; no such freeing goes on when this is executed:

```plaintext
set .mydynara = {}; set .mbox = [];
```

### 6.10.2.2 Fpp Tasks And Record Class Accesses

flush_on_return and close_on_return are opposites: flush_on_return tasks making record class accesses are faster because they keep those accesses open between invocations (and of course, reinitialize them for use on each subsequent invocation). On the other hand, if file descriptors are a rare commodity, then close_on_return tasks become attractive because they close all of their open database files before returning. Any record access opened by a helper fpp remains open for the duration of its parent task’s invocation.

### 6.10.2.3 Fpp Import And Definition Placement

Where then do VBL and fpp imports and definitions go, once written? For task definitions, imports and definitions of all varieties may precede or follow the CmdSeq appearing in the ‘body’ of the task definition. For helper fpp definitions, varieties are restricted to local variable definitions and to fpp imports. (Recall that Daytona automatically imports tasks (compiled with the same run of Tracy) into other tasks in addition to importing all helper fpps in the same task into each other.) Local variable definitions and fpp imports may also appear after the opening brace of a Do group; these are processed by relocating them to the start of the smallest, enclosing fpp.

### 6.10.2.4 Global Environment And User C-Extensions

In addition, VBL and fpp imports and definitions can be made global by putting them into the global environment. The system has many Cymbal VBL and fpp imports and definitions in the file $DS_DIR/sys.env.cy. Correspondingly, the user may place such in their file usr.env.cy, which should be placed in some directory in the path $DS_PATH. usr.env.cy may not contain VBL export or local statements. Likewise, if there are user Cymbal definitions and imports that are specific to a particular application, say, app in $DS_APPS, they may optionally be placed in the file app.env.cy, which will be found by Daytona given that it is locatable through $DS_PATH. Likewise again, if there are user-level Cymbal definitions and imports that are specific to a particular project, say, proj = $DS_PROJ, they may optionally be placed in the file proj.env.cy, which will be found by Daytonagiven that it is locatable through $DS_PATH. Note that app.env.cy and proj.env.cy are for the benefit of all who are using the respective application and project whereas the intent of usr.env.cy is that it contain the customizations proper to an individual user.

Cymbal fpp task definitions in these environment files will automatically be included as tasks in queries that call the fpps. Likewise, Cymbal helper fpp definitions in these environment files will automatically be included as helpers in any query task where they are called. Declarative
Definitions, such as those for macro and path PREDs, may also be placed in the environment files.

The imports serve to give Daytona the interface to C variables and functions; in this regard, the C\_external keyword should be used to keep Daytona from appending characters to the variable and function names. The corresponding C programs may be in the file usr.env.c in some directory in $DS\_PATH or in the file app.env.c for some application app in $DS\_APPS or in the file proj.env.c for $DS\_PROJ = proj.

In short, all the user has to do to extend the system with their own C fpps is to declare or import them in Cymbal using C\_external in some *env.cy (or explicitly in particular queries) and to instruct Daytona to have the C compiler link in the corresponding C object modules. This latter can be done as simply as by placing the C code in usr.env.c or widget.env.c for some application or project widget or by including the relevant instructions for finding/building the C code in a MAKE\_GOODS description in their pjd or apd as described in Chapter 3. All the rest is handled by Daytona.

Other examples of task definitions may be found relative to $DS\_DIR/EXAMPLES/ in sys/*cy, usr/orders/usr.env.cy, and usr/orders/Q/validsupp.Q.

6.11 Defining CLASSes

The file $DS\_DIR/sys.env.cy also contains the definitions of the atomic CLASSes that the system uses. Here are a couple basic ones:

```cy
define CLASS INT(?)
    with_default_specifier ^_long_
    with_default_interval_by_arg ^1^n
    with_c_init_val ^0^n
    with_cy_init_val ^0^n
define CLASS INT(_long_)
    with_c_type ^Int32^n
    with_min_val -2147483647
    with_max_val 2147483647
    with_dummy_val -2147483646
```

The first definition specifies properties which all varieties of INTs have in common or else what those properties are by default. The second definition specifies properties of a particular variety of INT, the INT(_long_). While these definitions may occasionally be of interest to the user, if only to enumerate all of the types that are available, they are mainly intended for the use of the system itself. Tracy reads $DS\_DIR/sys.env.cy into an internal Cymbal-description form which it consults as a dictionary to look up the bits of information it needs in order to process queries. That’s why the information in $DS\_DIR/sys.env.cy has to be correct because, ruling out vestiges, there is some query in the test suite that needs sys.env.cy to be correct in order for Tracy to process the query.

While Daytona does not support the user defining new CLASSes by writing their own definitions in this way (because there is much, much more involved in implementing a new CLASS than what appears in sys.env.cy), there are two class definitions available for the convenience of the user.

The first is to define an atomic CLASS in terms of another CLASS, usually not atomic. This is the Cymbal analog of a C typedef. The following examples should make clear what the required syntax is...
and how useful this capability is (classdef.1.Q):

```
define CLASS MYSTR = STR(25)
define CLASS BIN_TUPLE
    = TUPLE[ UINT(_huge_), STR, STR, STR(1), STR, STR(1) ];
define CLASS MY_SET = SET{TUPLE[(2) INT, FLT, (3) STR]
    : with_reverse_lexico_order with_sort_spec[5]}
define CLASS TU_TYPE_2  // the complete defn is in usr.env.cy
local:
    TU_TYPE_2 .tu
    MY_SET .mys
    STR .xyx = "YXY"
    MYSTR .zzz
set .zzz = (MYSTR).xyx;
```

And here is the official syntax:

```
ClassDef ::= define CLASS upper = with_symbols Bunch |
             define CLASS upper I = Type IP |
             define CLASS upper subclass_of upper ;
define ::= define | def ;
```

The first one is used in sys.env.cy for defining enumerated types but cannot be used by the user at this

time. It is the second two that are under discussion here. The apparently incomplete definition of
TU_TYPE_2 is required for the parser to successfully parse this query (by being notified that
TU_TYPE_2 is a CLASS name). The way to think of it is as a (partial) declaration of the CLASS name

with the actual definition appearing elsewhere. The actual complete definition is in one of the
*.env.cy files that the user maintains. Alternatively, it is sufficient for a complete CLASS definition to appear
before it is used later in the same query file.

As another convenience, the user may wish to define query variables that have the same type as
RECORD_CLASS FIELDS but without having to look up possibly changing FIELD definitions in the
aars. To that end, the user can use the `typed_like` keyword as illustrated by typed_like.[12].Q:

```
declare CLASS PART subclass_of RECORD
local:
    typed_like PART.Weight .x // which is a FLT
set .x = 44;
_Show_Exp_To(.x)
```

Once again, the (incomplete) CLASS declaration is needed to inform the Cymbal parser that ‘PART’ is a
RECORD_CLASS (defined by an rcd some place else) and not something else. The complete definition

of course is the rcd!
6.12 Exception Handling

There are occasions when writing a block of procedural code that the programmer becomes acutely aware that not only can many kinds of things go wrong in many places therein but also that untoward events can occur during the execution of functions that are called from within a stack of other function calls. In these situations, it can be very tedious to co-locate (possibly highly similar) error-handling code with the code that detects the problem and in the case of nested function calls, it could be that the best place to handle (or continue to handle) any error is in the code that called the first/top function in the nest because that would be where the best context for identifying and handling the problem would be located. Cymbal offers an exception handling capability to mitigate these problems by allowing the user to group and place error handling code in places and/or files potentially quite far away lexically and computationally from the code that would be generating the errors. This is accomplished by using try-else blocks as illustrated by the following taken from try.1.Q:

```cymbal
try {
    set .x = date_for_str( .dt );
} else {
    when( .last_exception = "date_for_str" )
    {
        do Write_Line( "Probably bad DATE format" );
    } else { // catch-all
        do Write_Line( "Caught .last_exception exception"ISTR );
    }
}
```

The code which will be generating the exceptions is located in the try block whereas the code that will be branched to handle any exceptions is encapsulated by the else block. The semantics is that when a try-else block is encountered, the code in the try block will be executed up until the first exception is raised (if any) and only in that event will the code in the else block be executed. When executing the else block, the value of the last_exception variable is a STRING(*) which identifies the current exception. Having executed the entire else block, execution continues with the first statement following the else block. try-else blocks must be written so that if there is any branch of execution out of the else block (as perhaps done by a goto), then it must be backwards to do a re-try as explained below. Otherwise, Daytona must be allowed to execute without interruption past the end of the else block. .last_exception is set to the empty string upon execution leaving the try-else block.

There are two ways to set .last_exception. In the first case, for built-in Daytona fpps, Daytona sets it for common errors that may occur in common built-in fpps. Such errors will result in program termination unless they are caught by this exception handling mechanism. (If users find that a built-in fpp is failing without setting .last_exception, then they are encouraged to report that to the Daytona development team.) The second way is to set .last_exception is to do so in the process of explicitly raising an exception in user Cymbal code.

By calling the Raise PROC, the user can raise their own exceptions identifying them by means of implicitly setting .last_exception to be whatever STRING they wish:
import: PROC( STR .exception ) Raise

Here is an artificial but instructive example of how that is done:

```csharp
define PROC User_Proc
{
    do Raise( "user condition 2" );
}
try {
    do User_Proc;
} else {
    when( .last_exception = "user condition 2" )
    {
        do Write_Line( "User Condition 2 has been raised" );
    } else { // catch-all
        do Write_Line( "Caught .last_exception exception"ISTR );
    }
}
```

Here a user PROC named User_Proc is considered to be executing potentially arbitrary code on behalf of the caller and if it runs into trouble, then the Cymbal code will use Raise to raise an exception (identified by its STR argument) for the caller to handle.

Here is an example of writing a try-else block to retry code up to 3 times (try.1.Q):

```csharp
{ local: INT .times_gone_by = 0
try_again:
try {
    do Raise( "user condition 4" );
} else {
    when( .last_exception = "user condition 4" )
    {
        do Write_Line( "Caught .last_exception exception"ISTR );
    } set .times_gone_by++;
    when( .times_gone_by <= 3 ) go try_again;
}
}
7. Various Built-in Functions, Predicates, And Procedures

Daytona’s other built-in functions, predicates, and procedures (fpps) are clustered into several major groupings: arithmetic, string, date/time, UNIX interaction, datatype conversion, and miscellany. Imports for these built-in functions are given in $DS_DIR/EXAMPLES/sys/sys.env.cy$.

7.1 INT, UINT and FLT Functions

There are, of course, the usual infix arithmetic operators $+$, $-$, $\ast$, $/$, $\%$, and $\ll$ (aka $\text{pow}$) which obey the customary precedences and associativities. ($x \% y$ is the integer remainder of dividing $x$ by $y$ and $\text{pow}(x, 4)$ is $x$ raised to the 4th power).

Special Note: $\%$ is not the remainder (AKA "modulus") operator from C! Instead, it is the infix form of the Cymbal $\text{mod\_int}$ (or $\text{mod\_hg}$) function which itself computes what is called the common residue of a given integer for a given integer modulus. Not incidentally, there is an overloaded $\text{mod}$ which resolves on INT(?) arguments to $\text{mod\_int}$ or $\text{mod\_hg}$, as appropriate. Users who wish to call the C modulus function from Cymbal need to call the Cymbal function $\text{cmod}$ by that name (i.e., there is no corresponding infix operator). When the first argument is positive, mod and cmod agree. Otherwise, they generally don’t; indeed when they don’t agree, the mod value is positive (as it always is) and the cmod value is negative (and different in magnitude).

So, how are these two functions defined? The mod or common residue of two integer arguments, say, $\text{mod}(a, m)$, is the canonical element of an equivalence class of integers defined by the following equivalence relation which defines what is means for $a$ to be congruent to $b$ mod $m$:

$$a \equiv b \mod m \iff a - b = k \times m$$

for some integer $k$.

The canonical element, i.e., the common residue, is always such that $0 \leq \text{mod}(a, m) < m$. So, $\text{mod}(a, m)$ is such that $a - \text{mod}(a, m)$ is an integer multiple of $m$. This notion of congruence was introduced by Leonhard Euler circa 1750. It has many nice properties, a few of which are used by Cymbal’s $\text{mod\_time}$.

On the other hand, the $\text{cmod}$ function is defined in the C language to always evaluate as follows on integers:

$$\text{cmod}(a, m) = a - \lfloor a / m \rfloor \times m$$

They are what they are. The world would be too simple without two mod functions.

FLT(_long_) FUN: $\text{mod\_flt}(\text{FLT}\_a, \text{FLT}\_m)$

FLT(_short_) FUN: $\text{mod\_sgl}(\text{FLT(_short_)}\_a, \text{FLT(_short_)}\_m)$

This overloaded $\text{mod}$ function resolves to either $\text{mod\_flt}$ or $\text{mod\_sgl}$. The return value of either is $a - b \times \lfloor a / b \rfloor$. This turns out to agree with what $\text{mod\_int}$ returns when INTs are used instead. Remember that the $\%$ operator will resolve to one of these functions as appropriate.
The following arithmetic functions are available courtesy of the Section 3M <math.h> C library.

\[
\begin{align*}
\text{INT|FLT ( \_long\_ ) FUN:} & \ \text{minus( INT|FLT ), abs( INT|FLT ), sign( INT|FLT\_STR )} \\
\text{FLT ( \_long\_ ) FUN:} & \ \sqrt{\text{FLT}}, \log(\text{FLT}), \exp(\text{FLT}), \\
& \ \sin(\text{FLT}), \cos(\text{FLT}), \tan(\text{FLT}) \\
\text{FLT ( \_long\_ ) FUN:} & \ \text{floor( FLT ), ceil( FLT )} \\
\text{FLT FUN:} & \ \text{pow( FLT, FLT )}
\end{align*}
\]

\text{pow} takes the first argument to the power of the second argument so that \text{pow}( 2, 3 ) = 8. The infix operator for exponentiation is \text{\textasciicircum}.

Here are the infix operators corresponding to the bitwise operations on INTS and UINTS:

\[
\begin{align*}
\neg \quad \text{one’s complement (unary operator)} \\
\sim \quad \text{relative complement (binary operator)} \\
\text{\textasciicircum} \quad \text{bitwise or} \\
\text{\textasciicircum}\text{\textasciicircum} \quad \text{bitwise and} \\
\% \quad \text{bitwise exclusive or} \\
<< \quad \text{left shift} \\
>> \quad \text{right shift}
\end{align*}
\]

\text{INT ( \_long\_ ) FUN:} \text{rand_int}( \text{INT .int_max = \_prev\_}, \text{INT .seed = \_next\_})

returns a random INT from within the range \([ 0 -> \text{first_argument} ]\). The default value for the first argument is the one specified for the first argument by the most recent call to \text{rand_int}. The seed for the random number generator can be specified by the second argument; it defaults to the next one in the pseudo-random sequence.

\text{FLT ( \_long\_ ) FUN:} \text{rand_uni}( \text{INT .seed = \_next\_})

returns a random FLT from within the open interval from 0 to 1. The seed for the random number generator can be specified by the first argument.

overloaded \text{UINT FUN( STR(\?) \_str, UINT .seed = 0 \_) strhash}

returns a random UINT based on its first argument. Its optional second argument is seed value for the hashing.

\text{INT FUN:} \text{int_for_octal_str( STR )}

returns the INT corresponding to a octal string representation of an INT, as in "02351".

\text{INT FUN:} \text{int_for_hex_str( STR )}
returns the INT corresponding to a hex string representation of an INT, as in "0xA34D".

INT FUN: int_for_binary_str( STR )
returns the INT corresponding to a binary string representation of an INT, as in "01110001100".

UINT FUN: uint_for_binary_str( STR )
returns the UINT corresponding to a binary string representation of a UINT, as in "01110001100".

STR FUN: octal_str_for_int( INT )
returns the octal string representation of an INT.

STR FUN: hex_str_for_int( INT )
returns the hex string representation of an INT.

STR FUN: binary_str_for_int( INT )
returns the binary string representation of an INT.

There are corresponding functions for handling INT(_huge_). Look for the likes of hex_str_for_hg and hg_for_hex_str in sys.env.cy.

### 7.2 String Handling Functions

#### 7.2.1 Lexical Analysis By FUN tokens

tokens is one of Daytona’s most powerful string handling functions. It provides for the arbitrary regular expression lexical analysis of strings by using the regular expression matching capabilities of egrep(1). Here is its prototype:

```python
import:
LIST[ ( 1->9 ) STR] FUN: tokens(
for STR,
matching STR|RE|CRE,
( 0->1 ) but_if_absent TUPLE[ ( 1->9 ) STR]
)
```

This import means that tokens returns a LIST of from one to 9 STRs as a result of using the matching pattern argument to split up the for argument into tokens. Here is an example:
set [ .x, .y, .z ] =
  tokens( for "Tunel, Jerry (x3256)"
    matching "\([\^,]*\), *\([\^ ]*\) *(x\([\^]\)*\))" )

Each `tokens` call uses the pattern in the `matching` argument to extract one or more lexical tokens from the `for` argument. Each token pattern is an RE delimited by `''` and `''` (```). The generation of token patterns is done by matching the entire pattern (ignoring the token pattern delimiters) on the `for` string and then naming the appropriate sub-strings by the corresponding token pattern index, i.e., $1$ for the first sub-string matched, $2$ for the second, and so on up to $9$ total sub-patterns. These sub-strings are then given as values to the corresponding Cymbal variables. Regarding the example above, after execution, $y$ has gotten $2$'s value which is "Jerry".

The regular expressions used are essentially but not exactly the ones specified for regexp(5) or egrep(1) and are defined as follows:

- Matches any single character except new-line.

- Matches the start of the string.

$ Matches the end of the string; \n matches a new-line.

[ ] A bracketed nonempty string $s$, i.e., [ $s $], ( or \[ s \] ) matches any character in (or not in) $s$. In $s$, \ has no special meaning and ] may only appear as the first character where it is taken literally with no special meaning. In $s$, if – appears first, it is taken literally with no special meaning. If necessary for both ] and – to be included in the character class, have them appear first in $s$ in that order. $s$ may contain occurrences of \[dlsw] or of the backslash quoted C escapes including octal escapes (see below).

- Within brackets, the minus means through. For example, [a-z] is equivalent to [abcd...xyz]. The – can appear as itself only if used as the first character, unless ] or ^ is the first character. For example, the character class expression [ ]−[ ] matches the characters ] and – and [− ] matches either a blank or a – but [ − ] is illegal.

* A regular expression followed by * means that regular expression appears zero or more times. If there is any choice, the longest leftmost string that permits a match is chosen.

+ A regular expression followed by + means that regular expression appears one or more times. If there is any choice, the longest leftmost string that permits a match is chosen. For example, [0-9]+ is equivalent to [0-9][0-9]*.

? A regular expression followed by ? means that regular expression appears zero or one times.

| Two regular expression separated by | means that exactly one of those regular expression appears.
These escaped parentheses are used for grouping pattern sequences into single patterns. An operator, e.g., *, or +, can work on a single character or a regular expression enclosed in escaped parentheses. For example, \((a\ast\((cb+\ast)\))\). Observe that groups can be nested and that they are numbered in the occurrence of their initial boundary marker from left to right.

\$n
This refers to the \textit{nth} grouped expression. $n$ is used instead of the more traditional \textit{\textbackslash n} so that users can use unambiguously use octal escapes in their regular expressions. \textit{n} ranges from 1 through 9.

\&
This refers to the entire expression that was matched. It is used in primarily in \textit{substit} calls.

An RE matching one character (but not using \textit{\textbackslash(\textbackslash)\textbackslash}) and followed by \textit{\textbackslash[m\textbackslash]}, \textit{\textbackslash[m,m\textbackslash]}, or \textit{\textbackslash[m,m,n\textbackslash]} is a RE that matches a range of occurrences of the one-character-matching RE. The values of \textit{m} and \textit{n} must be non-negative integers. \textit{m} matches exactly \textit{m} occurrences; \textit{m}, matches at least \textit{m} occurrences; \textit{m},\textit{n} matches any number of occurrences between \textit{m} and \textit{n} inclusive. Whenever a choice exists, the RE matches as many occurrences as possible. See re.mn.[12].Q for examples.

\d
This is shorthand for a decimal digit between 0 and 9.

\D
This is shorthand for any character but a decimal digit between 0 and 9.

\l
This is shorthand for a lower- or upper-case letter.

\L
This is shorthand for any character but a lower- or upper-case letter.

\s
This is shorthand for a space, \textit{\textbackslash t}, \textit{\textbackslash n}, \textit{\textbackslash f}, \textit{\textbackslash r}, \textit{\textbackslash b}.

\S
This is shorthand for any character but a space, \textit{\textbackslash t}, \textit{\textbackslash n}, \textit{\textbackslash f}, \textit{\textbackslash r}, \textit{\textbackslash b}.

\w
This is shorthand for a lower- or upper-case letter, digit or underscore.

\W
This is shorthand for any character but a lower- or upper-case letter, digit or underscore.

By necessity, the characters \texttt{.\ast\&?\^{}[]$} are special. They must, therefore, be escaped with a \textbackslash (backslash) to be used as themselves. Otherwise, the usual C backslash escapes \textbackslash n, \textbackslash t, \textbackslash v, \textbackslash b, \textbackslash r, \textbackslash f, and \textbackslash a are honored as well as the octal escapes such as \textbackslash 101.

Here is another example illustrating how \textbackslash s and nested groups can take apart Cymbal Multiplicity specifications consisting of the likes of 0, 1->, 0->1, and 10 -> 20. (A smaller pattern will also do the trick; this one was chosen for pedagogical reasons.)

```
set [ .single, .entire, .first, .second ] =
tokens( for "10 -> 20"
       matching "\((0-9)+\)\((0-9)+\) \*-> *\((0-9)+\)?\)" )
```

In this case, the value of \textit{single} is \\texttt{""}, the value of \textit{entire} is \textit{10} \texttt{\textasciitilde} \texttt{20}, the value of \textit{first} is \textit{10} and the value of \textit{second} is \textit{20}.
If the match fails, then no variable values are changed unless a \textit{but if absent} argument is present, in which case, the values that it specifies are used as the \textit{TUPLE} returned by \textit{tokens}. In the procedural setting, whether \textit{tokens()} matched its RE or not is faithfully reported by the predicate \textit{RE\_Match\_Worked} and its opposite \textit{RE\_Match\_Failed}. The \textit{scanf} capabilities of Read enable it to do simple, non-RE-based lexical analysis. \textit{tokens} can also be used in declarative Cymbal assertions. The \textit{matching} argument is considered to be an \textit{RE} even if it comes in as a \textit{STR}. Of course, an \textit{RE} or \textit{CRE} could be used as well.

\subsection*{7.2.2 String Substitution}

Daytona’s string substitution capabilities are quite powerful as the following program shows.

\begin{verbatim}
import: /* automatically included from sys.env.cy in user queries */
STR FUN: substi( this STR, for STR|RE|CRE, in STR, max_times INT = 1 )
STR FUN: gsubsti( this STR, for STR|RE|CRE, in STR, max_times INT = _all_ )

{ local: STR: .new_str
  set .new_str = substi( this "ZZ" for "c" in "abc.def" );
  _Say_Eq( .new_str, "abZZ.def" );

  set .new_str = substi( this "ZZ" for "." in "abc.def" );
  _Say_Eq( .new_str, "abcZZdef" );

  set .new_str = substi( this "ZZ" for ".RE in "aaa.def" max_times 2 );
  _Say_Eq( .new_str, "ZZZZa.def" );

  set .new_str = gsubsti( this "ZZ" for "RE in "aaa.def" );
  _Say_Eq( .new_str, "ZZZZZZZZZZZZ" );

  set .new_str = substi( this "$ZZ$1 != &" for "\(...\).\(...\)"RE in "abc.def" );
  _Say_Eq( .new_str, "defZZabc != abc.def" );

  when( _nbr_substi_ = 0 ) do Exclaim_Line( "error: no substitution made" );
}
\end{verbatim}

The program above contains a number of calls to \textit{substi} as well as output statements that (correctly) claim that the answer is equal to some \textit{STR} constant. (The output statements make use of the \textit{Say\_Eq} macro defined in \texttt{sys.macros.m}). The first two examples should be self-explanatory. Of course, in all these examples, general terms could be used instead of \textit{STR} constants.

Nota bene: Only when the \textit{for} argument is an \textit{RE} or \textit{CRE} do any of its characters have special meaning (beyond of course the usual C backslash escapes). When an \textit{RE} is used though, Daytona searches the \textit{in} \textit{STR} for a substring that matches the \textit{RE}. That substring, called the doomed guy, is the string that will be replaced by the \textit{this} argument. When more than one substitution is requested, regular expression pattern matching occurs successively, which is to say that the regular expression is searched for again and again as required in the currently unmatched portion of the source string. Observed that this is what happened in the third and fourth examples. If there is no match then \textit{substi} returns its unchanged \textit{in} argument.
The same REs usable in tokens are usable here. This allows for some very sophisticated substitutions as illustrated by the last example where substrings of the doomed guy are included in the replacement. In fact, as represented by &, the entire doomed guy is a substring of its own replacement.

Finally, after any substi command, _nbr_substi_ is set to the number of substitutions made.

There is much more on tokens in Chapter 10.

### 7.2.3 String Handling Miscellany

Here are some less sophisticated but nonetheless useful string handling fpps.

```pascal
~> or PRED: Matches [ STR, STR|RE|CRE,
( 0->1 ) with_match_start_index_vbl alias INT = re_match_start_index,
( 0->1 ) with_match_length_vbl alias INT = re_match_length ]

is_true if and only if the first argument matches the pattern specified by the second argument. If the second argument is an STR, it is considered to be an RE anyway. The predicate RE_Match_Worked is also defined appropriately. The infix operator for Matches is `~>`. The user may supply INT VBLS for either one or both of the ancillary variable keywords with_match_start_index_vbl and with_match_length_vbl. In that event, as a side effect of evaluating the Matches predicate, the value of any with_match_start_index_vbl VBL argument will be set equal to the integer index of the first character of the matched string, if any, and to 0, if there is no match. Similarly, the value of any with_match_length_vbl VBL argument will be set to the length of the string matched, if any, or else to -1. Here is an example:

```pascal
{ local: INT VBL match_start_index, match_length
  when( "abcabcdefghi" ~> "def"
    with_match_start_index_vbl match_start_index
    with_match_length_vbl match_length )
  {
    do Write_Words( .match_start_index, ",", 7 );
    do Write_Words( .match_length, ",", 3 );
  }
}
```

CRE FUN: compiled_re( STR )

returns the compiled RE version of the argument. CREs can never be printed. This function is only rarely needed since Daytona pre-compiles REs at compile-time whenever possible.

PRED: Is_A_Substr_Of[ STR, STR ]

is_true if and only if the first argument is a substring of the second argument.

Copyright © 2013 AT&T
All Rights Reserved.
September 15, 2013
PRED: Contains[ STR, STR ]

is_true if and only if the second argument is a substring of the first argument.

or
PRED: Str_Starts_Str[ STR, STR ]

is_true if and only if the first argument is an initial subsequence of the second argument. In
the case of IP(_heko_), the subsequence is determined by whole octets.

overloaded PRED : Ends [ STR|HEKA, STR|HEKA ]
or
PRED: Str_Ends_Str[ STR, STR ]

is_true if and only if the first argument is a terminating subsequence of the second argument.

STR FUN: substr( STR, INT, INT = _right_ )

returns a substring of the first argument which, using emacs terminology, is the string of
characters between the mark character and the point character, inclusive. In what follows, let j
and k be positive integers. The mark character is the character at the position indicated by the
second argument. Suppose the second argument is j, then the mark character is the jth character
beginning from the start of the first argument; if the second argument is −j, then the mark
character is the jth character beginning from the end of the first argument. Suppose the third
argument is k. Then, the point character is determined from the mark character by going k−1
characters to the right. If the third argument is −k, then the point character is determined from
the mark character by going k−1 characters to the left. If the third argument is _right_ (_left_),
then the point character is the farthest to the right (left) from the mark character. So, the
following three assertions are true:

\[
\begin{align*}
\text{substr}( \text{"abcde"}, 2, 2 ) &= \text{"bc"} \\
\text{substr}( \text{"abcde"}, -2, 2 ) &= \text{"de"} \\
\text{substr}( \text{"abcde"}, -2, -2 ) &= \text{"cd"}
\end{align*}
\]

INT FUN: index( of STR, in STR, at_occ INT = 1 )

returns the integer position of the occurrence numbered by the third argument of the first
argument in the second argument. If the at_occ argument is negative, then the search begins at
the end of the in string and goes towards the beginning. The function returns 0 if the requested
occurrence cannot be found. For example,
INT FUN: index_of_first_unequal_char(STR, STR)
returns the integer position of the first character where its two arguments differ. It returns 0 if its two arguments are equal.

INT FUN strlen(STR(?)|LIT(?)|RE(?))
returns the length of its argument, which can be any of the STR/LIT/RE types. This computed by the C strlen function which counts each character in the OBJECT.

INT FUN length(STR(∗)|RE(∗)|LIT(∗))
returns the length of its argument. While length() is actually overloaded, its appearance here highlights its use on STR(∗)|RE(∗)|LIT(∗) where it is computed by returning just the system-stored value of the length, which is clearly faster than counting the number of characters in the OBJECT as strlen() would do.

+ or STR FUN: plus(STR, STR)
returns the result of concatenating its arguments.

STR FUN: concat(manifest TUPLE[(1->)OK_OBJ], STR .sep = "")
returns the result of concatenating the WRITABLE versions of the elements of its TUPLE argument. Here is a sample call:
concat( ["His address is", .addr, "and his age is", .age ] )
The understanding is that any element of the TUPLE will be cast to a STRING if it is not already one. If the second argument to concat is present, then it will separate the WRITABLE versions of the TUPLE elements. When given more than two arguments, concat is more efficient at doing its job than a corresponding number of plus_str calls is, in large part because it minimizes using the string garbage collection system. Thus, ISTR are more efficient because they are implemented using concat.

Actually, the requirement that concat’s first argument be a manifest TUPLE is a little misleading in that it needs to be manifest after it has been expanded as much as Tracy can expand it. So, for example, since Tracy can (and does) expand TUPLES and conventional
ARRAYS, the following query (concat.1.Q) illustrates another acceptable way to use \textit{concat}:

\begin{verbatim}
{ local:
  INT .a = 23, .b = 45, .c = 67, .d = 89
  TUPLE[ INT, INT, INT, INT ] .tuu
  INT ARRAY[4] .araa

  set .tuu = [ .a, .b, .c, .d ];
  do Write_Line( concat( .tuu, "." ) );
  set .araa = [ .a, .b, .c, .d ];
  do Write_Line( concat( .araa, "." ) );
}
\end{verbatim}

The use of \textit{concat} can also be facilitated by the splicing operator \%:

\texttt{concat( [ %.myara, %.mytuple ], "." )}

\begin{verbatim}
set .x += <string-like-obj>
\end{verbatim}

Obviously, this assignment is not a function call but it does illustrate an efficient way to grow the contents of a VBL whose type is one of \texttt{STR|LIT|RE|HEKSTR|CMD|SHELLP|SAFE\_STR}. By means of this construct, the RHS is appended to the contents of the LHS VBL in a particularly efficient way that avoids the garbage collection system.

\begin{itemize}
  \item or \texttt{STR FUN: sub( STR, STR|RE|CRE )}
  \end{itemize}

returns the result of removing the first occurrence of an STR second argument from a copy of the first argument. If the second argument is an RE or CRE, then it is the substring that is matched by the pattern that is removed.

\begin{itemize}
  \item or \texttt{STR FUN: mult( INT, STR )}
  \end{itemize}

returns the result of concatenating the second argument with itself for the number of times specified by the first argument.

\begin{itemize}
  \item or \texttt{INT FUN: div( STR, STR|RE|CRE )}
  \end{itemize}

returns the number of times that the second argument appears in the first argument. If the second argument is an RE or CRE, then it is the substring that is matched by the pattern whose occurrences are counted.

\begin{itemize}
  \item or \texttt{STR FUN: mod( STR, STR|RE|CRE )}
  \end{itemize}

returns the result of removing all occurrences of an STR second argument from a copy of the
first argument. This use of the overloaded `mod` resolves to `mod_str`, `mod_re`, or `mod_cre`, as appropriate. If the second argument is an RE or CRE, then it is the substrings that are matched by the pattern whose occurrences are removed. Remember that the `%` operator will resolve to one of these functions as appropriate.

+ or STR FUN: `plus_str_int( STR .base_str, INT .offset )`

returns the STR that begins at `.offset` into `.base_str`. Giving a negative offset or an offset that is longer than the base string will result in erroneous results possibly including segmentation violation. Such misuses can be caught by giving the -SOC option to Tracy.

STR FUN: `copy_of_str( STR )`

returns a copy of its argument. This function is only rarely needed since Daytona both copies strings and performs garbage collection when needed. So, unless the user understands fully the implications of using this function, they should not use it; the penalty of inappropriate use is a memory leak because `copy_of_str()` results are not garbage collected. See the discussion on STR(=) in Chapter 6.

PRED: `Is_A_Digit_Str[ STR\LIT ]`
PRED: `Is_An_Int_Str[ STR\LIT ]`
PRED: `Is_A_Flt_Str[ STR\LIT ]`
PRED: `Is_An_Int_Or_Flt_Str[ STR\LIT ]`
PRED: `Is_A_Decimal_Str[ STR\LIT ]`
PRED: `Is_A_Date[ STR\LIT ]`
PRED: `Is_A_Lower[ STR\LIT ]`
PRED: `Is_An_Uplow[ STR\LIT ]`
PRED: `Is_An_Upper[ STR\LIT ]`
STR FUN: `lower_of( STR\LIT )`
STR FUN: `upper_of( STR\LIT )`
STR FUN: `uplow_of( STR\LIT )`

The predicates test for membership in the indicated classes. Note that INTS really different from FLOATS, even though one expects an INT to be automatically converted to a FLOAT in many situations. At any rate, `Is_An_Int_Or_Flt_Str` characterizes the syntax of both. `Is_A_Decimal_Str` characterizes numbers using decimal points or else just INTS; here are some examples: 123, -123., 1234.56, and 1,234,567.89. Note that the use of commas as thousands-separators is supported. The FUNCTIONS convert a copy of their argument to a member of the indicated class with the exception that non-alphanumerics are ignored during this conversion process. So, if need be, the corresponding predicate can be used to test whether the result of the function is truly a member of the class or not. These functions are also known as `uplow_for_str()`, etc.
STR FUN( STR ) blank_out_white
returns the result of converting all occurrences of blanks, tabs, new-lines, carriage-returns, formfeeds, and vertical tabs (i.e., C isspace characters) to blanks in a copy of its argument.

STR FUN( STR ) blank_out_punct
returns the result of converting all occurrences of punctuation (i.e., C ispunct characters) to blanks in a copy of its argument.

STR FUN( STR ) condense_blanks
returns the result of removing all leading and trailing blanks from a copy of its argument as well as reducing all interior subsequences of blanks to length one.

STR FUN( STR, STR .fill_ch = " ") wipe_out_punct
returns the result of converting all occurrences of punctuation (i.e., C ispunct characters except for underscore) to the character .fill_ch in a copy of the first argument. The string .fill_ch must have length 0 or 1; if it is the empty string, then punctuation is elided completely from the first argument.

STR FUN( STR, INT .where_from = _start_and_end_, STR .doomed_chars = " ") trim
returns the result of removing either from _start_, _end_, or _start_and_end_ all occurrences (if any) of the .doomed_chars from the first argument.

STR FUN( STR .tgt, STR .doomed_chars, STR .fill_ch = " ") wipe_out_these_chars
returns the result of converting all occurrences of any characters in the .doomed_chars STR argument to the character .fill_ch in a copy of the first argument. The string .fill_ch must have length 0 or 1; if it is the empty string, then any doomed characters are elided completely from the first argument.

STR FUN( STR, STR .fill_ch = " ") wipe_out_nonascii
returns the result of converting all occurrences of non-ASCII characters (i.e., bytes with their 8th bit set) to the character .fill_ch in a copy of the first argument. The string .fill_ch must have length 0 or 1; if it is the empty string, then non-ASCII characters are elided completely from the first argument.

overloaded STR(*) FUN( in STR(*) VBL | STR(*) .tgt, from STR(=) .f, to STR(=) .t = "", (0->1) with manifest TUPLE[ (0->) INT ] = [ ],
(0->1) \texttt{with

\texttt{\_translated\_vbl alias INT = nbr\_translated \) translate

\texttt{// 'with' options: \_delete\_ \_complement\_ \_squash\_}

\texttt{translate} is a general function which subsumes several of the preceding functions such as \texttt{blank\_out\_punct}. The \texttt{translate} function is the Cymbal implementation/version of the Perl "tr" function. (Note: The functionality, while similar, is not identical to the UNIX "tr" command.) Invoked without the optional arguments, this function will transliterate each character found in the \texttt{in} string that matches a character in the \texttt{from} string for the positionally corresponding character in the \texttt{to} string. However, in contrast to Perl, the NUL character (\texttt{\000}) cannot be considered to be part of any of these strings because, obviously, Cymbal considers NUL to represent the end of the string.

A character range or subsequence of the ASCII collating sequence may be specified with a hyphen (-), e.g., "A-F", "x-z", "0-9", and "\200-\377". In a character range, the starting character must precede the end character in the current collation order, or else a runtime error will occur. To avoid ambiguity when desiring to translate the hyphen character itself, specify it first or last in the string. Any transliteration for a character after the first specified for that character is ignored, e.g.:

\begin{verbatim}
set .laugh = translate(from "AAA" to "OPQ" in "HAHAHA");
\end{verbatim}

will translate all instances of "A" to "O", resulting in "HOHOHO". The full set of Cymbal-supported 8-bit characters are allowed. If there are fewer characters in the \texttt{to} string than the \texttt{from} string, then the last character in the \texttt{to} string is considered to be replicated in the \texttt{to} string until they are the same length -- \textit{unless \_delete\_ has been specified in which case the \texttt{to} string is left exactly as the user wrote it (see below for more on \_delete\_)}. If the \texttt{to} string is empty, then the \texttt{to} string is considered to be the same as the \texttt{from} string. (This idiom can be used in conjunction with the \texttt{with\_nbr\_translated\_vbl} argument to compute the number of times characters from the \texttt{from} argument appear in the \texttt{in} argument.)

One or more \texttt{with TUPLE} components may be specified, and they have the following effects.

\begin{itemize}
\item If the \_delete\_ TUPLE component is used, then any \texttt{in}-string character found in the \texttt{from} string without a corresponding character in the \texttt{to} string is deleted.
\item If the \_complement\_ TUPLE component is used, then the \texttt{from} string character set is complemented (i.e., all characters not in the \texttt{from} string are used instead in the current collation order). This is most useful when used in conjunction with \_delete\_ (or \_squash\_) and an empty \texttt{to} argument because then every character except what is in the original/given \texttt{from} string will be deleted (or squashed).
\item If the \_squash\_ TUPLE component is used, then maximal sequences of characters that were transliterated to the same character are squashed down to a single instance of that character.
\end{itemize}

The optional \texttt{with\_nbr\_translated\_vbl} alias keyword argument VBL can be used to capture the number of characters replaced, deleted, or squashed. Note that the default global VBL \texttt{nbr\_translated} is available otherwise.
For efficiency’s sake, translate can be run in-place (hence, without the use of malloc or the garbage collection system) on presumably long strings simply by passing the in argument in as an alias.

Test queries include translate.[12].Q. At the end of translate.1.Q, there is some code that illustrates how to produce a listing of all of the C-identifiers that a file contains.

Notes on differences from the Perl "tr" function:

• Since the NUL character (\000) is not supported here, if you specify the _complement_ option, then the first character of the to string will be ignored. This is because translate will never substitute the NUL character in the target in string, and you cannot specify a NUL character in the from string to have it not be considered when complementing. The first character is skipped in order to otherwise produce the same results as you would get for the same inputs in Perl "tr". A runtime error is returned if you only specify one character in the to string when using the _complement_ option without using the _delete_ option.

• The range string "a-m-z" is interpreted as "a-m", ",", and "z". Perl will give an ambiguity error with this type of range string.

UINT(_tiny_) FUN( STR(1) ): uty_for_ch
returns the UINT(_tiny_) ASCII code for the single character argument.

STR FUN( UINT(_tiny_) ARRAY[ [ 1-> ] ] .ara, INT .count = _array_len_ ) str_for_uty_array
returns the STR associated with the ASCII codes in its ARRAY argument. Here is an example (uint.Q):

    local: UINT(_tiny_) ARRAY[ 3 ] .ara
    set .ara[1] = uty_for_ch( "B" );
    set .ara[2] = uty_for_ch( "o" );
    set .ara[3] = uty_for_ch( "b" );
    set .sss = str_for_uty_array( .ara );
    _Say_Eq(.sss, "Bob" );

7.3 Date Functions

Daytona offers a complete collection of DATE operations. These include the usual equality and inequality predicates.

PRED[ STR ] Is_A_Date_Str

This predicate is _true_ if and only if its STR argument can be converted into a valid DATE.
STR FUN: str_for_date( DATE(?), INT .fmt = _arg1_date_fmt_ )

formats its DATE first argument as a STRING. By default, the format chosen for the STRING form of the DATE is the one that comes with the first argument. Otherwise, the user may specify the format for the result by providing second arguments like _Mddyyyy_, thus specifying the subclass.

STR FUN: shell_date()

is an inexpensive way to get the output of the shell date(1) command without invoking or otherwise communicating with a shell.

DATE FUN: today()

returns today’s DATE.

+ or DATE FUN: plus_date_int(DATE, INT)

returns the result of adding second argument number of days to first argument DATE.

− or DATE FUN: sub_date_int(DATE, INT)

returns the result of subtracting second argument number of days to first argument DATE.

− or INT(_long_) FUN: sub_date_date(DATE, DATE)

returns the signed number of days between its two DATE arguments.

DATE FUN: add_months( DATE, INT )

returns the DATE computed by adding the second argument number of months to the first argument. The second argument can be a negative integer.

DATE FUN: min(DATE, DATE), max(DATE, DATE)

returns the minimum (or maximum, respectively) of its two DATE arguments.

STR FUN: day_of_week( DATE )

returns the day of the week corresponding to its DATE argument. The day names are: "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat".

DATE FUN: next_day_of_week( DATE, STR )
returns the DATE corresponding to the next specified day of the week following its DATE argument. The next day is specified by one of the following string: "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat". Actually, the routine turns the first three day characters into an UPLOW and looks for a match with the above day name abbreviations.

STR FUN: month_of_year( DATE )

INT FUN: day_of_year( DATE )
returns the ordinal position of the day of the year corresponding to its DATE argument.

INT FUN: days_per_month( DATE )
returns the number of days in the month containing its DATE argument.

DATE FUN: first_day_of_month( DATE )
returns the DATE of the first day of the month containing its DATE argument.

DATE FUN: last_day_of_month( DATE )
returns the DATE of the last day of the month containing its DATE argument.

INT FUN: days_per_year( DATE )
returns the number of days in the year containing its DATE argument.

INT FUN: day_of( DATE ), month_of( DATE ), year_of( DATE )
returns an INT for the corresponding part for its DATE argument. For example, month_of( 2-4-87 ) = 2 and day_of( 2-4-87 ) = 4.

7.4 Clock Functions

PRED[ STR ] Is_A_Clock_Str
This predicate is true if and only if its STR argument can be converted into a valid CLOCK.
STR FUN: \texttt{str	extunderscore for	extunderscore clock}(\texttt{CLOCK(?), INT .fmt = _arg1	extunderscore clock	extunderscore fmt_})

formats its CLOCK first argument as a STRING, rounding it to the precision specified by the format. By default, the format chosen for the STRING form of the CLOCK is the one that comes with the first argument. Otherwise, the user may specify the format for the result by providing second arguments like \_hhmm\_, thus specifying the subclass.

CLOCK FUN: \texttt{time	extunderscore of	extunderscore day()}

returns the current time of day in 24-hour time.

TIME FUN: \texttt{time	extunderscore for	extunderscore clock()}

returns the TIME in seconds since 00:00:00 corresponding to a given CLOCK.

CLOCK FUN: \texttt{clock	extunderscore for	extunderscore time()}

returns the CLOCK corresponding to the number of seconds since 00:00:00 given by the TIME argument.

CLOCK FUN: \texttt{clock	extunderscore for	extunderscore secs	extunderscore nanosecs( INT, INT )}

returns a CLOCK whose seconds since \^00:00:00^{}CLOCK is the first argument and whose nanoseconds is the second argument.

CLOCK FUN: \texttt{round	extunderscore down	extunderscore to( for	extunderscore clock( CLOCK, TIME )}

returns the CLOCK that results by rounding the first argument down to multiples of the second argument. This function can be called by calling its overloaded correspondent \texttt{round	extunderscore down	extunderscore to}. For example,

\begin{verbatim}
round_down_to(\^22:14:22.721334^{}CLOCK, \^10m^{}TIME) = \^22:10:00^{}CLOCK
\end{verbatim}

CLOCK FUN( CLOCK, TIME ): \texttt{plus	extunderscore clock	extunderscore time, sub	extunderscore clock	extunderscore time}

returns the result of adding (subtracting) the TIME argument to the CLOCK argument, modulo 24 hours so that the result is still a CLOCK. For example,

\begin{verbatim}
\^2pm^{}CLOCK + \^2h30m^{}TIME = \^4:30pm^{}CLOCK
\^11pm^{}CLOCK + \^2h30m^{}TIME = \^1:30am^{}CLOCK
\end{verbatim}

The + and – infix operators can be used instead.

– or TIME FUN( CLOCK, CLOCK ): \texttt{sub	extunderscore clock	extunderscore clock}
returns the result of subtracting the first CLOCK argument from the second CLOCK argument, always assuming that the first argument is later in time. For example,

\[
\begin{align*}
^4:00^\text{CLOCK} & - ^3:00^\text{CLOCK} = ^1\text{h}\text{TIME} \\
^2:00^\text{CLOCK} & - ^10:00^\text{CLOCK} = ^16\text{h}\text{TIME} \\
^2:00^\text{CLOCK} & - ^10:00\text{pm}^\text{CLOCK} = ^4\text{h}\text{TIME}
\end{align*}
\]

This can be useful in benchmarking situations just to find out how much TIME elapsed between the end and the start CLOCKS, even when the end CLOCK is from the next day. If more than one day is involved, then sub\_date\_clock\_pair will do the right thing. In any event, sub\_clock\_clock() will always return a positive TIME. The \(-\) infix operator can be used instead.

\[
\text{sec}_\text{for}_\text{hms}( ^"04:33:22" ) = 16402.0 \\
\text{sec}_\text{for}_\text{hms}( ^"11:3.22" ) = 663.22
\]

### 7.5 Date\_Clock Functions

**PRED[ STR ] Is\_A\_Date\_Clock\_Str**

This predicate is _true_ if and only if its STR argument can be converted into a valid DATE\_CLOCK.

**STR FUN: str\_for\_date\_clock( DATE\_CLOCK(\?), INT \_fmt = \_arg1\_date\_clock\_fmt\_ )**

formats its DATE\_CLOCK first argument as a STRING, rounding it to the precision specified by the format. By default, the format chosen for the STRING form of the DATE\_CLOCK is the one that comes with the first argument. Otherwise, the user may specify the format for the result by providing second arguments like _DDDMMMDddyyyy\_hhmm_, thus specifying the subclass.

**DATE\_CLOCK FUN: date\_clock\_now( )**

returns the current time-of-day down to seconds as a DATE\_CLOCK.

**DATE\_CLOCK FUN: dc\_now( )**

same as date\_clock\_now().

**DATE\_CLOCK FUN: nano\_dc\_now( )**
returns the current time-of-day down to nanoseconds as a DATE_CLOCK. The CLOCK format used is _24_hhmmssf_. This is good for taking timestamps.

DATE_CLOCK FUN: utc_date_clock_now( )
returns the current UTC time-of-day down to seconds as a DATE_CLOCK.

DATE FUN: date_of( DATE_CLOCK )
returns the DATE contained in its argument.

CLOCK FUN: clock_of( DATE_CLOCK )
returns the CLOCK contained in its argument.

DATE_CLOCK FUN: date_clock_for_date_and_clock( DATE, CLOCK )
returns the DATE_CLOCK formed by associating its first argument with its second.

– or TIME FUN: sub_date_clock_pair( DATE_CLOCK(?), DATE_CLOCK(?) )
returns the TIME difference of the second argument subtracted from the first. The minus infix operator can also be used.

DATE_CLOCK(?) FUN( DATE_CLOCK(?), TIME ): plus_date_clock_time, sub_date_clock_time.
adds (subtracts) the TIME second argument from the first, yielding another point on the DATE_CLOCK dimension. The + and − infix operators can be used instead.

DATE_CLOCK FUN: round_down_to_for_date_clock( DATE_CLOCK, TIME )
returns the DATE_CLOCK that results by rounding the first argument down to multiples of the second argument. Please note that it is only the CLOCK portion of the DATE_CLOCK that is rounded down. This function can be called by calling its overloaded correspondent round_down_to. For example,

round_down_to(“^4/15/1993@22:14:22.721334”DATE_CLOCK, ^10m^TIME)  = “^4/15/1993@22:10:00”DATE_CLOCK

DATE_CLOCK FUN: utc_date_clock_for_unix_time( INT .unix_time )
returns a DATE_CLOCK equivalent to the UNIX time argument. See the entry for unix_time.
INT FUN unix_time_for_utc_date_clock( DATE_CLOCK(? ) )

  returns a UNIX time equivalent to the DATE_CLOCK argument. See the entry for unix_time.

DATE_CLOCK FUN( DATE_CLOCK .local_dc, STR .tz = _this_tz_ ) utc_date_clock_for

  This function takes a DATE_CLOCK in the specified time zone and returns the equivalent UTC
  DATE_CLOCK in the sense that if the date and clock are that of .local_dc in .tz, then they
  are utc_date_clock_for( .local_dc, .tz ) in Greenwich, England. See query
  localtz.1.Q in the test-suite for other examples.

DATE_CLOCK FUN( DATE_CLOCK .utc_dc, STR .tz = _this_tz_ ) local_date_clock_for

  The first argument is a DATE_CLOCK that is taken to be a UTC time point, where UTC is the
  (surprising) acronym for Universal Coordinated Time, formerly known as Greenwich Mean
  Time. If the second argument is the default _this_tz_, then the function returns the
  corresponding DATE_CLOCK for the timezone corresponding to the machine that the executable
  is running on. More precisely, it corresponds to the localtime associated with the value of the
  TZ variable in the shell environment that the executable is being run in. Optionally, the user can
  provide a TZ value as the second argument to the function at which point the function will
  return the local DATE_CLOCK corresponding to the UTC DATE_CLOCK and the TZ value
  offered as the second argument.

  The official documentation for acceptable TZ values for your platform can be found in the
  man pages for tzset, localtime, or environ (section 5). Pretty much, the different platforms conform
  to the POSIX standard and will be consistent with the following description. If the first character
  of the TZ string is a colon, then the interpretation is implementation or system dependent. On
  Sun Solaris, this means that the value can be any path found in /usr/share/lib/zoneinfo. For
  Linux, it can be any path under /usr/share/zoneinfo or /usr/lib/zoneinfo. SGI does not
  apparently provide any implementation meaning for TZ values beginning with colon. As an
  example, a Solaris TZ value of US/Eastern corresponds to the Eastern timezone of the United
  States.

  Of more general utility is the case where the user themselves create their own timezone
  specifications as values for TZ (or for the second argument of local_date_clock_for). Quoting
  from the Linux manual,
The value of TZ can be one of three formats. The first format is used when there is no daylight saving time in the local time zone:

    std offset

The std string specifies the name of the time zone and must be three or more alphabetic characters. The offset string immediately follows std and specifies the time value to be added to the local time to get Coordinated Universal Time (UTC). The offset is positive if the local time zone is west of the Prime Meridian and negative if it is east. The hour must be between 0 and 24, and the minutes and seconds 0 and 59.

The second format is used when there is daylight saving time:

    std offset dst [offset],start[/time],end[/time]

There are no spaces in the specification. The initial std and offset specify the standard time zone, as described above. The dst string and offset specify the name and offset for the corresponding daylight savings time zone. If the offset is omitted, it defaults to one hour ahead of standard time.

The start field specifies when daylight savings time goes into effect and the end field specifies when the change is made back to standard time. These fields may have the following formats:

Jn This specifies the Julian day with n between 1 and 365. February 29 is never counted even in leap years.

n This specifies the Julian day with n between 1 and 365. February 29 is counted in leap years.

Mm.w.d This specifies day d (0 <= d <= 6) of week w (1 <= w <= 5) of month m (1 <= m <= 12). Week 1 is the first week in which day d occurs and week 5 is the last week in which day d occurs. Day 0 is a Sunday.

The time fields specify when, in the local time currently in effect, the change to the other time occurs. If omitted, the default is 02:00:00.

For example, the most complete setting for New Jersey in 1986 could be

    EST5EDT4,116/2:00:00,298/2:00:00

or simply

    EST5EDT

An example of a southern hemisphere setting such as the Cook Islands could be

    KDT9:30KST10:00,63/5:00,302/20:00
The query localtz.Q in the test-suite contains other examples.

\[
\text{DATE\_CLOCK FUN( INT .unix\_time, STR .tz = \_this\_tz\_ ) local\_date\_clock\_for\_unix\_time}
\]

Returns the DATE\_CLOCK for the UNIX time in seconds that is local to the given timezone.

7.6 Time Functions

Daytona offers a complete collection of TIME operations. These include the usual equality and inequality predicates. In addition, there are:

**PRED[ STR ] Is\_A\_Time\_Str**

This predicate is _true_ if and only if its STR argument can be converted into a valid TIME.

**TIME FUN: time\_for\_str( STR )**

**STR FUN: str\_for\_time( TIME )**

These functions convert back and forth between TIMES and STRINGS that represent TIMES.

**FLT(_long_) FUN: flt\_for\_time( TIME )**

**TIME FUN: time\_for\_flt( FLT(_long_) )**

These functions convert back and forth between TIMES and quantities of seconds expressed as FLOAT(_long_).

**TIME FUN( TIME, TIME ): plus\_time, sub\_time, min\_time, max\_time**

These functions add, subtract, minimize, and maximize their two arguments, respectively. Since the appropriate infix operators are available, it is not necessary to call plus\_time and sub\_time explicitly by those names.

**TIME FUN: mod\_time( TIME, TIME )**

returns the common residue of the first argument modulo the second argument. See the discussion of mod\_int. This function can be called by calling its overloaded correspondent _mod_ or by using the % infix operator.

**TIME FUN: mod\_clock\_time( CLOCK, TIME )**

returns the common residue of the first argument taken as a TIME modulo the second argument. See the discussion of mod\_int. This function can be called by calling its overloaded correspondent _mod_ or by using the % infix operator.
TIME FUN( FLT, TIME ) mult_flt_time
This function yields the TIME resulting from multiplying a TIME by a FLT. The * infix operator
eliminates any need to call this function explicitly.

TIME FUN( TIME, FLT ) div_time_flt
This function yields the TIME resulting from dividing a TIME by a FLT. The / infix operator
eliminates any need to call this function explicitly.

FLT FUN( TIME, TIME ) div_time_time
This function yields the FLT resulting from dividing a TIME by a TIME. The / infix operator
eliminates any need to call this function explicitly.

INT(? ) FUN: secs_of( TIME\|CLOCK )
returns the number of seconds contained in its argument.

INT FUN: nanosecs_of( TIME\|CLOCK )
returns the number of nanoseconds contained in its argument (after removing the number of
seconds).

TIME FUN: time_for_secs_nanosecs( INT, INT )
returns a TIME whose seconds is the first argument and whose nanoseconds is the second
argument.

TIME FUN: round_down_to_for_time( TIME, TIME )
returns the TIME that results by rounding the first argument down to multiples of the second
argument. This function can be called by calling its overloaded correspondent round_down_to.
For example,

round_down_to(`2d3h22m3.5s`TIME, `10m`TIME) = `2d3h20m0s`TIME

7.7 IP Networking Functions

Remember that other IP networking functions are presented in Chapter 5 Section 2.
UINT(_long_) FUN( IP(_heko_) ) ip_uint_for_ip_hoko
returns the result of converting an IP(_heko_) address into a UINT(_long_) in network byte
order (which is big-endian) where each octet yields its binary representation in zero’s and one’s.

IP(_heko_) FUN( UINT(_long_) ) ip_heko_for_ip_uint
returns the result of converting a UINT(_long_) in network byte order (which is big-endian) into an IP(_heko_) where each byte of the UINT yields an octet for the IP.

IP(_heko_ | _uint_ ) FUN( IP(_heko_ | _uint_ ) | ip, IP(_heko_ | _uint_ ) | INT ) masked_ip
IP(_v6_ ) FUN( IP(_v6_ ) | ip, INT ) masked_ip
returns the IP that results by masking the argument IP with the indicated mask. In the case of IPv4, if the mask is an IP like ‘255.255.255.0’IP, then the concatenation of the binary representations of the octets is anded with the binary representation of the argument IP and the result is converted back into an IP. If the mask is an INT n, it stands for a mask whose n 1 bits extend from the left and are completed to 32 bits by 0’s. So,

```
masked_ip( `144.22.11.33`IP, `255.255.255.0`IP ) = `144.22.11.0`IP
masked_ip( `144.22.11.33`IP, 24 ) = `144.22.11.0`IP
```
Likewise for IP6 inasmuch as the masked_ip() function is overloaded to work with IP(_heko_), IP(_uint_), and IP6/IP6Z.

By the way, the anding operation mentioned above assumes (correctly) that the IP is represented by a UINT stored in network byte order (big-endian). For big-endian UINTS (or 128-bit integers), the bits are stored in memory in order of their associated powers of two, with the bits for the highest powers of two stored first in the lowest memory address for the UINT bytes. This would be too logical for little-endian machines, which include all Intel x86 boxes. There the bytes of the word are stored in increasing memory locations by increasing significance in their powers of two; on the other hand, within each byte, the bits are stored in decreasing significance in their powers of two. This hybrid approach rules out any simple anding to compute netmasks as well as ruling out other simple implementations of other fpps like inequality predicates. Note that big-endian machines run just fine and there are machines in the world where the user can actually choose which flavor of endian they want. That’s how useful little-endian is as a construct.

otherwise_ok IHOST FUN( for STR(*) | IP(?) .ip,
(0->1) with_ihost_aliases_vbl manifest alias STR(*) = ihost_aliases ) ihost_for_ip
define CLASS GETHOST_CALL_STATUS
with_symbols { _host_not_found_, _no_data_, _try_again_, _no_recovery_, }

This function uses the DNS services underlying gethostbyaddr or getipnodebyaddr to map IP addresses to their IHOST hostnames.

The ancillary VBL argument for ihost_for_ip will assign to the VBL a STRING value consisting of a (vertical bar separated) list of additional IHOSTS that are associated with the IP
An optional otherwise statement enables concise and exhaustive error handling appropriate for the four different error scenarios, which go here by their official, if not particularly apt, names. _host_not_found_ means that the system cannot find any information on the given IP. _no_data_ is an unlikely scenario that indicates a DNS server has information for an IHOST but does not have any IP addresses associated with that IHOST. _try_again_ means that the authoritative DNS server is temporarily unable to resolve the enquiry. _no_recovery_ means that the DNS system encountered an unrecoverable persistent system error.

This function is overloaded to work with IP, IP2, IP6/IP6Z, and STR(*)

```plaintext
otherwise_ok STR(*) FUN( for IHOST .ihost, 
(0->1) with_ip_aliases_vbl manifest alias STR(*) = ip_aliases ) ip_str_for_ihost 
```

define CLASS GETADDRINFO_CALL_STATUS 
with_symbols { _eai_addrfamily_, _eai_again_, _eai_badflags_, _eai_fail_, 
_eai_family_, _eai_memory, _eai_nodata_, _eai_noname_, _eai_overflow_,
_eai_service_, _eai_socktype_, _eai_system_ } 

This function uses the DNS services underlying getaddrinfo to map IHOST hostnames to their IP addresses, returning the first valid result as a STRING value.

The ancillary VBL argument for ip_str_for_ihost will assign to the VBL a STRING value consisting of a (vertical bar separated) list of additional IP addresses that are associated with the IHOST argument. (find_ip.IQ, ihost.1.Q)

An optional otherwise statement enables error handling appropriate for the twelve different error scenarios listed above. More information on each of these can be found in the man page for getaddrinfo.

See also the overloaded Starts PREDICATE which can take IP(_heko_) arguments but not IP(_uint_) arguments.

7.8 UNIX System Fpps

7.8.1 UNIX System Fpps: File-Oriented

Several of Daytona’s built-in functions provide for productive interaction with the UNIX file system. Each of the file and directory arguments below may use the . and .. abbreviations, the $ shell metacharacter, and the ~ ksh special character if ksh is being used.

PRED: File_Exists[ STR ]

is_true_ if and only if the file argument exists.
PRED: File_Is_Ordinary [ STR ]
    is_true if and only if the file argument is an ordinary file. The file argument is assumed to exist.

PRED: File_Isa_Dir [ STR ]
    is_true if and only if the file argument is a directory. The file argument is assumed to exist.

PRED: File_Isa_Fifo [ STR ]
    is_true if and only if the file argument is a FIFO. The file argument is assumed to exist.

PRED: File_Is_Read_Only [ STR ]
    is_true if and only if the file argument is read-only. The file argument is assumed to exist.

PRED: File_Is_Openable_For [ STR .file , STR .mode ]
    is_true if and only if the file argument is openable as indicated by the mode string. The mode string is the standard one used by fopen such as "r+";

INT FUN( STR ) file_mod_time
    returns the UNIX file modification time in seconds since 1970-01-01 for the UNIX file path expressed by its first argument. If the file does not exist, then "-1s"TIME is returned. See the man page for stat(2).

INT FUN( STR ) file_acc_time
    returns the UNIX file access time in seconds since 1970-01-01 for the UNIX file path expressed by its first argument. If the file does not exist, then "-1s"TIME is returned. See the man page for stat(2).

INT FUN( STR ) file_stat_time
    returns the UNIX file status time in seconds since 1970-01-01 for the UNIX file path expressed by its first argument. The status is changed when the attributes of the file in the filesystem are changed. See the man page for stat(2). If the file does not exist, then "-1s"TIME is returned.

INT(_huge_) FUN: file_size( STR )
    returns the length of the file in bytes.

UINT FUN: file_mode( STR )
    returns the stat.h st_mode of the file. This includes the permissions.

INT FUN: file_uid( STR )
    returns the stat.h st_uid of the file.

INT FUN: file_gid( STR )
    returns the stat.h st_gid of the file.
STR FUN: whence( this STR, in STR )

returns the first directory in the colon separated directory sequence in argument that contains this, the file argument.

PRED: Was_Modified_After[ STR, STR ]

is_true if and only if the file first argument was modified after the file second argument, which is considered to be the case if the file second argument doesn't exist at all.

PRED: Touched[ STR ]

is_true if and only if the modification time of the file first argument was successfully changed to the current time. Failure will certainly occur if the user is neither the owner of the file nor has write permission for the file. Error messages are printed out in time of error.

BOOL FUN: unlink( STR ); PROC: Unlink( STR )

unlinks (or removes) its file first argument from the file system, whether or not the file existed before. The FUN returns true on successful removal. Shell expressions are acceptable arguments to these fpps.

BOOL FUN( STR .fl, INT .pid_spec = _this_pid_, (0) INT .patience = 90 ) lock_file BOOL unlock_file( STR )

The lock_file FUN returns true if and only if Daytona is able to create a .L lock file for the file path specified by its first argument. If a lock file is already in place, lock_file will try each second for 90 seconds to get its own lock file before accepting failure. Incidentally, the .L lock files are identified by a process id (pid) and so, if that process has terminated without removing its .L’s, then those .L’s are considered orphans and are removed when detected. The identifying pid is given by .pid_spec: it is the invoking process's pid if _this_pid_, the pid of the parent of the invoking process if _par_pid_, or else .pid_spec itself. The unlock_file FUN returns true if and only if Daytona is able to remove the associated lock file.

Get_Lock_File_For <name>, Rm_Lock_File_For <name>

Firstly, neither one of these are Cymbal fpps! Both are executables. Their use is in shell scripts. Each takes one argument which is just a name lockable that is used to create/remove the Daytona lock file lockable.L. Their only subtlety is that the lock file they create is owned by the parent shell process that calls them so that when Get_Lock_File_For exits, the lock file that it creates is still owned by a running process, i.e., the parent.

Get_File_Lock <file> [ <time_in_secs=15> <mode=_read_> <patience=_wait_on_block_> ]

Firstly, this Get_File_Lock is not a Cymbal fpp! It is an executable to be used at the shell level. It serves as an interface to fcntl(2) advisory file locking. The mandatory first argument is a file path. The optional third argument is the mode of the lock which is one of _read_ or _write_ or _update_: _update_ will first try to get a _read_ lock and upon success, will then attempt to
upgrade that to a _write_ lock. The optional second argument is the time in seconds that Get_File_Lock will run before trying to upgrade the lock (if it tries at all) at which point, the upgraded lock is held for 15 seconds. The optional fourth argument is the Cymbal patience used to obtain the lock.

PROC( STR(=) .\_path, INT .mode = _write_ ) Lock_Blockers_For_File_Path

prints out information on some lock, if any, that is blocking the acquisition of a lock with the indicated mode on the specified file. The two mode alternatives are _write_ and _read_. This information includes the pid of a blocking process, if any, as well as "ps -fl" information on that process.

STR FUN(STR): expanded_fl_name, rooted_expanded_fl_name

expanded_fl_name takes a STR file name/path as its argument and returns the result of expanding all shell $ and tilde expressions as well as . and .. . rooted_expanded_fl_name returns the value of expanded_fl_name if it begins with slash, else it prepends the value of $PWD to the value of expanded_fl_name.

STR FUN() getcwd

returns the current working directory as provided by the Section 2 getcwd system call. If getcwd fails, the program will abort with an error message.

otherwise_ok INT(_int_) FUN( constant STR(=) .\_path, INT .mode = int_for_octal_str("755") ) mkdir

is an interface to the UNIX mkdir function which makes directories, as illustrated by:

set ? = mkdir( "./Source", int_for_octal_str( "750" ) )
o otherwise do Exit(1);

Note the use of otherwise to catch errors. Like a typical UNIX system call, mkdir returns 0 if successful but in Daytona's case, if unsuccessful, it will return the value of the UNIX global variable errno. This is handy because if the directory already exists, then mkdir will fail with call status _eexist.; consequently, such an eventuality can be ignored by using code like this:

set .call_status = mkdir( "./dS_TeSt" );
when( ! (.call_status = 0 |=_eexist_) ) do Exclaim_Line( mkdir_call_status_msg() );

_ernoent_ is the call_status if a component of the path prefix does not exist.

otherwise_ok INT(_int_) FUN( constant STR(=) .\_path, INT .mode = int_for_octal_str("755") ) mkdirp

is an interface to the UNIX mkdirp function which makes directories, as illustrated by:
set ? = mkdirp( "./Source/Perl/Perl5/", int_for_octal_str( "750" ) )
    otherwise do Exit(1);

mkdirp is distinguished from mkdir because it will create intervening directories as needed. Note the use of otherwise to catch errors. mkdirp returns 0 on success and on failure, it returns either errno if nonzero or else -1. Just like mkdir, it can return _eexist_ or _enoent_.

otherwise_ok INT(_int_) FUN( constant STR(=) .path ) rmdir
    removes the directory specified by .path by calling the UNIX Section 2 rmdir function. Note the use of otherwise to catch errors. Like a typical UNIX system call, rmdir returns 0 if successful but in Daytona's case, if unsuccessful, it will return the value of the UNIX global variable errno. _eexist_ is the call_status if the directory contains entries other than those for . and ..; _enoent_ is the call_status if the named directory does not exist.

otherwise_ok INT(_int_) FUN( constant STR(=) .path ) rmdirp
    removes all directories specified by .path by repeatedly calling the UNIX Section 2 rmdir function. rmdirp will reject paths that start with / and paths that contain .. . Note the use of otherwise to catch errors. mkdirp returns 0 on success and on failure, it returns either errno if nonzero or else -1. Just like rmdir, rmdirp can return call status _eexist_ or _enoent_.

otherwise_ok INT(_int_) FUN( constant STR(=) .path, INT .mode ) chmod
    changes the mode of the directory specified by .path by calling the UNIX Section 2 chmod function.

        set ? = chmod( "./Source", int_for_octal_str( "750" ) )
        otherwise do Exit(1);

Note the use of otherwise to catch errors. Like a typical UNIX system call, chmod returns 0 if successful but in Daytona's case, if unsuccessful, it will return the value of the UNIX global variable errno.

otherwise_ok INT(_int_) FUN( constant STR(=) .old_path, constant STR(=) .new_path) rename
    uses the Section 2 system call rename to rename a file from .old_path to .new_path. Note the use of otherwise to catch errors. Like a typical UNIX system call, rename returns 0 if successful but in Daytona's case, if unsuccessful, it will return the value of the UNIX global variable errno.

7.8.2 UNIX System Fpps: General

PROC: Sleep( INT | TIME )
    causes the program to sleep for a time equal to its argument. If the argument has type INT, then
the amount of time is in seconds; if the argument has type TIME, then the amount of time is that specified by the argument. One advantage of using a TIME argument is that the sleep interval can be specified down to microseconds.

PROC: Pause( )
causes the program to pause using pause(2), waiting indefinitely for a (not-ignored) signal like SIGCONT or SIGUSR1 to arrive. Typically, one would install a no-action signal handler that catches the desired signal(s): then upon receipt of such a signal, the program would resume execution as if nothing had happened.

PROC: Exit( ( 0->1 ) with_msg STR, INT .status )
causes the program to exit with a status equal to its INT argument. In addition, stdout is flushed and if the status is not 0, then a helpful message is printed. The user's own message can be attached by using the with_msg keyword.

INT FUN: getpid()
returns the process ID of the calling process.

INT FUN: getppid()
returns the process ID of the parent of the calling process.

INT FUN: getpgrp()
returns the process group ID of the calling process.

PRED[ INT .pid ] Proc_Exists
is_true_ if there is a process with the given pid; else_false_.

PRED[ INT .pid ] Proc_Is_Alive_And_Well
is_true_ if there is a process with the given pid that is not a zombie; else_false_. A zombie is a process that has exited but which has not yet been wait(2)ed on by its parent so as to pass along its exit status information.

PRED[ INT .pid, STR(=) .machine ) Proc.Exists_On
returns_true_ if there is a process with the given pid on the machine with the given DNS name argument; else_false_. Since this command will use ssh to contact the given machine (if it is not
local), the user must have ssh configured to be available to execute remote commands successfully on that machine without any prompting or else the command will fail or the user will be prompted.

otherwise_ok INT FUN( INT .pid, INT .sig ) kill

is an interface to the Section 2 kill UNIX system call and causes the given signal to be sent to the process with the given process id. The signals should be taken from CLASS SIGNAL as defined in $DS_DIR/sys.env.cy .

      set .call_status = kill( getpid(), _usr2_ ) otherwise do Exit(1);

PROC: Get_Pid_Info(

      INT .pid,

      ( 0->1 ) with_par_pid alias INT = pid_par,
      ( 0->1 ) with_grp_ldr_pid alias INT = pid_grp_ldr,
      ( 0->1 ) with_pct_cpu alias FLT = pid_pct_cpu,
      ( 0->1 ) with_img_siz alias INT(_huge_) = pid_img_siz_in_K,
      ( 0->1 ) with_res_set_siz alias INT(_huge_) = pid_res_set_siz_in_K,
      ( 0->1 ) with_heap_siz alias INT(_huge_) = pid_heap_siz_in_K,
      ( 0->1 ) with_stack_siz alias INT(_huge_) = pid_stack_siz_in_K,
      ( 0->1 ) with_invoc alias STR(*) = pid_invoc,
      ( 0->1 ) with_nice alias INT = pid_nice,
      ( 0->1 ) with_syscall alias INT = pid_syscall,
      ( 0->1 ) with_state alias STR(1) = pid_state
)

takes a mandatory pid argument and retrieves current information about the associated process that is described by the various optional ancillary variable keyword arguments. Clearly, if no optional VBL arguments are provided, then the indicated global VBLS will be set appropriately. Used in the Daytona executable Stat_Proc.

PROC: Get_Times_For_Pid(

      INT .pid,

      ( 0->1 ) with_elapsed_time_vbl alias TIME = dummy_time,
      ( 0->1 ) with_own_user_time_vbl alias TIME = dummy_time,
      ( 0->1 ) with_own_sys_time_vbl alias TIME = dummy_time,
      ( 0->1 ) with_kids_user_time_vbl alias TIME = dummy_time,
      ( 0->1 ) with_kids_sys_time_vbl alias TIME = dummy_time
)

takes a mandatory pid argument and retrieves current timing information about the associated
process that is described by the various optional ancillary variable keyword arguments. Used in the Daytona executable Stat_Proc.

_Init_Time_Store
_Print_Time_Delta

are two ds_m4 macros that are used to print out timing information characterizing portions of a process’s execution. To initialize the counters, just have the execution pass through a _Init_Time_Store statement and then the elapsed, own user, own system, kids user, and kids system times for the corresponding interval will be printed out to stderr upon encountering any subsequent _Print_Time_Delta statements. Any other Cymbal I/O CHAN (e.g., _stdout_) will be used for the output if provided as the first and only argument to _Print_Time_Delta.

INT FUN: unix_time( )

returns the current UNIX time, the time of day in seconds since 00:00:00 UTC, January 1, 1970, where UTC is the (surprising) acronym for Coordinated Universal Time, which used to be known as Greenwich Mean Time.

STR FUN: localtime_for_unix_time( INT )

returns a readable string like "Sat May 17 00:03:23 1997" revealing the local time corresponding to the UNIX time argument. The result is essentially asctime(localtime()) for the ctime(3) C functions. See also local_date_clock_for.

STR FUN: utctime_for_unix_time( INT )

returns a readable string like "Sat May 17 00:03:23 1997" revealing the UTC DATE_CLOCK corresponding to the UNIX time argument. The result is essentially asctime(gmtime()) for the ctime(3) C functions.

PROC: Diagnose_Sys_Err()

prints out to _stderr_ an explanation of the last system runtime error errno, if any. This is very handy, for example, when functions like new_channel encounter _failed_fopen_ because Diagnose_Sys_Err will pinpoint the exact reason why.

PROC( INT .priority, STR(=) .msg_fmt, STR(=) .optarg = _null_str_ ) Write_To_Syslog

writes the printf-style format .msg_fmt STRING, possibly augmented by the optional .optarg STR argument, to syslog with the given priority. The only format specifications allowed in .msg_fmt are %s (or variants thereof) and %m, the latter used to include the error message string associated with the current value of errno. This is just a simple interface to the syslog(3C) function.
UINT(_long_) FUN physmem_in_K UINT(_long_) FUN avail_physmem_in_K INT(_long_) FUN nbr_config_cpus INT(_long_) FUN nbr_online_cpus

On Linux and Solaris, these functions return the values indicated.

PROC(
(0->1) for_1_min alias FLT = load_avg_for_1_min,
(0->1) for_5_mins alias FLT = load_avg_for_5_mins,
(0->1) for_15_mins alias FLT = load_avg_for_15_mins ) Get_Load_Avg

On Linux and Solaris, these functions return the indicated CPU load values.

A variety of signal handling functions are described in Chapter 19.

7.8.3 UNIX System Fpps: Shell Interaction

In addition to _popkorn_, Daytona provides more conventional facilities for interacting with the shell.

PROC: Shell_Elex( STR|CMD|SHELLP )
otherwise_ok INT FUN( STR|CMD|SHELLP ) shell_exec

When invoked with a STR|CMD, both of these fpps cause a system(3) command to be executed on their argument, which has to fit on a single line; the FUN returns the terminating/stopping signal number (if any!) or else the exit status of the child (shell) process and the PROC causes the program to abort on error. More flexible and interesting behavior occurs when a SHELLP is the argument. First, the SHELLP can be an arbitrary multi-line Korn shell program. Secondly, while the PROC will print an error message and abort the program if the SHELLP fails, the FUN will:

1. return 0 on success (as do all UNIX system calls)
2. return the exit status of the shell program if it calls exit
3. return 1000 + the signal number if the program was killed
4. return 10000 + the signal number if the program was stopped

shell_exec is otherwise_ok, thus giving rise to a convenient idiom:

set ? = shell_exec(.my_cmd) otherwise do Exit(2);

Here’s an example of the PROC using a SHELLP argument:
The execution of both the FUN and PROC is synchronous, meaning that execution of the calling program does not continue until the shell command or program has finished.

**STR FUN: shell_eval**

returns the result of executing the argument as a shell command. This return value must be less than 10240 characters. Longer values can be dealt with by using `_popkorn_`. The return value cannot contain the control characters `^E` or `^P`. Due to interface issues with ksh, unless the string `print -n` appears as a literal/exact substring of the command argument, the output of `shell_eval` will be terminated by exactly one new-line, even if the command argument output contains several trailing new-lines. There are a variety of exotic examples of `shell_eval` use in syscall.1.Q.

**STR FUN: shell_echo**

returns the result of echoing its argument. This is very handy for expanding shell variables and for doing ksh tilde expansion if ksh is present.

**STR FUN: shell_env**

returns the value of its argument in the shell environment.

**BOOL FUN: put_shell_env**

returns `_true_` if and only if it modified the shell environment as directed by its "<shell_vbl>=<value>" STR argument (as in "HOME=/usr/john").

### 7.8.4 Unix System Fpps: Getting And Setting Process Resource Limits

Here is Daytona’s interface to getting and setting limits on a query’s consumption of any of seven system resources as implemented via the UNIX getrlimit(2) and setrlimit(2) commands:
define CLASS SYSTEM_RESOURCE
    with_symbols { _rlimit_corefile_bytes_, _rlimit_cpu_secs_,
    _rlimit_dataset_bytes_, _rlimit_outfile_bytes_,
    _rlimit_openfiles_, _rlimit_stack_bytes_, _rlimit_vmem_bytes_ }
define CLASS SYSTEM_RESOURCE_VALUE with_symbols { _rlimit_infinity_ }
import gbg_not_here:
    PROC( INT .resource, INT(_off_t_) .hard_limit ) Set_Hard_Resource_Limit
    PROC( INT .resource, INT(_off_t_) .soft_limit ) Set_Soft_Resource_Limit
    INT(_off_t_) FUN( INT .resource ) get_hard_resource_limit
    INT(_off_t_) FUN( INT .resource ) get_soft_resource_limit

The seven different system resources that one can identify by means of the special constants in
SYSTEM_RESOURCE should be self explanatory along with their one special value _rlimit_infinity_.
Each resource has both a soft and hard limit. The soft (or current) limit must always be less than
the hard limit. The hard limit can be irreversibly lowered to any value between the current hard limit and
the current soft limit, with the understanding that only a process with an effective user ID of super-user
can raise a hard limit. The actual integer that is represented by _rlimit_infinity_ is revealed in
/usr/include/sys/resource.h. Resource limits are preserved and transferred to child processes created by
means of fork and/or exec. Further information on this functionality can be found in the UNIX man
pages for getrlimit/setrlimit.

7.9 Datatype Conversion Functions

Daytona provides type conversion functions for every type A that can be coerced into some
type B. The name for each such function is usually the lowered version of Bs name followed by
"_for_" followed by the lowered version of As name. For example,

    INT( _long_ ) FUN: int_for_str( STR )

Other than being wordy, this style of casting has the additional disadvantage of being even more
wordy and sometimes idiosyncratic when it comes to casting from subclasses to subclasses as from
DATE(_ddMMMyyyy_) to DATE(_yyyymmdd_). Fortunately however, Daytona also supports C-style
casting wherein the expression to be casted is preceded by the target Cymbal type (enclosed in
parentheses). Here are several examples:

    local: INT .x = 123456
    set .y = substr( (STR) .x , 4, 3 );
    _Say_Eq( .y, '456' )
    set .z = '1999-12-31';
    do Write_Line( (DATE(_ddMMMyyyy_)) .z );
    set .a = (INT(_short_))12345;
    set .b = (STR(3)).a;
    _Say_Eq(.b, '123')

(_Say_Eq is a system ds_m4 macro that prints out a satisfaction claim saying that its two arguments are
equal.)
Note that in this usage, the source type is not explicitly given, because Daytona can infer it. Hence this C-style of casting is the preferred way to cast: it is brief, reuses existing syntax, and generalizes to even more complicated casts (such as casting from one SET type to another, not that is supported at this time). It also insulates the user from changes in Daytona implementation which have been known to change the names of the lower-level casting functions.

And as a bonus, this C-style casting supports casts which the system will never perform automatically on any occasion. Note how the \texttt{STR(3)} cast serves to truncate (the string version of) its argument. Note how easily the format of a DATE can be changed for output purposes. These are accomplished much more easily with this (common) convention than by trying to remember and use the many different conversion functions explicitly. Important note: the parenthesized type casting unary operator has the same precedence as exponentiation. This implies that it binds more tightly than addition, subtraction, multiplication, and division. Consequently, the following is true in general:

\[
\text{INT}(_\text{short}_).5*.x \neq \text{INT}(_\text{short}_)(.5*.x)
\]

### 7.10 Miscellaneous Fpps

Here are some useful odds and ends.

the unary splice operator \% takes a TUPLE or conventional (fixed-size) ARRAY argument and makes a sequence consisting of the elements of the aggregate for immediate inclusion into a surrounding TUPLE or BUNCH. Think of it as carefully removing the peas from their pod while preserving their order. Its semantics rely on making a distinction between a sequence of items and a contained sequence of items, the latter being a TUPLE.

```c
local:
    TUPLE[ INT, INT, INT ] .tu = [ 1, 2, 3 ]
    INT ARRAY[ 4 ] .ara = [ 4, 5, 6, 7 ]
    set .bbb = [ %tu, %ara ];
    fet .x Is_In .bbb { do Write_Line(.x); }
```

It’s an unusual operator because its result, being a sequence, is not even part of the Cymbal type system. That’s why the application of the operator must occur inside a (manifest) TUPLE or BUNCH so that the ‘spilled-out’ elements can be immediately integrated into a typed quantity. Please note that since this expansion into a sequence takes place at compile-time, splice cannot accept a dynamic associative array as an argument.

```c
INT FUN( STR(?)|LIT(?)|RE(?)|BITSEQ|SAFE_STR
    |ISO8859(?)|BASE64(?)|ISVC|IHOST|IHOST_PORT|IHOST_SVC|UDPATH
    |TEXT(?)|AUDIO(?)|VIDEO(?)|PHOTO(?)|BLOB(?)|CMD(?)|SHELLP(? ) ) length
returns an appropriate length for its argument.
```
BOOL FUN: truth( ASN )
  returns _true_ if and only if its argument assertion is true. \textit{truth()} is the only exception to the rule that assertion arguments to \texttt{fpps} must be parenthesized, although of course, it doesn’t hurt.

BOOL FUN: negation( BOOL )
  returns _true_ if its argument assertion is _false_ and _false_ if its argument is _true_.

OK_OBJ FUN: if\_else( ASN, OK\_OBJ, OK\_OBJ )
  returns its second argument if its parenthesized assertion first argument is true; otherwise, the third argument is returned. The second and third arguments must have the same datatype. (OK\_OBJ indicates that arguments of any type are acceptable.)

OK_OBJ FUN: same( OK\_OBJ )
  returns its argument. (OK\_OBJ indicates that arguments of any type are acceptable.)

STR getopt( STR .options )
  takes a STR of command line option flags as defined by the user in accordance with the POSIX command syntax standard (see intro(1)). \textit{As getopt} is called repeatedly, it returns the next option flag in the command line for the current executable invocation. It returns "" when there are no flags left that haven’t been visited. Here is an excerpt from getopt.1.IQ:
while( getopt( "ab:c:d" ) != "" )
{
    switch( .opt )
    {
    case( = "a" )
    {
        do Write_Line( "Found -a" );
    }
    case( = "b" )
    {
        do Write_Words( "Found -b", .optarg );
    }
    ...
    case( = "?" )
    {
        do Write_Words( "Found bad flag", .optopt );
    }
    }
}

The options indicate that the flags -a, -b, -c, -d are supported with -b, -c each taking one argument. The global STR .opt is set to be the last value returned from getopt. The argument to an argument-taking flag is provided in the global STR .optarg. The argument may or may not be separated from its flag by whitespace. In the event that an unauthorized flag is used or if there is no argument to a flag that requires one, getopt returns ?, .opt is set to ? and the global STR .optopt is set to the value of the bad flag.

The Cymbal getopt is implemented using its cousin getopt(3C). The + arguments that Daytona automatically supports for executables are appropriately ignored for the purposes of getopt. getopt operations are handled completely independently of _cmd_line_ work.

7.11 Convenient Macros

The following ds m4 macros are defined in $DS_DIR/sys.macros.m and provide convenient expansions into helpful code. Just use DS M4 to expand examples.

_BC
_EC

These stand for begin comment and end comment, resp. Since they are m4-based, they are processed before any other comment conventions and simply result in the code that they bracket totally disappearing from the text that is input to Tracy.

_Lowercase( _arg1 )
_Uppercase( _arg1 )
These two macros change the case of their argument as indicated. Of course, their argument does not have to be any construct that would be recognized by the Cymbal parser because the parser can only see the result of the expansion.

```c
_Show_Vbl_To(_vbl [, _chan=stdout ])
_Show_Exp_To(_vbl [, _chan=stdout ])
```

These two macros print out a variable or an expression to a CHAN as illustrated by:

```c
local: INT .x = 22
_Show_Vbl_To(\(x)\)
_Show_Exp_To(\(x^2\))
// And here is the result:
.x = 22
.x^2 = 484
```

```c
_Say_Eq(_expr1, _expr2 )
_Must_Eq(_expr1, _expr2 )
```

These two macros print an assertion that the value of two expressions are equal. _Must_Eq goes further and aborts with an error message if they are not; it also insists that the two expressions be of the same type.

```c
local: INT .x = 22
_Say_Eq(\(x+22\), 2*.x)\)
_Must_Eq(\(x+22\), 2*.x)\)
// And here is the result:
44 = 44
44 = 44
```

```c
_Say(_text, _outchan=stdout )
_Say_ISTR(_text, _outchan=stdout )
```

These macros Write Words out their _text arg to the indicated _outchan CHAN. They both begin by skipping a line. _Say_ISTR puts its _text arg into an ISTR and adds an extra new-line to its output.

```c
_Say(hello, world)
_Say_ISTR(@< >@\.choice = .choice)
```
8. I/O: Reading And Writing On Channels

Daytona supports a uniform, stream-oriented notion of I/O from a variety of sources including files, pipes, strings, and the command line.

8.1 Channels

Central to Cymbal I/O is the notion of an I/O CHANNEL. An I/O CHANNEL is a stream of bytes. Input channels come into the Cymbal program and output channels leave the program. In contrast to *awk*, this stream of bytes has no intrinsic or necessary structure. In particular, there is no built-in awareness of lines since the new-line character has no intrinsic, special meaning.

Using their Cymbal names, the different kinds of I/O channels are `stdin`, `stdout`, `stderr`, `cmd_line`, `file`, `string`, `text`, `pipe`, `bipipe`, `fifo`, `fipifo`, `funnel`, `tcp`, `unix_domain`, and `popkorn`. The so-called constant channels, `stdin`, `stdout`, `stderr`, `cmd_line`, and `popkorn`, are more than just different kinds of channels since each one is, in fact, the sole member of its class; consequently, the name of this class is also taken to be the name of its solitary member. `stdin` and `stdout` are the special byte sources and sinks of almost the same names that are associated with any UNIX process. `cmd_line` refers to the sequence of command-line arguments that the shell makes available to C-based executables via the *argc-argv* mechanism. `cmd_line` is considered to be a stream of string tokens separated by unspecified delimiters. The number of tokens present is made available to the user as the builtin constant `nbr_of_args`.

`popkorn` is a special read/write I/O channel for interacting with a Korn shell co-process: just write shell commands into the `popkorn` channel and read/Read the output lines that come back. Use the PROC Reset_Popkorn to discard any unread, uninteresting characters before sending another command down but note that what Reset_Popkorn does is to read characters until it gets a special system-generated character that tells it that it has consumed the entire output of the preceding shell command and so it will hang if `popkorn` does not provide it with that character. Therefore, for example, do not call Reset_Popkorn twice in a row because the second one will hang. Note that the message protocol Daytona uses with `popkorn` involves the control characters `E` and `P` -- only disappointment will ensue if the output of the transmitted shell command contains these characters. Lastly, the PROC Kill_Popkorn_As_Needed is true to its name.

A `file` channel is one that provides access to a UNIX disk file. A `string` channel is one where the source or destination for the byte stream is a string in the program’s memory. The `text` kind of channel enables reading from and writing to TEXT objects. A `pipe` channel allows reading from or writing to (but not both) a UNIX pipe whereas a `bipipe` channel allows two-way communication between the program and a co-process. A `fifo` channel is the Cymbal manifestation of UNIX FIFOs or named pipes. (See the end of this chapter for an in-depth discussion of `CHAN(fifo)`.) `CHAN(fipifo)` is the union of `CHAN(file)`, `CHAN(pipe)`, `CHAN(fifo)`, `CHAN(funnel)`, `CHAN(tcp)`, and `CHAN(unix_domain)`: a `CHAN(fipifo)` VBL can assume values from one of several alternative new_channel calls that return either objects belonging to any of the alternatives of the specified union of `CHAN` types. `CHAN(funnel)`, `CHAN(tcp)`, and `CHAN(unix_domain)` are...
defined in Chapter 19 and Chapter 22.

As far as possible, the I/O fp's treat all I/O channels the same regardless of their kind.

A channel can operate in one of 8 modes relative to the Cymbal program it belongs to: _read_, _write_, _append_, _clean_slate_append_, _update_, _append_update_, _append_share_, and _clean_slate_update_. Each of the update modes (_update_, _append_update_, _append_share_, _clean_slate_update_) provides read and write access to the channel, with _update_ and _clean_slate_update_ initially positioning the channel cursor at the start of the channel and with _append_update_ and _append_share_ positioning it at the end. Also, _write_, _clean_slate append_ and _clean_slate_update_ begin by removing the previous contents of the channel, if any. Each of the _write_, _append_, _clean_slate append_, _append_update_, _append_share_ and _clean_slate_update_ modes will create the file, if it isn’t already there. When a file is opened with mode _append_, _append_update_, or _append_share_, it is impossible to overwrite data already in the file. _append_share_ is only for files -- see below for more information.

The constant channels _stdin_, _stdout_, _stderr_, _cmd_line_, and _popkorn_ are automatically and always open, are always available for use by referring to their names explicitly, and cannot be closed. (Actually, in the code synthesis case (Chapter 18), the user’s C code must open the constant channels by issuing a call to Initialize_Constant_Channels_Modulo with argument 1 if _popkorn_ is to be opened and 0 otherwise.) Other channels must be explicitly opened by using new_channel() and closed, when desired, by Close(). Is_Open is a PRED which can be used to determine whether a CHAN is currently open or not.

To explore these concepts, consider the following uses of new_channel:

```
local: STR: .line = "a b c d e f g "
    STR .file = "$HOME/file99"
set .in_chan = new_channel( for .file );
when( .new_channel_call_status != _worked_ )
{
    do Exclaim_Line( "error: unable to open new CHAN .in_chan" );
    do Exit( 1 );
}
set .out_chan = new_channel( via _string_ for .line with_mode _write_ )
otherwise with_msg "error: unable to open new CHAN .out_chan" do Exit(1);
```

This sample program begins by using the new_channel() function to open the file $HOME/file99 for reading, yielding a corresponding channel value which is then assigned to the variable in_chan. Note that the existence of the global new_channel_call_status VBL supports the use of the equivalent and more convenient otherwise construct that is illustrated in the second new_channel call.
new_channel() is a function which takes 11 keyword arguments: name, via, for, with_mode, with_patience, with_bufsize, with_locking, with_whole_msgs, with_msg_terminator, and with_user_sync. The optional name argument is used to identify the channel in error messages. The via argument, which defaults to _file_, is used to indicate the kind of channel. The default mode is _read_. On rare occasions, it is useful for a program to determine what kind of CHAN one has: to that end, via_for_chan will return the INT code that represents the kind of its CHAN argument. That code can be compared for equality to "via" Cymbal constants like _string_, which are C-declared to be the INTs specified in R.sys.h.

The for argument is used to specify where the stream of bytes for the channel is coming from or going to. For _file_ channels, the for argument is the UNIX file name which is free to be either the full or partial path and which may contain shell metacharacters. For _pipe_ and _bipipe_ channels, the for argument is the shell-level command which will invoke the desired sub-process; this shell command should fit all on one line without being grouped by braces or parentheses. For a _fifo_ channel, the for argument is the file system path of the FIFO. For _text_ channels, the for argument is the TEXT name itself. A positive with_bufsize argument value is taken to be the initial size of the I/O buffer for a _file_ channel or the initial size of the string buffer for a _string_ channel, although, for a _string_ channel, that size can increase (automatically) as needed. For a _fifo_ channel, a with_bufsize argument value of 0 causes the I/O channel to be unbuffered; a value of -1 causes sfio to choose a buffer of a size it considers optimal.

For _string_ channels, the STR(*) for argument is the string contents of the channel, in which case on opening a _string_ channel, its for argument is copied to a safe place where it constitutes the initial value for the _string_ channel. The channel will grow indefinitely as directed and that space is only freed up when the _string_ channel is closed.

The special for_alias argument is an alternative to the for argument. for_alias is used only for _string_ channels (i.e.,Q). It specifies that the string to be associated with the new channel is to be aliased from some specified STR VBL so that any subsequent changes to the CHAN(_string_) will be done to the STR VBL value itself. Not only does this save the string copy mentioned above but it also enables the power of CHAN(_string_) processing to be applied to any STR VBL. There is one very important caveat though: the user must take care when the channel is open to only change the STR
VBL string through CHAN(_string_) processing, not through any regular STR VBL operations. The reason for this requirement is that CHAN(_string_) processing involves its own bookkeeping which will be invalidated by exogenous changes to the string involved. The penalty for violating this stricture is all kinds of trouble including incorrect results and program abort. Caveat lector.

Furthermore, the system supports the for or for_alias argument containing interior null bytes, since STR(*) can. This unusual situation can sometimes present itself when working with stated_sizes.

Finally, .out_chan gets to be an output channel to an infinitely expandable string. The associated new_channel() call uses the via argument to specify that .out_chan will be a _string_ channel and it uses the for argument to say that the contents of line should be taken as the contents of that channel. The with_mode argument of the third new_channel call requires that the channel be an _update_channel instead of the default_read_.

8.1.1 Managing Concurrent Channel Access

If the optional keyword with_locking is present and via is _file_ or _text_, then when the new_channel is created, it will be created with the appropriate lock, i.e., a System V file lock of the appropriate type will be obtained immediately after opening the file. To be specific, a share (or read) lock on the file will be obtained for a file opened with _read_ mode or _append_share_ mode and an exclusive (or write) lock will be obtained otherwise.

The _append_share_ mode is relying on the UNIX semantics for file append which at the Cymbal level imply that buffers will be written atomically with "whole messages". However, since with_locking implies getting a share/read lock for _append_share_, multiple processes using _append_share_ can be appending to the same file "at the same time" possibly resulting in the interleaved output of whole messages. Remember that Daytona buffers output so that only whole messages are written out on buffer flush. The point of using _with_locking_ with _append_share_ is to lock out any other non-_append_share_ process that wants to change the file.

The with_patience argument is used to specify how patient the user is with regards to obtaining the new channel; it defaults to _wait_on_block_. Of course, in most cases, new channels are created immediately. But in the case of files to be opened with file locking, it may well be that the user cannot get the resource they want immediately. The with_patience argument enables the user to specify how long they are willing to wait: not at all, indefinitely, or some number of seconds in between. If with_locking and with_patience _wait_on_block_ are specified, then if a conflicting lock already exists, the process requesting the lock will wait until the lock can be granted. Both deadlock and exhausting the system maximum number of locks are two of the possible error conditions that will cause program termination in this case. If with_locking and with_patience equal to some positive integer, then if a conflicting lock already exists, the process requesting the lock will wait until the lock can be granted or until the time period has elapsed, whichever comes first. If with_locking and with_patience _fail_on_block_, then the event of either deadlock or a conflicting lock will result in a failure to open the file and will cause new_channel() to return the unusable _null_chan_.

The locking capabilities of new_channel() enable the user to write their own customized on-line backup of data files. All that is needed is to just write a Cymbal program that obtains a _read_ lock on the file via new_channel() (and in addition, then a lock file, if the file is a horizontal partition bin) and then copies that file to wherever desired and finishes by releasing the lock(s). This locking policy will
lock out any Daytona process that wants to modify the file. The user can also write their own waiting strategy in the event that they cannot immediately get the desired lock.

8.1.2 Error Handling For new_channel

As illustrated in the example above, a new_channel call makes available its status to subsequent code by means of setting the global new_channel_call_status variable. There are thirteen possible values for the status of a new_channel call: _worked_, _file_not_there_, _fopen_failed_, _deadlock_, _interrupted_, _timed_out_, _would_block_, _system_error_, _addr_in_use_, _conn_refused_, _conn_aborted_, _net_unreachable_, and _host_unreachable_. If a system call associated with the new_channel is interrupted by a signal, then _interrupted_ is the status. If new_channel is trying to open a file and the file cannot be found, the result is _file_not_there_ (errno equal to either ENOENT or ENOTDIR). Otherwise, if any fopen associated with the new_channel fails for any other reason than _interrupted_ or _file_not_there_, then _fopen_failed_ is the status of the call. There are of course many, many ways for an fopen, hence a new_channel, to fail; as it turns out, most of them cannot reasonably be handled at runtime, other than by program termination with a suitable error message. However, the user can always find out exactly what happened simply by calling the PROC Diagnose_Sys_Err before termination in order to be notified as to exactly the UNIX errno involved. _deadlock_ is raised when the operating system declines to get a lock that would otherwise cause deadlock. If an attempt to get a lock times out, then _timed_out_ is the result; If the new_channel patience is _fail_on_block_ and new_channel finds that some other process has a lock on the channel, then _would_block_ is the result. _system_error_ is the value used when the call fails to work for a reason other than the other possibilities. The last five error statuses can appear when using new_channel to open a socket; they are described in the Networking chapter. Here is an example of testing for _would_block_:

```plaintext
set .out_chan = new_channel( for "dUm...." with_mode_write_
   with_locking with_patience _fail_on_block_ )
otherwise_switch {
   case( = _would_block_ )
   {
      do Exclaim_Line( "error: failure to get lock: lock blocked" );
   }
   default {
      do Exclaim_Line( "error: failure to open out_chan" );
   }
}
```

8.1.3 redefined_channel

There are rare occasions when the user needs to redefine an existing I/O CHAN. For example, it may be convenient to redefine stdin or stdout to be associated with a file in the filesystem. This can be accomplished by calling redefined_channel:
otherwise_ok CHAN(_file_) FUN redefined_channel(
    this CHAN(_file_), 
    for STR = "",  
    with_mode INT = _read_, 
    with_patience manifest INT = _wait_on_block_, 
    with_bufsize INT = -1,  
    with_locking = −, 
    with_msg_terminator STR = "\n", 
    with_user_sync = −)

Obviously, redefined_channel only works for CHAN(file), including of course, stdin, stdout, stderr. The mandatory this argument refers to the CHAN(file) that is being redefined. The for argument is the filesystem path to be opened as the new source or sink of stream data for this channel. If the for argument is the empty string, then the only changes that will be made involve the other arguments, such as for example by changing the buffer size as in:

    set ? = redefined_channel( this _stdin_ with_bufsize 256*1024 ) 
    otherwise do Exit(1);

redefined_channel returns the same CHAN(file) it was called with when success, otherwise null_chan.

If a for argument is given, then the existing CHAN(file) is closed before being reopened with its new definition. (At the heart of this is a call to the stdio freopen function.) The principal value of this maneuver is to allow code that is Reading/Writing to a given CHAN(file) to remain unchanged while dynamically changing the definition of the CHAN(file) during run-time. Here is how it looks for redefining stdout (io.rdfc.1.Q):

    do Write_Line( "Hello from redefined_channel" );
    set ? = redefined_channel( this _stdout_ for "DuMmY" with_mode _write_ ) 
    otherwise do Exit(2); 
    do Write_Line( "Goodbye from redefined_channel" );

Note that the second Write_Line is going to put its output into the file DuMmY, not to the tty or any place else. Likewise note that redefined_channel can be used to cause error messages (sent to stderr) to start going to a filesystem file (or several in sequence) in the course of execution.

8.2 Writing

Writing is simple in Daytona in large part because Daytona automatically formats into (printable) strings the values of non-STRING scalar classes as well as values of certain COMPOSITE classes.
set .out_chan = new_channel( for "./dummy" with_mode _update_ );
to .out_chan   skipping 4
do Write( "x = ", .x, "\n" );
do Close( .out_chan );
do Exclaim_Line( "You just put the value of x into ./dummy" );

Notice that, not needing any further specification, Daytona will use a default format to print out whatever scalar value x has.

The to keyword argument specifies the channel to be written into. As a procedure, Exclaim is the same as Write except that its default value for to is _stderr_ instead of _stdout_.

The skipping keyword argument (which can be an expression) specifies how many new-lines will appear before the first item is written. The use of a _Line suffix on Write or Exclaim mandates that a new-line appears at the end of the last item written.

The import for Write is:

PROC: otherwise_ok Write(
    to CHAN = _stdout_ ,
    ( 0->1 ) skipping INT,
    ( 0->1 ) trailing INT,
    ( 0->1 ) ending_with STR,
    ( 0->1 ) with_sep STR,
    ( 0->1 ) flushing,
    ( 0->1 ) with_stated_sizes,
    ( 0->1 ) with_tuple_format STR, /* use only %s in printf format */
    ( 0->1 ) with_tuple_C_format STR, /* as in printf; caveat emptor */
    ( 0-> ) WRITABLE )

Regarding CHAN(_string_), please note that an initially empty _string_ CHANNEL can be opened for output and subsequent Writes will automatically extend the string as needed. As remarked later, it is also possible to seek in and rewind _string_ CHANNELS, thus enabling previously written characters to be read or overwritten.

trailing is the converse of skipping in that a number of new-lines equal to the trailing argument (which can be an expression) is written to the output channel after writing all specified WRITABLE arguments. If a with_sep STR argument is provided, then all WRITABLE arguments will be separated by an instance of the with_sep argument. An ending_with argument is a STR that will appear at the very end of the output of Write procedure; this is a feature convenient for sending messages terminated by a protocol-specific string into pipes and sockets.

If a flushing argument is present in the call, then whatever is written to the channel will be flushed out to the channel immediately after it is written; otherwise, it will be flushed whenever UNIX thinks it is appropriate.

Write_Line is equivalent to a Write with a trailing 1 argument whereas Write_Words is equivalent to a Write_Line with a with_sep " " argument.

TUPLEs and conventional (fixed-size) ARRAYs can be written out as units. (Eventually, this
capability will be extended to other aggregate types.) Consider this way to write out a TUPLE-valued dynara:

```plaintext
local: TUPLE[ INT .order_cnt, FLT .sum_qty, INT .max_days_open ]
ARRAY[ INT(_short_) .supp_nbr ] .supp_stats
for_each_time .supp_nbr is_such_that( .supp_stats[ .supp_nbr ] = ? )
{    
do Write_Words( .supp_nbr, .supp_stats[ .supp_nbr ] );
}
```

This results in a series of lines, one for each mapping in the associative array, where each line contains four items separated by spaces: the .supp_nbr and its associated .order_cnt, .sum_qty, .max_days_open.

As this indicates, Write flattens out any structured items it is called upon to write:

```plaintext
local: INT ARRAY[4] .ara1
do Write_Words( %.ara1, [ .4, .z ], [% .ara1], [], [5, [.y,[7],8],.a] );
```

The above just results in a sequence of scalar values on a line separated by spaces.

The `with_tuple_format` and `with_tuple_C_format` keywords enable the user to provide a C printf-style formatting string for specifying how the output is to be printed. `with_tuple_format` is safer to use because the only printf format specifications allowed are the ones for strings, i.e., the ones based on %s. As is customarily the case for Write calls, Daytona will automatically cause the print items to be converted to string types for output, and in this case, then printed according to the printf format specification.

On the other hand, `with_tuple_C_format` is much more dangerous because it allows the user to use any printf format specification in conjunction with Daytona not converting print items to string types for output since that is what the printf format specifications are for. While Daytona verifies that the number of % specifications in any STR constant format equals the number of arguments given to the Write to print, it does not yet do any further typechecking (nor could it ever if the `with_tuple_format` argument were a variable dereference or a function call). As C programmers know all too well, type mismatches between the % specification and the argument actually given to it frequently result in core-dumps. Consequently, anyone who uses `with_tuple_C_format` arguments must take responsibility to use them with care and to examine their usage closely in the event of executable core-dumps.

This all being said, however, probably the most pleasant way to specify the interleaving of text and variable values is to use ISTRs, the interpolated strings defined in Chapter 5, as in:

```plaintext
do Write_line( "The file in question, .file, does not have permissions .perm"ISTR );
```

### 8.2.1 Writing Binary Objects By Writing With Stated Sizes

One of the significant characteristics about writing out values separated by delimiters is that the delimiters allow the values they delimit to be of varying length. On the other hand, if all the items are of stated/known size, then there is no need for delimiters to identify where one value ends and another begins because the offsets alone are sufficient to do that.

When the `with_stated_sizes` keyword is used, Daytona will Write out each item in the argument list.
in a fixed, i.e., constant, number of bytes as determined by the type of the item. In this stated_sizes context, types are interpreted somewhat differently than usual. For example, a STR(k) VBL where k is some positive integer is taken to have values of size exactly k, instead of having values of size at most k, as would otherwise be the case. In this regard, if a STR(k) VBL has been assigned a value of size less than k, then when it is Written with stated_sizes, the entire k bytes allocated to the VBL is written anyway, including the bytes following the first null character, which bytes the user may or may not consider to be random garbage. This supports Writing with stated_sizes sequences of known length of arbitrary, possibly non-ASCII, bytes that may intentionally contain several null characters.

In this stated_size situation, Daytona will also be happy to write out the current value of an STR(*) VBL: note that since this is precisely the value that the user put there, the happy consequence is that there will not be any possibility of garbage or nulls being written out as an appendage (as in the previous case re STR(k)). However, such values cannot be read back in again using stated_sizes unless the length() of the STR(*) value is also written out (as a binary INT) in such a way that the “reading” Cymbal program can read that first and then use that now-known length to read in the previously written STR(*) value. Obviously, a convenient way to do that is to use stated_sizes to write out the INT length first, then followed immediately by the STR(*) value. All this same logic applies to the other supported types in this category, i.e., RE, LIT, THING, HEKA, and HEKSTR, as well as in general any type implemented using a Byte_Seq_Box such as VIDEO. On the other hand, obviously Daytona must deny any request to Write out a STR(=).

Note that in their binary representations, INTs and FLT_s are of fixed/known size within their subclasses such as _long_. So, Daytona does the obvious thing in the stated_sizes context, which is to Write out INTs, UINTs, FLOATs, IP2s, and IP6s, in their binary, machine representations of a length in bytes that corresponds to the C-implementation of their class/subclass. For example, an INT(_long_) will be written as a 4 byte, machine-dependent binary quantity when with stated_sizes has been specified.

As seen in statedsiz.?Q, it is very simple to do this kind of Writing:

```plaintext
local:
  INT .x = 47
  INT(_short_) .x2
  STR(4) .y = "abcd" /* exactly 4 in this case */
  STR(4) .y2 = "ab"
  STR(4) .y3 = ""
  HEKA(8) .h = ~123`HEKA(3)
  HEKSTR(8) .h2 = ~12345678`HEKSTR(8)
  IP .iii = ~12.12.12.111`IP
  IP2 .iii2 = ~10.10.10.222`IP2
  FLT .z = 98.6

set .ochan = new_channel( for "TmP" with_mode _write_ );
to .ochan with_stated_sizes
  do Write( 10?, "ABCDE", 25000, .x, .y, .y2, .y3, .h, .h2, .z, .iii );
```

Copyright © 2013 AT&T
All Rights Reserved.
September 15, 2013
Skolems such as \texttt{10} in this Writing context are considered to represent the indicated number of null bytes, in this case, 10. Optionally, one could use \texttt{(10)}.

Please note that as of this time, there is no support for Writing with \texttt{stated\_sizes} \texttt{directly} any other types; this includes such types as \texttt{DATE}, \texttt{CLOCK}, \texttt{TIME} and \texttt{BITSEQ} (although such is technically possible for those types) and it includes any and all \texttt{COMPOSITE} types. Also, the only type of \texttt{CHAN} that does \texttt{not} support \texttt{with\_stated\_sizes} \texttt{is \_cmd\_line\_}, obviously.

### 8.2.2 How To Format Items For Written Output

As mentioned before, any term (i.e., constant, dereferenced variable, or function call) can be written out. The user need not be aware of the datatypes of the items written; if not strings, they will be converted to strings as necessary (and possible). On the other hand, the following functions can be used to gain exact control over how items for output are formatted. A \texttt{WRITABLE} is any datatype that can be converted into a \texttt{STR} for printing purposes.

\texttt{str\_for\_dec} \texttt{( FLT, INT .dec places, INT .options = \_plain\_ )} returns the string value of rounding its first argument to the number of decimal places specified by its second argument. If the second argument is \_no\_dot\_, then the number is rounded to 0 places and no decimal point is printed. The four possible values for \texttt{.options} are: \_plain\_, which means unadorned (i.e., without any of the other options), \_fin\_, which means using the parentheses of financial-minus for negative numbers, \_1000s\_, which means using commas to demarcate groups of three whole number digits, and the now obvious \_fin\_1000s\_.

\texttt{lb} Also known as \texttt{left\_block( WRITABLE, INT, STR = " " )}. This function left-justifies its \texttt{WRITABLE} first argument in a block of copies of the third argument of total length equal to the value of the second argument.

\texttt{rb} Also known as \texttt{right\_block( WRITABLE, INT, STR = " " )}. This function right-justifies its \texttt{WRITABLE} first argument in a block of copies of the third argument of total length equal to the value of the second argument.

\texttt{cb} Also known as \texttt{center\_block( WRITABLE, INT, STR = " " )}. This function centers its \texttt{WRITABLE} first argument in a block of copies of the third argument of total length equal to the value of the second argument.

\texttt{db} Also known as \texttt{decimal\_block( FLT, INT, INT )}. This function rounds its \texttt{FLT} first argument to the number of places specified in its third argument and right justifies the result in a block of blanks of length equal to the value of the second argument. If the third argument is \_no\_dot\_, then the number is rounded to 0 places and no decimal point is printed.

\texttt{zb} Also known as \texttt{zero\_block( FLT, INT, INT )}. This function rounds its \texttt{FLT} first argument to the number of places specified in its third argument and right justifies the result in a block of zeros of length equal to the value of the second argument. If the third argument is \_no\_dot\_, then the number is rounded to 0 places and no decimal point is printed.
Also known as financial_block( FLT, INT, INT ). This function rounds its FLT first argument to the number of places specified in its third argument and right justifies the financial version of the result in a block of blanks of length equal to the value of the second argument. In financial syntax, negative numbers are represented not by using a minus sign but rather by enclosing the quantity in parentheses. If the third argument is _no_dot_, then the number is rounded to 0 places and no decimal point is printed.

cmb

Also known as STR FUN ( INT(_huge_)|FLT .nbr, INT .size, INT .round = 0, BOOL .fin = _false_ ) comma_block. This function rounds its FLT (or INT(_huge_)) first argument to the number of places specified in its third argument and right justifies the result in a block of blanks of length equal to the value of the second argument. If the fourth argument is _true_, then negative numbers are represented not by using a minus sign but rather by enclosing the quantity in parentheses. If the third argument is _no_dot_, then the number is rounded to 0 places and no decimal point is printed. Last but not least, the property that gives the function its name is that the string of digits to the left of the decimal point is grouped into groups of three by using commas as in (1,234,567.230).

fcmb

Also known as STR FUN ( INT(_huge_)|FLT .nbr, INT .size, INT .round, BOOL .fin = _false_ ) financial_comma_block. This operates the same as comma_block() with the exception that negative numbers are printed using the financial notation with parentheses.

rtn

Also known as round_to_nearest( FLT, FLT ). This function returns a FLT, not an STR, which is the result of rounding the first argument to the nearest multiple of the second argument. Sample calls are rtn( .x, .001 ) and rtn( .y, 2.5 ).

Also, regarding output formatting, the filters Daytona uses to convert UNIX flat files to tables and packets are discussed in an appendix.

### 8.2.3 Writing TUPLES, BOXes, and ARRAYS

As a convenience, Cymbal supports Writing VBL dereferences for such COMPOSITE types as TUPLEs, BOXes, and ARRAYs. Here is a simple example:

```plaintext
local: INT ARRAY[ 2, 2 ] .myara = [ 1, 2, 3, 4 ]
with_sep " -> " do Write_Line( .myara );
```

and the output is:

```
1 -> 2 -> 3 -> 4
```

TUPLE VBLs may also be written out as a group as in:

```plaintext
local: TUPLE[ INT, STR, DATE, BITSEQ ] .tu
  = [ 47, "hello", ^1/1/84^, "101"B ]
with_sep " -> " do Write_Line( .tu );
```

The aggregates in the preceding examples had a fixed number of components. However, Daytona will also Write out the BOX and ARRAY(_dynamic_) values of VBLs. In all these cases, the rule is that
if the separator is not specified, then it is a new-line (agg.write.1.Q).

Generally speaking, once a channel for writing has been opened, writing proceeds smoothly. In the unlikely event that a writing error has occurred, Daytona will call Exit_Clean with argument 101.

8.2.4 Handling Write And Flush Errors

Ordinarily, Writes are executed without incident. If an error does occur, then Daytona will abort the program with an error message. There are occasions, especially writing to a CHANNEL that is a TCP socket to another computer, where Write errors of certain kinds will happen with some regularity. In those cases, the user may want to gain programmatic control over such error events and handle them without doing a forced automatic abort and exit. To this end, the otherwise_ok feature is supported for the Write family and Flush as illustrated by:

```plaintext
to .remote_chan
do Write_Line(.x)
otherwise_switch{
  case{ = _broken_pipe_ }{ go cleanup_and_service_next_request; }
  default{ do Exit(2); }
}
```

Since the otherwise clause is considered to be modifying the Write call, there cannot be a semicolon at the end of the list of WRITABLES being written out. When used with otherwise, these PROCS set the write_call_status VBL which the otherwise mechanism can hook into. Here are the different possibilities for error:

```
write_call_status values

_worked_
_interrupted_
_conn_reset_
_host_unreachable_
_stale_nfs_
_broken_pipe_
_deadlock_
_link_down_
_timed_out_
_device_full_
_quota_exceeded_
_filesize_exceeded_
_system_error_
```

_interrupted_ means that the system received an interrupt signal (SIGHUP) while the call was in progress. If the write call attempts to write on a pipe/socket that has no listener, then _broken_pipe_ is the result. If the write call is on a socket which has been closed at the other end, then
8.3 Reading

Reading is not as simple as writing.

8.3.1 The Reading Paradigm

Daytona’s paradigm for reading from I/O channels is to tokenize a stream of bytes into a sequence of string tokens which are then converted to appropriate datatypes for assignment to specified variables or comparison to specified values. The tokenizing takes place in one of 6 ways:

1. One or more delimiter characters are specified and tokens are defined as the strings not containing delimiters that are separated or terminated by exactly one delimiter character.

2. One or more delimiter characters are specified and tokens are defined as the strings not containing delimiters that are separated or terminated by one or more consecutive delimiter characters.

3. The input is considered to be a sequence of messages terminated by (multi-character) message terminator STRINGs.

4. The tokens can be specified by using a regular expression argument for a matching keyword in exactly the same way as is done in a tokens() call. For non-string CHANNEL, the matching will necessarily be confined to one line at a time.

5. Each token is defined by a known type of fixed size or length. For example, the CHANNEL may consist of a sequence of four-byte little-endian integers.

6. For the _cmd_line_ CHANNEL, the tokens are the strings provided to the program by the argc/argv mechanism, excluding argv[0] (i.e., the program name). The _cmd_line_ channel has the unusual property that it is conceived to consist of a sequence of tokens (if any) separated by an unspecified delimiter character and then followed by an infinite sequence of unspecified delimiter characters. As will be seen, this supports a nice default value mechanism in the event that not enough arguments are given on the command line.
8.3.2 The read FUN

The read function has the following prototype, which will be explained immediately, step by step:

LIST FUN: read(
  from CHAN = _stdin_ ,
  /* default: upthru " \t\n" */
  ( 0->1 ) upto STR,
  ( 0->1 ) upthru STR,
  ( 0->1 ) matching STR|RE|CRE,
  ( 0->1 ) with_stated_sizes,
  ( 0->1 ) with_star_sizes manifest TUPLE[ (0->) INT ],
  ( 0->1 ) ending_with STR,
  ( 0->1 ) but_if_absent manifest TUPLE,
  ( 0->1 ) with_default_bia,
  ( 0->1 ) with_no_default_bia
)

Here is an example of a use of the read function:

    local: STR .n; DATE .dd; INT .a
    set .in_chan = new_channel( for "/usr/ralph/some_data" );
    set [ .n, .dd, .a ] = read( from .in_chan upto ":\n" );

(In practice, it is very important to write additional code to handle any errors that may occur in reading; error handling for reads will be addressed shortly.)

In this example, the file "/usr/ralph/some_data" is being opened for reading (by default) and three tokens are being read out of it. By means of the upto argument, these tokens are defined as being maximal strings terminated by single occurrences of either a colon or a new-line. So, for example, suppose the contents of the file (including a trailing new-line) were:

Thomas Jefferson:7-4-1826:83

Then, the read call causes 3 string tokens to be identified by the colon and new-line delimiters: "Thomas Jefferson", "7-4-1826", and "83". As is always the case, Daytona automatically converts these values to the appropriate types and assigns them so that "Thomas Jefferson" becomes the STR value of n, "7-4-1826" becomes the DATE value of dd, and 83 becomes the INT value of a.

Note that the read returns LISTs of values and so therefore, a LIST-oriented assignment must be used. This is true even if just one item is being read, as illustrated by the mandatory use of brackets around .x in:
set [.x ] = read( from "/usr/ralph/dummy" );

Sequences of delimiters will be considered as one delimiter when the upthru keyword is used. This corresponds to the second tokenizing mode listed above. This mode of reading is useful, for example, when the tokens to be read are separated by one or more white-space characters. In order to read the next token, this mode of reading proceeds as follows: delimiter characters are read upto the start of the next token, the characters consisting of the next token are read as delineated by reading the first (and only the first) delimiter character following the token. This concludes Daytona's work in processing this token. The same procedure is used in processing subsequent tokens.

Delimiters used as terminators require that every token be immediately followed by one or more delimiters; delimiters used as separators require that tokens be separated by one or more delimiters. The only place where the two concepts offer a distinction is at the end of the channel. When using delimiters as separators, it is permissible for the last token to be immediately followed by the end of the channel; when using delimiters as terminators, every token, including the last one, must be immediately followed by one or more delimiters. Consequently, it is an error when delimiters are used as terminators for what would otherwise be the last token to be followed immediately by the end of the channel.

Whether or not the delimiters specified by the upto or upthru arguments are used as terminators or separators depends on the kind of CHANNEL. They are considered to be separators for _string_ and _text_ channels only. They are considered to be terminators for UNIX-file-type channels, meaning the non_string, non_cmd_line channels. upto and upthru arguments are not allowed for the _cmd_line_ since it is automatically considered to be tokenized by some (unidentified) delimiter.

Daytona provides two helpful read-based function name abbreviations: read_line is the same as read with an upto "\n" argument and read_words is the same as read with an upthru " \t\n" argument. If a read call has no upto or upthru argument, then upthru " \t\n" is assumed if that is permissible for the channel being read from.

If for some reason (like wanting to facilitate subsequent tokenizing by delimiters), one wants to read a line and retain the new-line, then here is the best, most efficient way to do it:

```
set .line = read_line( ) otherwise do Exit(1);
set .line += "\n";
```

Another important way to tokenize input is by using an ending_with keyword argument to divide up the channel into a sequence of messages terminated by the (multi-character) STR ending_with argument. ending_with is a generalization of the upto in that the upto argument is a single character terminator (or separator, if reading from a CHAN(_string_)) whereas an ending_with is a typically multi-character terminator (regardless of CHAN subclass). Note that ending_with is also distinct from upto, which specifies as a delimiter an indefinitely long string of bytes taken from some specified set: the difference can be seen by observing that the ending_with terminator is a specific string with a specific order and identity for each of the bytes which make it up.

The tokens determined by ending_with are thought of as messages. Since they can contain any sequence of characters that does not match the terminator STRING, it is quite possible for a message to contain many new-lines and other syntactic complications that would not otherwise be possible when
using \textit{upto} delimiters. The message construct is quite useful when conceptualizing how to send information back and forth over pipes and sockets.

8.3.3 Reading From Channels By Matching REs

Alternatively, instead of using \textit{upto} or \textit{upthru}, the \textit{matching} keyword can be used with an RE (regular expression) argument to indicate how to tokenize the input. Tokenizing takes place using exactly the same syntax employed by the \textit{tokens} function described in Chapter 10; just make sure that the number of variables being read into equals the number of tokens specified using \texttt{(\(\backslash(\ \backslash(\)\)\)\()\)\)\) in the RE. Notice that each time an RE pattern is matched in the channel, the read-position cursor is advanced the length of the matched string so that the next read from the channel will occur at the following byte. When reading from a non-\texttt{_string}_ CHANNEL, such as a CHANNEL(file), please be aware that processing is done one line at a time and thus, no RE will match across a new-line and furthermore, the RE must not contain a new-line since the system strips that away as a result of working one line at a time. Also, when a match fails, \_\texttt{read\_call\_status} becomes \_\texttt{failed\_re\_match}\_ (and \_\texttt{RE\_Match\_Worked} and \_\texttt{RE\_Match\_Failed} are set appropriately). Since the notion of a missing value (which is juxtaposed delimiters) is orthogonal to regular expression matching, \texttt{but\_if\_absent} and \texttt{with\_default\_via} are disallowed when using \textit{matching}.

8.3.4 Missing Values In I/O Channels

The same notion of missing value that is used for data records is also supported for reading tokens terminated/separated by single delimiters. When single delimiters are being used as terminators, then adjacent delimiters indicate a missing value. Otherwise, when single delimiters are being used as separators, not only do adjacent delimiters indicate a missing value, but so also does the situation of where a delimiter is followed immediately by the end of the channel.

Since the purpose of a \texttt{read} statement is to produce values to assign to variables, when a missing value in the data is encountered, there is just no value there to assign to whichever variable was expecting to get a value at that point. Consequently, the reading stops immediately (i.e., no further values are read), the \_\texttt{missing\_value\_call\_status} (described shortly) is set, the target variable’s value is not changed, and execution continues at the statement following the \texttt{read} call. Consider, for example, the following \texttt{read} from \_\texttt{stdin}:

\begin{verbatim}
local: STR .x = "-no-value-
   do Write( "Please enter a value for x: " );
   set [ .x ] = read( upto "\n" );
   do Write\_Line( ".x = ", .x );
\end{verbatim}

When running this program, if the user simply types new-line in response to the prompt for a value for \texttt{x}, then the output will be: \texttt{.x = -no-value-}. Some people are surprised by this: they think that if you just type \texttt{<new-line>}, then \texttt{x} should get \"\" as its value. But that would not be in accord with Daytona’s missing value philosophy, which is that "You can’t work with something that is not there". As with data records, the absence of a value between delimiters is taken to mean the missing-value situation, not the empty string. Note that if \texttt{x} was an INT VBL or a DATE VBL then it certainly could not get \"\" as a value: thus this missing value philosophy makes it unnecessary to determine what
value other than "" should the variable get in these cases.

Fortunately, the same **but_if_absent** construction that is used for Cymbal descriptions to provide on-the-spot defaults for missing values can be used here:

```plaintext
local: STR .x = "-no-value-

do Write( "Please enter a value for x: " );

set [ .x ] = read( upto "\n" but_if_absent[ "" ] );
do Write_Line( ".x = ", .x );
```

In general, the argument for **but_if_absent** is a TUPLE of default values which will be assigned when no value is read for the variable corresponding to them positionally in the sequence of tokens being read. If there are more variables to get values than there are elements of the **but_if_absent** TUPLE, then the last item in the **but_if_absent** TUPLE will be used for the remainder. Any **but_if_absent** TUPLE elements not paired with target variables are ignored. *bia* is a convenient abbreviation for **but_if_absent**. In the following example, the default value for zz is ".

```plaintext
set [ .xx, .yy, .zz ]
= read( from .in_chan upto ":" but_if_absent[ 0, "" ] );
```

Using **but_if_absent** when missing values are allowed is a good fail-safe practice to employ in order to ensure that variables have the values the user may tend to expect. It is equally important to always test reads for success or failure as will be explained shortly.

As remarked above, for **cmd_line** alone, the end of the channel is considered to be extended by what amounts to be an infinite sequence of missing values. This can be a very convenient situation because in the event that no (or not enough) arguments are given on the command line, then the program can work with default values instead:

```plaintext
set [ .xx, .yy, .zz ]
= read( from _cmd_line_ but_if_absent[ 0, "" ] );
```

If a variable that is being assigned to by a **read** call does not have an explicit definition, then its type will be inferred from its corresponding **but_if_absent** value, if any, or will else otherwise be considered to be STR(*). This latter may not be what the user wants since a token "45" will not then be automatically converted into an INT.

There is yet another alternative to handling missing values. Instead of generating a **missing_value_error** or using a value specified by **but_if_absent**, the user may direct Daytona to treat any missing value instead as an instance of the initial value for the type. All Daytona types have an initial value specified with the **with_cy_init_val** keyword in the CLASS definitions in **sys.env.cy**. When a **read** call uses the **with_default_bia** keyword, then this is exactly what occur: when a missing value is encountered, it will be treated as an occurrence of the missing value for the associated type. An example from io.7.Q:
Here is the output:

```
| 0 | 0.0 | 9999-12-31 | 0
```

Note that none of the values these variables had before the `new_channel` call have been preserved.

Obviously, no `read` call can fail with `_missing_value_` if `with_default_bia` is being used.

### 8.3.5 Detecting And Handling I/O Errors

Reading is probably the one operation in any procedural language which is the most fraught with peril. It's like having children: you never know what you're going to get. There are a variety of ways for a `read` call to fail but there is only one way for one to succeed, namely, all the variables get values, whether by reading them from the channel or by assigning them `but_if_absent` defaults. This can easily be determined by the `Read_Worked` predicate or its opposite, `Read_Failed`, as in:

```plaintext
set [ .xx, .yy, .zz ]
   = read( from .in_chan but_if_absent[ 0, "" ] );
when( Read_Failed ){
    do Exclaim_Line( "error in reading from .in_chan" );
    do Exit( 2 );
}
```

Every `read` call also sets the global `read_call_status` VBL to have one of the following status values:
If the `read` call worked, then `.read_call_status` is `_worked_`. If it failed due to finding a missing value, then the value is `_missing_value_`. If the delimiters are terminators, then any would-be token abutting the end of the channel results in a `_missing_terminator_` status. If the token read cannot be converted to a value of the required type, then `_type_mismatch_` is the result. If the `read` is using a *matching* regular expression argument to tokenize and the RE pattern cannot be matched, then `_failed_re_match_` is the status of the `read` call. If the token is unexpectedly and unacceptably long, then `_overflow_` is the result. If there is an attempt to read beyond the end of the channel so that not so much as a single character is read, then `_instant_eoc_` is the result. In other words, just reading up to and including the last byte in a channel is not enough to trigger `_instant_eoc_` -- the user has to cause Daytona to try to get the next byte after that (which is not there) -- and then `_instant_eoc_` is raised. If, when reading with _stated_sizes_, Daytona is asked to read a negative number of bytes, then `_negative_read_` is the result. For _file_-type channels, if `ferror` returns true, then if the call was interrupted by SIGINT, then `_interrupted_` is the result. If the `read` call is on a socket which has been closed at the other end, then `_conn_reset_` is the result. If the `read` call is going against an NFS filesystem that becomes inaccessible, then `_stale_nfs_` is the result. Otherwise, then `_system_error_` is the result. The `_failed_comparison_` status is discussed below in conjunction with the `Read` procedure.

When `_overflow_` occurs during the execution of a transaction, the transaction will abort; otherwise, `_overflow_` causes the program to exit with status 101. An error message is printed in either event. When `_type_mismatch_` occurs, a warning message is printed out and execution continues, giving the programmer the opportunity to recover from this problem. When the other statuses occur, no message is printed out, the read call terminates immediately (i.e., with no further input processing), `.read_call_status` is set, and execution continues.

The statuses `_missing_value_`, `_missing_terminator_`, `_type_mismatch_`, `_failed_comparison_`, and `_failed_re_match_` all constitute syntax errors. The `Syntax_Error` predicate is `true` if and only if the preceding `read` or `Read` call experienced a syntax error. `Syntax_OK`
is the opposite.

A read or Read call will cause the At_Eoc predicate to be True if and only if that call encountered the end of the channel in the process of reading. At_Eoc not only occurs with read call status _instant_eoc_ but also with _missing_ _terminator_ , _missing_value_ (in the event that a separator is the last character in a channel), and in fact, other call statuses such as _type_mismatch_ when the last (illegal) value in a separator-based channel is followed immediately by the end of the channel. In short, if the little character-reading drone for a Cymbal read/Read statement bumps into the end-of-the-channel, At_Eoc is set to be True (and False otherwise). At_Eoc takes a CHAN argument which is _prev_chan_ by default, meaning the CHAN just previously read from; otherwise, At_Eoc will report the value for its designated CHAN argument.

As a convenience, read calls may use the otherwise feature, so that the above read example may be rewritten as:

```plaintext
set [ .xx, .yy, .zz ]
= read( from .in_chan but_if_absent [ 0, "" ] )
otherwise {
  with_msg "error in reading from .in_chan" do Exit(2);}
```

For each read call, the global nbr_values_found variable is set to equal the number of variables which got values as a result of the read, either from channel tokens or from defaults.

Each read (or Read) of one or more values from a non-_cmd_line_ channel defines the Eol_Seen PRED so that it is True if the read encountered one or more new-lines and False otherwise. (This only occurs however when one of the delimiters is a new-line.) Eol_Seen takes a CHAN argument which is _prev_chan_ by default, meaning the CHAN just previously read from; otherwise, Eol_Seen will report the value for its designated CHAN argument.

Be sure to use these variables and predicates that are set as the result of a read/Read call only after the call. For example, here is a typical mistake, potentially leading to subtle misbehavior:

```plaintext
while( !At_Eoc ){
  set [ .x19 ] = read( upto " " from .in_str_chan but_if_absent [ "" ] );
  do Write_Line( .x19 );
}
```

Note that At_Eoc is being used before it has an opportunity to be set by the read it is clearly intended to control. The danger is that, on entrance to the loop, the previously read channel may not be .in_str_chan. A better way to write this code is:

```plaintext
while( !At_Eoc[.in_str_chan] ){
  set [ .x19 ] = read( upto " " from .in_str_chan but_if_absent [ "" ] );
  do Write_Line( .x19 );
}
```

This is relying on the fact that At_Eoc is False immediately after the new_channel call for .in_str_chan. An even better and safer way to write this code is:
loop {
  set [ .x19 ] = read( upto " " from .in_str_chan but_if_absent [ "" ] );
  when( Read_Failed ) break;
  do Write_Line( .x19 );
}

Notice that here all the different ways for the read to fail are being tested for.

### 8.3.6 Reading UNIX Flat Files

Not infrequently, a user will want to read lines out of a file consisting of single-delimiter-separated fields, as illustrated by the colon-separated fields in:

Thomas Jefferson:7-4-1826:83
John Adams:7-4-1826:91

This can be handled adequately as above by:

loop {
  set [ .n, .dd, .a ] = read( from .in_chan upto ":\n" )
  otherwise_switch {
    case( = _instant_eoc_ ){ break; }
    case( = _missing_value_ ) do { ... }
    else do { ... }
  }
}

However, observe that if by design or by error, lines have differing numbers of colons, this style of reading will freely continue on to the next line to get values for the read variables, if not enough are present on the line it began on. In other words, it treats new-lines just like colons.

To gain further control over reading in this situation, the following two-stage approach is useful. First, read each line into a _string_ channel and second, read tokens from the _string_ channel:
loop {
    set [ .line ] = read( from .in_chan upto "\n" )
    otherwise_switch {
        case( = _instant_eoc_ ){ break; }
        case( = _missing_value_ ) do { ... }
        else do { ... }
    }
    set .str_chan = new_channel( via _string_ for .line )
    otherwise do Exit(2);
    set [ .n, .dd, .a ] = read( from .str_chan upto ":" )
    otherwise_switch {
        case( = _instant_eoc_ ){ break; }
        case( = _missing_value_ ) do { ... }
        else do { ... }
    }
    do Write_Words( .n, .dd, .a );
    do Close( .str_chan )
}

Notice how convenient it is for new-line to be a terminator in the _file_ channel and colon a separator in the _string_ channel. By using this second paradigm, the tokenizing of each line (and the handling of its errors and idiosyncrasies) can be handled completely in a compartmentalized way. For example, it is possible, in the event of surprises, to Rewind .str_chan and read it again using another strategy.

8.3.7 Read PROCEDURE

The Read procedure extends Cymbal’s reading capabilities by allowing the user to require that specified tokens have specified values (instead of considering those tokens to be new values for variables). For example, suppose a file contained lines like these:

```
Serial_Nbr = 721194
Salary = 44000
Occupation = Construction
```

Then, such a file could be read by:

```
local: STR .ssnbr, INT .sal, STR .occu
        from .in_chan
    do Read( "Serial_Nbr", "=", ssnbr );
        when( Read_Failed ) { ... }
    do Read( "Salary", "=", sal );
        when( Read_Failed ) { ... }
    do Read( "Occupation", "=", occu );
        when( Read_Failed ) { ... }
```
Notice that the third argument in each Read call is a VBL, not a VALCALL (i.e., a dereferenced VBL as in .x). VBL arguments to Read are considered to be alias arguments and what Read will do for each alias VBL argument is to assign to it a suitably typed value, if one can be found. Currently, no type inference is supported for Read alias VBLs: consequently, they must be defined explicitly lexically before their use in a Read call. Here is the import for Read:

```plaintext
PROCEDURE: Read(
    from CHAN = _stdin_,
    /* default: upthru " \t\n" */
    ( 0->1 ) upto STR,
    ( 0->1 ) upthru STR,
    ( 0->1 ) but_if_absent TUPLE,
    ( 0->1 ) matching STR|RE|CRE,
    ( 0->1 ) with_stated_sizes,
    ( 0->1 ) with_star_sizes manifest TUPLE[ (0->) INT ],
    ( 0->1 ) ending_with STR,
    ( 0->1 ) but_if_absent manifest TUPLE,
    ( 0->1 ) with_default_bia,
    ( 0->1 ) with_no_default_bia,
    ( 0-> ) OK_OBJ )
```

The zero-or-more OK_OBJ include alias VBLs, arbitrary expressions (i.e., constants, VALCALLS, and FUNCALLS), and ?s (i.e., skolems). The remaining arguments to Read have the same meaning they do for read. Here is an example:

```plaintext
local: INT .x, .z = 21
loop {
    from .in_chan upthru " \n" do Read( ?, "=" , x , "->", .z+2 );
    when( Read_Failed )
    {
        when( At_Eoc ) break;
        else { ... }
    }
    ...
}
```

In short, Read makes a strong distinction between variables and their values. If x is a variable, then if .x appears as an argument in a Read call, then Read will attempt to literally read the value of x; it will produce a _failed_comparison_ status if that value is not the possibly converted value of the appropriate token. On the other hand, if x itself appears as an argument in a Read call, then Read will cause the token that it reads at the appropriate position to be assigned as the new value of x, converting it to x’s type as necessary.

And, more generally, any scalar term can appear as the argument to Read. If it is a constant like 26 or ‘1-1-84’ then the corresponding token must equal that constant, after any appropriate datatype conversion. If it is a function call, then the function call will be evaluated and the result must appear in
the input stream at the required position. Finally, any token corresponding to a skolem (i.e., ?) is just read and ignored. Furthermore, but if absent defaults are insensitive to the presence of skolems, or to put it differently, but if absent defaults are used in sequence for the non-skolem VBLs only that are being Read.

This distinction between variable and value of variable is at the root of why a C variable would appear prefixed with "&" in a scanf statement but would not tend to appear prefixed with "&" in a printf statement (unless, of course, the programmer wanted to print the address of that variable). In fact, Read has essentially the same functionality as scanf with the additional feature that the user can define the tokenizing delimiters for Read while they must accept white space alone as the tokenizing delimiters for scanf. Furthermore, Read compares values on the basis of type (e.g., DATE) as opposed to just the string comparisons provided by scanf. Read also does not employ a separate format string for indicating the types of the values to be read.

8.3.8 Reading Binary Objects By Reading With Stated Sizes

There would be little point in writing stated_size data if it weren’t possible to read it back in. This is as simple as can be (when not reading a Byte_Seq_Type like STR(*)): just declare the appropriate VBLs with stated_size types and read them in by using a with_stated_sizes keyword (statedsiz.?Q):
Both the `read` function and `Read PROC` can be used to read with `stated_sizes`; each allows the use of skolems like `15?`. In the `stated_sizes` context, a single skolem `?` is considered to be one byte; hence, to read `15?` is to read fifteen bytes. However, note that past 8 bytes (for `_fifo_CHAN`), reading multiple skolems is done efficiently by seeking. As before with Writing, types like `STR(4)` are taken to be of size exactly 4.

In order to read `STR(*)` quantities, as well as, in general, quantities for any type implemented by a `Byte_Seq_Box`, it is necessary to convey to Daytona at runtime how long those quantities are. This is done by means of the `with_star_sizes` `TUPLE` argument. Here is an example taken from `statedsiz.3.Q`:

```plaintext
local:
  INT .x = 47
  INT(_short_) .x2
  STR(4) .y = "abcd" /* exactly 4 in this case */
  STR(4) .y2 = "ab"
  STR(4) .y3 = ""
  HEKA(8) .h = `123`HEKA(3)
  HEKSTR(8) .h2 = `12345678`HEKSTR(8)
  IP .iii = `12.12.12.111`IP
  FLT .z = 98.6

set .ochan = new_channel( for "TmP" with_mode _write_ );
to .ochan with_stated_sizes
  do Write( 10?, "ABCDE", 25000, .x, .y, .y2, .y3, .h, .h2, .z, .iii );
do Close(.ochan);

set .ichan = new_channel( for "TmP" );
from .ichan with_stated_sizes
  do Read( 15?, x2, x, y, y2, y3, h, h2, z, iii );
when( Read_Failed ) with_msg "bad read" do Exit( 2 );
do Close(.ichan);
  with_sep "<->" do Write_Line(.x2,.x,.y,.y2,.y3,.h,.h2,.z,.iii);
do Unlink( "TmP" );
}
local: STR(1) .sz_str
  STR .str1, .str2
set .ichan = new_channel( via _string_ for "8 asdfasdf 5 uvxyz 0 3 abc" )
  otherwise do Exit(1);
do Rewind(.ichan);
set [ .sz_str, ?, .str1, ?, .sz_str, ?, .str2 ] =
  read( from .ichan with_stated_sizes
    with_star_sizes [ (INT).sz_str, (INT).sz_str ]
  ) otherwise do Exit(3);
_Say_Eq(.str1,"asdfasdf")
_Say_Eq(.str2,"uvxyz")

Once again, the purpose of the with_star_sizes TUPLE argument is solely to provide the sizes for the values of the VBLs whose types are implemented using a Byte_Seq_Box, like the STR(*) VBL str1 -- not for any other VBL, like the STR(1) sz_str. Note that expressions for these star sizes are (lazily) evaluated at runtime as they are needed. Thus the first component of the star_sizes TUPLE will evaluate to 8 and the second will evaluate to 5. A very useful convention is that if there are more star_size VBLs than there are star_sizes, then the last expression in the TUPLE is used again and evaluated repeatedly as needed. Thus, a with_star_sizes TUPLE [ (INT).sz_str ] could replace the one above with the same effect.

As it turns out, these star_size VBLs support missing values, whereas nothing else in the stated_size scenario does. A missing value for a star_size VBL occurs when its (runtime) stated length is 0. In this case, default values can be used as illustrated by:

local: STR(1) .sz_str
  STR .str1, .str2
  STR ARRAY[4] .str
set .ichan = new_channel( via _string_ for "8 asdfasdf 5 uvxyz 0 3 abc" )
  otherwise do Exit(1);
do Rewind(.ichan);
set [ .sz_str, ?, .str1, ?, .sz_str, ?, .str2, 
  read( from .ichan with_stated_sizes with_star_sizes [ (INT).sz_str ]
    bia[ "N/A" ] // the last default here is replicated as needed
  ) otherwise do Exit(3);
_Say_Eq(.str1,"asdfasdf")
_Say_Eq(.str2,"uvxyz")
_Say_Eq(.str[3],"N/A")
_Say_Eq(.str[4],"abc")

One restriction is that when reading with_stated_sizes, no comparisons with constants are supported yet. There are only five read_call_status values possible when reading with_stated_sizes: _worked, instant_eoc, missing_value, negative_read, interrupted, and system_error. missing_value can only appear when STR(*) are being read.
8.3.9 Error Handling For The Read PROCEDURE

The same PREDICATES Read_Worked, Read_Failed, At_Eoc, Syntax_Error, Syntax_Ok and global variables read_call_status and nbr_values_found used for read have the same meaning and are used in the same way for Read. The same call statuses used for read are also used for Read. Read also uses the _failed_comparison_ call status to indicate that the current converted-as-needed token fails to equal the corresponding expression provided by the Read call. The call terminates immediately in such a situation and execution continues with the statement following the call.

_type_mismatch_ can also occur when a comparison is required and it is not possible to convert the current token to a value of the required type. A warning message is printed in this case and the call terminates immediately in such a situation and execution continues with the statement following the call.

8.3.10 Reading conventional ARRAYs and TUPLEs

Analogous to Writing, the user can read/Read (conventional fixed-dimension) ARRAYs and TUPLEs as the aggregates themselves:

```plaintext
local: TUPLE[ INT, STR, DATE, BITSEQ ] .tu, .tu2
       INT ARRAY[4] .ara
set .tu = read_words();
set [.tu, .tu2] = read_words();
do Read_Words( tu, tu2, ara );
set .ara = read_words();
```

8.3.11 Read PROC Miscellany

Any but _if_absent_ argument provides values for assignment purposes only: it does not provide values for comparison purposes.

Daytona provides two helpful Read-based function name abbreviations: Read_Line is the same as Read with an _upto_ "\n" argument and Read_Words is the same as Read with an _upthru_ " \t\n" argument. If a Read call has no _upto_ or _upthru_ argument, then _upthru_ " \t\n" is assumed, if that is permissible for the channel being read.

The Eol_Seen PRED is _true_ after a Read that encountered a new-line (given that one of the delimiters is a new-line); it is _false_ otherwise.

8.4 Bipipe Sorting Example

Here is an interesting I/O example involving bi-directional pipes that shows how to use an external sort package in the middle of a Cymbal query (bipipe.2.Q):
locals:
    INT( _short_ ) .num
    STR(30) .supp
    STR(25) .city

set .chan = new_channel(via _bipipe_ for "sort -t '|' +2" with_mode _update_ );

set .out_cnt = 0;
for_each_time [.xnum, .xsupp, .xcity] is_such_that {
    there_isa SUPPLIER named .xsupp where(
        Number = .xnum
        and City = .xcity)
    ) do {
        set .out_cnt++;
        when( .out_cnt > 10 ) break;
        to .chan with_sep "|" do Write_Line(.xnum, .xsupp, .xcity);
    }
with_mode _write_ do Close(.chan);

loop {
    from .chan upto "|\n" do Read(num, supp, city);
    when( Read_Failed )
    {
        when( At_Eoc ) break;
        else {
            with_msg "error: Read failed with status .read_call_status"ISTR
                do Exit( 2 );
        }
    }
}

do Write_Line("City = ", lb(.city,25), "Name = ", .supp, " (", .num, ")");
do Close(.chan);

    Note the special call to Close() which causes the _write_ half of the bipipe to be closed so that
    sort(1) can receive EOF and continue on with its work.

8.5  _fifo_ CHANS

A CHAN(_fifo_) is a one-way pipe with a name, meaning that it is a special file created in a
filesystem and can thus be located by the associated path. Consequently, one significant advantage that
_a_fifo_ has over a _pipe_ is that processes which do not have the parent-child relationship required for
_a_pipe_ can nonetheless communicate from one to the other by means of opening and using a _fifo_
that exists and persists independently of them; as opposed to being known solely by a dynamic
runtime file descriptor integer, a _fifo_ can be located simply by its externally known filesystem path. Typically, one or more processes write into a _fifo_ whereas one process reads from it. As long as the amount written by a process into a _fifo_ is less than PIPE_BUF, the size of a pipe on the system, then that write occurs atomically. This means that without any other synchronization mechanism being employed, all of that write’s bytes get placed into the pipe in sequence without being interleaved or overwritten by any other process’s writes. (PIPE_BUF is 5120 on Solaris, 4096 on Linux, and 8192 on SGI, for example. See the likes of Solaris’ /usr/include/sys/param.h.)

There are four modes available when using new_channel to open a CHAN(_fifo_): _read_, _write_, _update_, clean_slate_update_. Obviously, a process that opens a CHAN(_fifo_) with mode _read_ can only read from it; likewise for _write_. In either case, the new_channel will block until there is another process attempting to or having done the opposite thing, at which point it will succeed. In particular, a writer will wait to open its CHAN(_fifo_) until there is a reader on the other end ready to receive any message subsequently sent. This is good -- after all, there is no point in being able to write unless you know there is someone able to read what you’ve written. On the other hand, there is an advantage to being able to read even when no-one is currently interested in writing, just on the off-chance that a writer will eventually appear. This is the case for daemon processes which wish to run indefinitely and handle messages sent by writers as the latter come and go. Such a reader can be obtained by using the _update_ mode. In all these cases, the given process will create the _fifo_ if it isn’t already there, whereas a process using clean_slate_update_ will create a new _fifo_ in any event and then function as a _fifo_ with mode _update_. Note that a syslog process that writes log records for a variety of other kinds of processes could be implemented by a single _update_ _fifo_ reader reading a _fifo_ that is being written into by a variety of other processes as they come and go.

Attempting to read from a _read_ _fifo_ that is empty and has no processes opening it for writing will result in _instant_eoc_; on the other hand, in that same situation, an _update_ _fifo_ will simply block waiting for a writer to appear with a message.

Since writes to a CHAN(_fifo_) are buffered, the entire amount written can only be guaranteed sent to the _fifo_ if the write is flushed.

Here is test query (fifo.1.Q) illustrating _fifo_ use:
local: STR: .file, .input, .output = "Check, check."
set [ .file ] = read( from _cmd_line_ but_if_absent [ "my_fifo" ] );

// two processes must be used else the new_channels will block waiting for each other
set ? = new_tendril( executing {
    set .fifo_writer = new_channel(via _fifo_ for .file with_mode _write_)
    otherwise{ do Exit(1); }
do Write_Line( "Wrote and flushed: ", .output );
    to .fifo_writer do Write(Line( .output );
    do Close( .fifo_writer );
});

set .fifo_reader = new_channel(via _fifo_ for .file with_mode _read_)
    otherwise{ do Exit(1); }
from .fifo_reader do Read_Line( input );
do Write_LINE( "Read: ", .input );
when( .input != .output ) do {
    do Exclaim_Line(  
        "error: fifo_1.Q: Input ( .input ) != Output ( .output )"ISTR );
    do Exit( 1 );
}
do Write_LINE( "Input ( .input ) == Output ( .output )"ISTR );
do Close( .fifo_reader );
do Unlink(.file);
_Wait_For_Tendrils

This has to use TENDRIL(_clone_) to get separate processes so that both can activate the _fifo_ for use at the same time and return from new_channel.

8.6 Miscellaneous CHAN Fpps

There is a CHAN datatype for channels although the user will rarely see it used since Daytona can usually figure out on its own when a variable is a channel variable or not. But if needed, then a definition would be done like:

    locals: CHAN( _file_ ): .out_file = _stdout_

The user may find useful the following CHAN procedures, most of which have stdio analogs:
Flush flushes the entire contents of the associated buffer — there are no exceptions. In particular, any message protocol that is in place is ignored (see Chapter 19). If the optional with_fsync is used with a _file_ CHAN, then the UNIX fsync(2) (for file sync) command will be executed after the user-space buffer for this CHAN has been flushed to kernel space. This will then cause the kernel-space buffers to be written to disk before this Flush command returns.

Close closes the CHAN or TENDRIL and thus removes all space assigned to and traces for that corresponding CHAN or TENDRIL from the process. If the optional with_fsync is used with a _file_ CHAN, then a with_fsync do Flush(...) is executed before the actual Close. By default, both the _read_ and _write_ capabilities (if present) are closed. In the cases of a CHAN(_bipipe_) or TENDRIL with downlink _converse_, it is possible to close down half of the connection by specifying the mode to be removed, as in with_mode _write_ do Close(...).

When Close is called with an otherwise option, then it has the same possibilities for error that Write/Flush do plus in addition _failed_kid_, _killed_kid_, _stopped_kid_ which can occur when a _pipe_ or _bipipe_ runs into abnormal termination. In that case, the INT VBL pipe_status gives the exit code or signal as appropriate.

The Seek_In PROC is used to change the current position in the CHAN by seeking to a specified offset from a specified position. The from keyword argument used in Seek_In may have the values _next_, _start_, and _end_. (Recall that the nature of offsets is that they begin at 0 so that seeking to offset 0 from the _start_ leaves the current position at the start.) Rewind is effectively a shorthand for doing a Seek_In to the _start_ of the CHAN. For a non-_cmd_line_ CHAN, chan_offset returns the offset in bytes from the beginning of the CHAN of the next byte to be read/written in the specified CHAN. For the _cmd_line_ CHAN, chan_offset returns the ordinal position minus 1 of the next item in the channel to be processed. On UNIX systems that support it, Truncate will truncate (i.e., discard) all bytes in a UNIX file CHAN that appear after the to byte number. In other words, if the size of the file is 25 bytes and the to argument is 20, then after the call to Truncate, there will be 20 bytes in the file. For a CHAN(_string_), the string contents are truncated to the offset argument given from the start but the malloc’d space housing the former occupant is left unchanged. str_for_chan returns an STR(*) consisting of the current contents of a _string_ CHAN. This can be thought of as a cast and so a more aesthetic way to apply this function is illustrated by (STR).my_str_chan.

An idiom useful when creating/writing an _update_ CHAN(_string_) for subsequent reading is to execute this after the last Write:
to chan_offset( .rw_schan ) do Truncate( .rw_schan );

This command will put down an end-of-channel marker at the indicated position, thus preparing the channel for subsequent reading.

Some perspective on Daytona’s I/O capabilities might be obtained by comparing it to that of awk. One major difference is that Daytona uses the channel construct to identify accesses instead of file or pipe names as awk does. So, in awk, no analog of new_channel is needed since file or pipe names are used as access identifiers. Cymbal thereby has increased functionality; for example, the channel construct enables Daytona users to make simultaneous _read_ accesses to the same file. In summary comparison with awk, I/O in Cymbal has the additional functionalities provided by the channel construct, concurrency control, the _update_ modes, the scanf capabilities of Read, and the ability to seek around in channels.
9. Declarative Cymbal

Declarative Cymbal assertions make use of the ands, ors, and nots of propositional logic and the variables, functions, predicates, and quantifiers of first-order logic. In addition, they may make use of Cymbal descriptions, which are special database-oriented constructs, as well as intensional Cymbal boxes (i.e., generalized set- and list-formers). This chapter describes how Daytona processes declarative Cymbal assertions, particularly the ones whose purpose is to specify a set of TUPLEs that satisfy them.

One of the most important ways to work with declarative Cymbal assertions is the Display procedure, which even though it is a procedure, is primarily declarative in nature because its task is to display in a formatted way a TUPLE of answers for each possible way of satisfying a given declarative assertion. The full semantics and characteristics of Display are given in this chapter.

The chapter concludes with a comparison of Cymbal’s design philosophy with that of other query languages.

9.1 Declarative vs. Procedural Semantics

Most Daytona users have much more experience programming procedurally than they do declaratively. A couple of definitions will make this distinction clear.

A query (or request) is declarative if it defines its answers by means of a characterizing assertion they (and they alone) must satisfy. A query is procedural if it defines its answers by means of a sequence of actions that must be taken over time in order to construct the answers. (A query is multiparadigm if it is neither declarative nor procedural but rather is composed of declarative or procedural subqueries.)

In other words, if the answers are described, i.e., their uniquely identifying characteristics are stated, then their specification is declarative. On the other hand, if the query describes a process which specifies how to make a system go from one state to another over time, then their specification is procedural.

For example, a regular expression is declarative because it describes a pattern that fits every string that matches it. A regular expression is not an algorithm consisting of a sequence of steps that a computer could follow to determine whether or not some string matched the pattern. Such an algorithmic specification is embodied in the finite automaton that is produced when a regular expression is compiled.

In a procedural language, assertions are evaluated just for their truth values. In a logic-based declarative language, assertions are sometimes evaluated for their truth values but most significantly, they are also processed in such a way that they generate new values for variables.

Since Cymbal uses both declarative and procedural constructs, it is very important for the user to be aware of when they are using either kind of construct. The tell-tale clues to look for are these: If the construct is an assertion possibly constructed from satisfaction claims and descriptions using logical
connectives, then that construct is declarative. Assignment statements, conditional branches, loops, and procedure calls are procedural. In fact, anything that contains a *do*, *when* or *time* in it is procedural. Also, parentheses are used to group declarative constructs while braces are used to group procedural ones.

### 9.2 Display

In its simplest form, a *Display* procedure call takes a logic assertion with an associated tuple of free variables and prints out a display of all tuples of constant values for those variables which satisfy that assertion. A TUPLE of constant values \([ o_1, \ldots, o_n ]\) *satisfies* an assertion \(A\) with free variables \([ V_1, \ldots, V_n ]\) if and only if when all occurrences of \(V_i\) in \(A\) are replaced with \(o_i\), the result is a true assertion. (Later, this definition will be extended slightly to handle ancillary variables.) For example, \(x\) and \(y\) are the free variables in:

\[
.x = 1 \text{ and } .y = 2 \\
\text{and there_exists .z such_that( .z Is_In [ .x -> .y ] )}
\]

and \([ 1, 2 ]\) satisfies this assertion since

\[
1 = 1 \text{ and } 2 = 2 \\
\text{and there_exists .z such_that( .z Is_In [ 1 -> 2 ] )}
\]

Consequently, for the Display call:

\[
do \text{ Display each[ .x, .y ] } \\
each\_time( \\
.x = 1 \text{ and } .y = 2 \\
\text{and there_exists .z such_that( .z Is_In [ .x -> .y ] ) } )
\]

\([ 1, 2 ]\) is an answer TUPLE (and, in fact, a simple argument will prove that it is the only answer TUPLE). This particular Display call takes two keyword arguments. The each argument is a TUPLE specifying the answer components to be produced and the each_time argument is a *BoundedAsn* that characterizes the answers. (Recall that a *BoundedAsn* is an assertion in parentheses or a Cymbal description.)

The semantics of the above Display call have been explained in a non-procedural manner. There are procedural semantics for understanding the non-procedural Display concept but those semantics will be presented after further discussion of the declarative semantics of Display. Notice that the declarative semantics of Display have an oracular character: answer TUPLEs are unveiled with not so much as a clue as to *how* the system generated them. One just verifies that in fact the purported answer TUPLEs do indeed satisfy the assertion (which is what it means to be an answer). and one can also develop an argument or proof that no other TUPLEs can satisfy the assertion.

For users new to declarative languages, the following procedure will help to ensure correct declarative understanding of Display calls. First, generate a TUPLE of permissible values for the each TUPLE variables. Second, using a screen editor, edit a copy of the each_time assertion and go through the assertion and literally replace each variable reference (like \(.x\)) with its corresponding permissible value (like 1025 or "John Jones"). Next, ask yourself, "Is this resulting assertion true?". If it is,
Display will produce it as an answer; if it is not, Display will not. At this point, it should be apparent as to whether the assertion accurately reflects what the user wants to ask or not. This exercise is an excellent way to ground one’s understanding of declarative, logic-based languages; it may be a little tedious to perform but the reward for doing it a few times is that you’ll never need to do it again and best of all, you’ll be able to easily understand Display calls.

Note that Display has a sort of amphibious character: a Display call is considered to be a procedure call which can be positioned within a procedural Cymbal program just like any other procedure call. However, when it comes to determining what it is that a Display call will print, then the non-procedural character of the call itself becomes apparent: it prints tuples which somehow satisfy a logic assertion. (Even though there are procedural semantics in addition to the non-procedural ones for Display, Display is nonetheless considered to be a fundamentally non-procedural or declarative construct by virtue of its fundamental reliance on the notion of assertion satisfaction.)

Continuing with further examples of Display, calls like the following may also be issued:

\[
\text{do Display each } \cdot x \text{ each_time there_is_a SUPPLIER named } \cdot x ;
\]

Here the presumed scalar \( \cdot x \) is considered to be a singleton TUPLE. Each answer TUPLE to this query is a singleton TUPLE consisting a Name for a SUPPLIER object.

Consider the following disjunctive query:

\[
\text{do Display each } \cdot x \\
\text{each_time(} \\
\quad \cdot x = 1 \text{ or } \cdot x = 2 \\
); 
\]

Both \([ 1 ]\) and \([ 2 ]\) are answer TUPLEs for this query because both of the following are true:

\[
1 = 1 \text{ or } 1 = 2 \\
/******* and *******/
2 = 1 \text{ or } 2 = 2
\]

\([ 3 ]\) is not an answer TUPLE because the following is false:

\[
3 = 1 \text{ or } 3 = 2 
\]

### 9.2.1 Display Output Formats

The default output format for Display is the DC data format, which thereby enables query output to be loaded directly into the database, if desired.

The output produced by:
do Display each[ .x, .y ]
each_time(
    (.x = 1 and .y = 2)
    or (.x = 3 and .y = 4)
);

is:

%msg1)
%msg2) Query File: test.Q
%msg3)
%msg4)
%msg5) recls)
%msg6) flds) X|Y
%msg7)
1|2
3|4

(To verify that these are answers, just verify that they satisfy the assertion, as was done above.)

In effect, a simple Display call is essentially just a for_each_time loop which prints out its variables. For example, the following for_each_time loop prints out the same answers as the preceding Display call, although without the %msg comments.

    for_each_time [ .x, .y ]
    is_such_that(
        (.x = 1 and .y = 2)
        or (.x = 3 and .y = 4)
    ){
        with_sep "|" do Write_Line( .x, .y );
    }

In short, Display calls are a kind of for-each-time-print whereas for_each_time loops are for-each-time-do .

The with_format option enables users to choose additionally from among the _table_, _packet_, _xml_, _safe_, and _desc_ formats. For example, the answers to:

    with_format _table_
do Display each[ .x, .y ]
each_time(
    (.x = 1 and .y = 2)
    or (.x = 3 and .y = 4)
);

are printed as:
Query File: test.Q

----
X   Y
----
1   2
3   4
----

The _packet_ format answers are:

TEST_1
X=1
Y=2

TEST_2
X=3
Y=4

The _xml_ format answers are:

<TMP_BUNCH>
<TMP>
  <X>1</X>
  <Y>2</Y>
</TMP>
<TMP>
  <X>3</X>
  <Y>4</Y>
</TMP>
</TMP_BUNCH>

Note how similar these are to the _desc_ format answers, which are:
Description-format answers are ideal for input to backtalk.

The _safe_ format is used by Daytona’s implementation: it is unlikely that users will have occasion to use it directly. The _safe_ format is used to safely transmit records over pipes and sockets as occurs when records are sent to the output formatters DC-prn and DC-pkt by Display calls or DS Show invocations or as occurs when pdq is sending records to clients. The problem that is being solved here is that, since these records have to be divided into fields by delimiters, havoc can ensue if the values of these fields contain those delimiters. This can surely happen for the in-memory representations of the SAFE_STR, ISO8859, and BASE64 types but it can even happen for the STR type on occasion. Transmitting these records in their DC (data file) representation is not attractive since it depends so much on the arbitrary choice of delimiters and of course, some data may be computed by a query and have no clear relationship to an rcd specification of delimiters. This variety of representation puts an undue burden on the clients receiving these records. So, the _safe_ format is used instead and it is very simple and reliable: it simply uses the SAFE_STR delimiters and any value that would or probably would conflict with those delimiters has its in-memory form converted to its BASE64 representation for the purposes of transmission.

9.2.2 More Display Examples

The following query yields a cross-product of answers:

```sql
with_format _table_
do Display each[ .x, .y ]
each_time(  
  (.x = 1 or .x = 2)  
  and (.y = 3 or .y = 4)  
)
```

which yields a cross-product set of answers:
Query File: test.Q

----
X  Y
----
1  3
1  4
2  3
2  4
----

It can be rewritten equivalently using BUNCHes and INTERVALs as:

```plaintext
with_format _table_
do Display each[ .x, .y ]
each_time(
  .x Is_In { 1, 2 }
  and .y Is_In [3->4]
);
```

[ 2, 3 ] is the sole answer to the following more complicated query:

```plaintext
do Display each[ .x, .y ]
each_time(
  (.x = 1 or .x = 2)
  and
  if( .x = 2 )
    then( .y = 3 )
    else( .y = 4 )
  and
  .x != 1
  and
  .y > .x
  and there_exists .z such_that( .z = .x + .y )
);
```

because [ 2, 3 ] satisfies the assertion in that:
(2 = 1 or 2 = 2)
and
if( 2 = 2 )
    then( 3 = 3 )
    else( 3 = 4 )
and
2 != 1
and
3 > 2
and there_exists .z such_that( .z = 2 + 3 )
and because the only other possibilities would have .x = 1 and those can’t work.

Displays are also frequently used to query a database:

    with_title_line "Get phone numbers of suppliers from St. Paul"
    with_format _table_
    do Display
    each [ .supplier, .phone ]
    each_time(
        there_is_a SUPPLIER where(
            Name = .supplier
            and Telephone = .phone
            and City = "St. Paul"
        )
    )

All of Cymbal is designed to be readable as passable English; conversely, if someone’s Cymbal doesn’t read passably well as English, it is probably not correct. In this case, the above Display call would read:

With title line "Get phone numbers of suppliers from St. Paul", with format _table_, do Display each tuple of values for the pair of variables supplier and phone each time there is a SUPPLIER record where the value of the Name attribute equals the value of the supplier variable and the value of the Telephone attribute equals the value of the phone variable and the value of the City attribute is "St. Paul".

For the sample database that comes with Daytona, the value pairs for supplier and phone that satisfy the assertion are given in the following output:
Query File:  test.Q

Get phone numbers of suppliers from St. Paul

--------------------------------
Supplier       Phone
--------------------------------
Acme Shipping  612-149-5678
Bouzouki Receiving  612-943-7416
Julius Receiving       612-309-3492
Sunshine Warehouse     612-303-7074
--------------------------------

Notice how with_title_line keyword argument has produced a title in the output.

9.3 OPCONDS

In order to fully understand the semantics of Display, it is necessary to introduce the OPCOND construct which either explicitly or implicitly underlies all of Cymbal’s declarative constructs. An open-coded condition or OPCOND is a construct that enables the user to present an assertion together with a statement of what some of that assertion’s free variables are, along with, optionally, their types. While the presence of OPCONDS in queries is usually hidden by other constructs such as Display calls, for_each_time loops, aggregate function calls, and intensional box definitions, OPCONDS do have their own stand-alone syntax which may be employed on occasion. Here are three equivalent Cymbal OPCONDS:

\[
\sim[ .x, .y ] \text{ such_that} ( \\
.x \text{ Is_In } \{ 1, 2 \} \text{ and } .y \text{ Is_In } [ 3->4 ] ) \\
\sim[ \text{VBL} x, \text{VBL} y ] \text{ such_that} ( \\
.x \text{ Is_In } \{ 1, 2 \} \text{ and } .y \text{ Is_In } [ 3->4 ] ) \\
\sim[ \text{INT VBL} x, \text{INT VBL} y ] \text{ such_that} ( \\
.x \text{ Is_In } \{ 1, 2 \} \text{ and } .y \text{ Is_In } [ 3->4 ] )
\]

These are equivalent because the type inference system will deduce that the first two are abbreviations for the third. These OPCONDS assert that x and y are free variables for the accompanying assertion. (There may be other free variables in the OPCOND assertion that are not in the OPCOND variable list. These are called outside variables as discussed below.)

OPCONDS are like in-line PREDICATEs, i.e., PREDICATEs whose definitions are given explicitly when invoked. In other words, OPCONDS can be used syntactically like SimplePreds:
[1, 4] \sim \{ .x, .y \} \text{ such that (}
  .x \text{ Is_In } \{ 1, 2 \} \text{ and } .y \text{ Is_In } \{ 3 \rightarrow 4 \})

/** is equivalent to saying **/
1 \text{ Is_In } \{ 1, 2 \} \text{ and } 4 \text{ Is_In } \{ 3 \rightarrow 4 \}

Notice that the latter assertion has resulted from replacing all occurrences of the free variables \( x \) and \( y \) with the corresponding values 1 and 4. In this way, OPCONDs are the declarative analogs of the lambda-expressions found in functional programming. Here is the relevant grammar production:

```plaintext
\text{OpCond} ::= \sim [ \text{VblSpecSeq} ] \text{ such that BoundedAsn ;
```

The scope of the VBLs in \text{VblSpecSeq} is \text{BoundedAsn}, which is also called the matrix of the OPCOND. By definition, each of the variables in \text{VblSpecSeq} must appear free in the matrix. The scope of every OPCOND either explicitly or implicitly begins with a \text{somehow} which serves to existentially quantify every variable in the scope that does not otherwise have a scope. Those variables that are scoped by an OPCOND’s \text{somehow} are said to be \text{OPCOND_SOMEHOW variables}. Those variables that appear free in the OPCOND as a whole and whose scope properly includes the OPCOND are said to be \text{outside variables}. For example, both \( p \) and \( q \) are outside variables for the OPCOND that appears in the following assertion:

\[
\begin{align*}
  &.p = 0 \mid = 1 \\
  &\text{and} \\
  &.q \text{ Is_In } \{ 5 \rightarrow 7 \} \\
  &\text{and} \\
  &[1, 4] \sim \{ .x, .y \} \text{ such that (}
    .x \text{ Is_In } \{ .p, 2 \} \text{ and } .y \text{ Is_In } \{ 3 \rightarrow .q \})
\end{align*}

/** which is equivalent to saying **/
\[
\begin{align*}
  &.p = 0 \mid = 1 \\
  &\text{and} \\
  &.q \text{ Is_In } \{ 5 \rightarrow 7 \} \\
  &\text{and} \\
  &1 \text{ Is_In } \{ .p, 2 \} \text{ and } 4 \text{ Is_In } \{ 3 \rightarrow .q \}
\end{align*}
\]

Although they are not always syntactically visible, there are OPCONDs underlying the following principal declarative constructs:

- Display calls
- \text{for_each_time} loops
- Aggregate function calls
- Intensional boxes
9.3.1 OPCONDS In Display Calls

Display calls can actually take OPCONDS as arguments:

```plaintext
do Display
tuples_satisfying [ .x, .y ] such_that(  
  .x Is_In { 1, 2 }  
  and .y Is_In [ 3->4 ]  
);  
```

produces exactly the same output as:

```plaintext
do Display each[ .x, .y ]  
each_time(  
  .x Is_In { 1, 2 }  
  and .y Is_In [ 3->4 ]  
);  
```

In fact, understanding the full semantics of Display rests on understanding that at root, Display takes one declarative argument and that is an OPCOND. In other words, Daytona rewrites Display calls that have `each` and `each_time` arguments into calls that have a single OPCOND argument for the keyword `tuples_satisfying`. In this case, the rewrite is straightforward.

However, in general, the components of a Display’s `each` TUPLE can be terms (see Chapter 5 for the definition of `term`). Display will only output scalar terms or TUPLE/STRUCT-valued terms. The meaning of a Display call in general is taken to be that of the corresponding OPCOND-based Display created in the following manner. The general Display form:

```plaintext
do Display
each [ t_1 \cdots t_n ]
each_time( assertion_A )  
additional_keyword_args  
```

where the \( t_i \) are arbitrary terms is rewritten by Daytona using potentially new variables (here represented by \( v_1 \cdots v_n \)) as:

```plaintext
do Display
tuples_satisfying [ v_1, \cdots, v_n ] such_that(  
  assertion_A  
  and .v_1 = t_1  
  and \cdots  
  and .v_n = t_n  
)  
additional_keyword_args  
```

where for each \( i, v_i \) is a new, unique, system-generated variable, unless \( t_i \) is a simple variable.
dereference,

in which case, \( .v_i \) is exactly \( t_i \). In other words, a new variable is introduced only if \( t_i \) is a constant, array element, tuple/structure member, or a function call; otherwise, the simple, original \( t_i \) variable itself is used as the OPCOND variable. Of course, in the latter case, the equality appended is of the form \( .x = .x \) and consequently has no effect.

The exact reasons for this different treatment for different kinds of terms are given in Chapter 12 where a similar transformation for box-formers is defined (see "Technical Digression: An Undesirable Alternative" in Chapter 12). Note that this treatment of variable dereferences for Display each-lists corresponds to the way they are treated for for_each_time variable lists, i.e., as new scoping locations for the variables. On the other hand, when a non-variable-dereference term appears in the each-list, outside variables are allowed to influence the computation of the term.

In the meantime, as an example, consider:

```plaintext
set .z = 5;
    with_format _table_
do Display each[ .x, .y, (.x + .y + .z)**2, 0 ]
each_time(
    .x Is_In { 1, 2 }
    and .y Is_In [ 3 -> .z ]
);
```

Observe that the outside variable \( z \) appears once in the each TUPLE and once in the each_time assertion. Also, observe that the each TUPLE contains not only variable dereferences but also function calls and constants. This Display call is rewritten by Daytona into:

```plaintext
set .z = 5;
    with_format _table_
do Display
    tuples_satisfying -[ .x, .y, .v1, .v2 ] such_that(
    .x Is_In { 1, 2 }
    and .y Is_In [ 3 -> .z ]
    and .v1 = (.x + .y + .z)**2
    and .v2 = 0
    );
```

The resulting output is:
9.3.1.1 Protecting Display Calls From Outside Elements

The fact that Display calls (and other OPCOND-related Cymbal declarative constructs) so easily and silently import variables from their Cymbal environment can lead to the odd surprise from time to time when implicitly scoped variables are used. For example,

```cymbal
with_format _table_
do Display each[ .x+10, .y ]
each_time(  
    .x Is_In { 1, 2 }  
    and .y Is_In [ 3 -> 4 ]  
    and .z = (.x + .y) % 2  
    and .z < .x
);
```

has output

```
----------
  Field_1  Y
----------
   11  3
   12  3
   12  4
----------
```

whereas different answers result when \( x, y, \) and \( z \) become outside variables:

```
----------
  X  Y  Field_3  Field_4
----------
   1  3   81    0
   1  4  100    0
   1  5  121    0
   2  3  100    0
   2  4  121    0
   2  5  144    0
----------
```
set .x = 2;
set .y = 5001;
set .z = 1;
   with_format _table_
do Display each[ .x+10, .y ]
each_time(
   .x Is_In { 1, 2 }
   and .y Is_In [ 3 -> 4 ]
   and .z = (.x + .y) % 2
   and .z < .x
);
yields

----------
Field_1  Y
----------
  12    3
----------

Actually, due to the nature of the transformation of the Display to using an OPCODE, only x and z (not y) are outside variables for the Display’s each_time assertion. The y in the assignment is in fact a different variable with the same name as the y used in the Display call. Nonetheless, the output of the Display call was changed by introducing and setting variables outside the call. This could be a very unwelcome surprise.

Total protection from the effects of introducing or deleting outside variables can be provided for Display calls by using only variable dereferences in the each-list and explicit quantification for the OPCODE SOMEHOW variables. In this way, all variables can be given explicit scopes. (Another option is to use the (explicit) OPCODE form for the call.) For the query above, the protection looks like this:

set .x = 2;
set .y = 5001;
set .z = 1;
   with_format _table_
do Display each[ .x_term, .y ]
each_time(
   there_exists [ .x, .z ] such_that(
   .x Is_In { 1, 2 }
   and .y Is_In [ 3 -> 4 ]
   and .z = (.x + .y) % 2
   and .z < .x
   and .x_term = .x+10
));

The values for x and z outside of the Display call have no impact on the result of the call, which,
therefore is the same as it was in isolation.

Incidentally, the OPCOND form for Display’s is handy when it is desired to explicitly state the types of the variables:

```plaintext
set .x = 2;
set .y = 5001;
set .z = 1;

with_format _table_
do Display
tuples_satisfying \[ \text{INT}(_\text{short}_).x\_term, \text{INT}(_\text{short}_).y \]
such_that(
  there_exists \[ \text{INT}(_\text{short}_).x, \text{INT}(_\text{short}_).z \] such_that(
    .x Is_In \{ 1, 2 \}
    and .y Is_In [ 3 -> 4 ]
    and .z = (.x + .y) \% 2
    and .z < .x
    and .x\_term = .x+10
  ));
```

9.3.1.2 Putting Outside Constant Values In Every Output Tuple From A Display

Sometimes, it is desirable to put the value of an outside variable in the each TUPLE for a Display call. Of course, this is probably a rare occasion since the corresponding column will have the same value for every row and thereby be quite redundant. Nonetheless, it can be easily accomplished by introducing a special ‘output’ variable as illustrated by vpath.1.Q:

```plaintext
local: \text{STR(25)}.color

set \[ .color \] = read(\ from _\text{cmd\_line}_ but_if_absent \[ "peach" \ });

with_format _table_
do Display each \[ .name, .out\_color, .weight \] each_time(
  \[ .name, .weight \] Is_The_First_Where(
    there_is\_a PART\_ using siz where(
      Name = .name
      and Color = .color
      and Weight = .weight )
  )
  and .out\_color = .color
);
```

9.3.2 OPCONDS in for_each_time loops

The OPCOND underlying the for_each_time loop
for_each_time SomeVblSpecs is_such_that BoundedAsn Do

is

\[ \sim \{ \text{SomeVblSpecs} \} \text{ such_that } BoundedAsn \]

The idea behind the for_each_time loop is that the Do portion is done each time a TUPLE is produced that satisfies the OPCOND. The scope of a for_each_time OPCOND’s variables is extended to include the Do.

The for_each_time variants:

\[
\begin{align*}
\text{for_each_time} & \quad \text{SomeVblSpecs} \quad \text{BoxFormerPred} \quad \text{BoundedAsn} \quad \text{HybridBoxKeywdArg} \\
& \quad \text{do } \text{BraceProg}
\end{align*}
\]

\[
\begin{align*}
\text{for_each_time} & \quad \text{SomeVblSpecs} \quad \text{Is_In} \quad \text{Aggregate} \quad \text{UseBoxKeywdArg} \\
& \quad \text{do } \text{BraceProg}
\end{align*}
\]

are considered to be abbreviations, respectively, for:

\[
\begin{align*}
\text{for_each_time} & \quad \text{SomeVblSpecs} \quad \text{is_such_that}( \\
& \quad \text{SomeVblSpecs} \quad \text{BoxFormerPred} \quad \text{BoundedAsn} \quad \text{HybridBoxKeywdArg} \\
& \quad \text{do } \text{BraceProg}
) \\
\end{align*}
\]

\[
\begin{align*}
\text{for_each_time} & \quad \text{SomeVblSpecs} \quad \text{is_such_that}( \\
& \quad \text{SomeVblSpecs} \quad \text{Is_In} \quad \text{Aggregate} \quad \text{UseBoxKeywdArg} \\
& \quad \text{do } \text{BraceProg}
) \\
\end{align*}
\]

9.3.3 Display Keyword Arguments And Their Functionality

The discussion on Display will now be concluded before proceeding on to present Cymbal’s procedural semantics for satisfying OPCONDs. Display offers many options to Daytona users as indicated by the keyword arguments listed in the Display import (i.e., prototype) given in the appendix $DS_DIR/sys.env.cy$. These keyword arguments fall into 5 groupings. First, foremost and already discussed are the keyword arguments like each and each_time that have to do with specifying Display’s required OPCOND.

The second group has to do with specifying various trappings that can be associated with the basic Display output. For example, the user can cause explanatory title lines to precede unfiltered or tabular output: if there is just one title line, then it can be the STR argument to the keyword with_title_line; 0 or more STR title lines can be collected into a TUPLE argument for the keyword with_title_lines (they will be printed as lines in the title in the order that they appear in the TUPLE). All heading/closing information can be suppressed by using the take-no-argument keywords with_no_heading / with_no_closing, respectively. with_no_heading will get rid of the Beginning to generate output message to stderr as well as the initial %msg) lines sent to stdout, which include the query file name, the title lines, and the column labels. with_no_closing gets rid of the End of output generation and ‘# answers generated’ messages sent to stderr.
Arbitrary STR labels for the columns of output may be given in the manifest TUPLE argument for the `with_col_labels` keyword; if not provided, column labels are generated from the names of the variables/terms given in the each TUPLE. By using manifest TUPLES of STRs as elements themselves within a `with_col_labels` TUPLE, the user can provide labels to Display each items that are COMPOSITE types like TUPLEs, STRUCTs, and fixed-length ARRAYs. Here is an example (display.2.Q):

```c
with_format _table_
    with_col_labels [ "Serial_Nbr", [ "Domain", "Range" ] ]
    do Display each[ .sn, .tu ]
    each_time(
        .tu Is_In [ [ .a, .b ]: .a Is_In [ 1 -> 4 ] and .b = .a*.a +1 ]
            with_selection_index_vbl sn
    );
```

Here is the output:

```
-------------------------
Serial_Nbr Domain  Range
-------------------------
  1     1     2
  2     2     5
  3     3    10
  4     4    17
-------------------------
```

This TUPLE nesting of Domain/Range within the `with_col_labels` TUPLE is mandatory: take the brackets away and the query will fail to compile.

This nesting of a TUPLE with the `with_col_labels` TUPLE can only be at most one level deep. Fortunately, this is without loss of generality because any Display element with a nested COMPOSITE type (as in a TUPLE of TUPLEs) is completely flattened to amount to be a single one-layer deep TUPLE for the purposes of Display.

As a convenience, if the Display each TUPLE contains an element COMPOSITE type which is the sole element of the each TUPLE and the `with_col_labels` TUPLE contains just a sequence of scalars, then Daytona will process the Display as if the `with_col_labels` TUPLE was a TUPLE containing the given TUPLE as its sole component, which is the correct form of the argument after all (and which would also be accepted if written by the user instead).

The third group has to do with formatting the body of the table itself. The `with_tuple_format` and `with_tuple_C_format` keywords are used to specify a C printf-style format for the output. See the discussion on `Write()` in Chapter 8 for more details, especially regarding the care with which these constructs must be used. The formatting of the entire table is determined by the argument to the keyword `with_format`. As discussed earlier in this chapter, the possible formats are `_table_`, `_packet_`, `_desc_`, `_xml_`, and `_data_`, the latter being the default. For the `_data_` format, the output separator and/or comment character can be specified by using the `with_output_sep` and/or `with_output_com_char` keyword arguments, respectively.

Copyright © 2013 AT&T
All Rights Reserved.
September 15, 2013
The fourth group of Display keywords have to do with the destination of the output. Display’s output will be sent to the created-as-needed file identified by the STR argument to the writing_to_file keyword; the output will be appended if the appending_to_file keyword is used instead. Finally, the output can be piped straight to a shell command by using piping_to_cmd. If an output CHAN already exists, then it can be used for the Display’s output by offering it as the argument to the writing_to_chan keyword. If none of these keywords are used, then the output goes to stdout.

The fifth group of keywords are taken from various box() constructs that are discussed in detail later in Chapter 12. They enable the user to sort, remove duplicates, and otherwise manipulate Display output in the same way that box contents are manipulated. Indeed, Daytona processes these box-analog keywords to Display by literally creating a box with the given keywords to store the Display output before presentation to the user. Here’s an example:

```plaintext
sorted_by_spec [ -2, 1 ]
with_candidate_index_vbl in_count
selecting_when( .in_count % 2 = 1 )
with_selection_index_vbl out_count
stopping_when( .out_count > 10 )
with_format _table_
do Display each[ .x, .y ]
each_time(
    .x Is_In [ 1 -> 20 ] and .y Is_In [ .x +4 -> .x -1 by -1 ]
);
```

The box material later in Chapter 12 will serve to explain that the above Display call selects for output every other answer generated by the OPCOND (starting with the first answer), stops generating answers after the 11th, and sorts the answers it generated first by the second component in decreasing order and then by the first component in increasing order. The output is:

```
----
  X  Y
----
 4  8
 3  7
 2  6
 4  6
 1  5
 3  5
 2  4
 1  3
 3  3
 2  2
 1  1
----
```
9.4 Procedural Semantics For Satisfying OPCONDs

There are of course no trustworthy oracles for producing the answers to queries. In order for Daytona to produce all TUPLEs of variable values which satisfy some OPCOND, it is necessary for the system to make the non-procedural procedural so that a computer program can compute all of the answer TUPLEs. This section enables users to understand better how OPCONDs work in queries by describing in general terms the algorithm Daytona uses to generate all TUPLEs which satisfy an OPCOND.

This algorithm is similar to the incremental-satisfaction, backtracking algorithm that Prolog uses. As it proceeds, the algorithm will treat certain assertions as finite generators of variable values whereas other assertions will be treated as tests. The purpose of the generators is to propose candidate TUPLEs for satisfying the OPCOND whereas the test assertions are used to determine whether candidate TUPLEs are to be certified as answers or else discarded.

Test assertions are easy to understand because they are used in a way similar to the way assertions are used in procedural languages: values are substituted for the free variable occurrences in the assertion and the truth value of the resulting instantiated assertion is computed. That truth value is used to determine whether the current candidate answer TUPLE passes or fails the test. Since the handling of test assertions is routine, it is the generating assertions that give logic programming (and declarative Cymbal) its unique properties.

Daytona’s algorithm for satisfying OPCONDs will be presented by describing how it processes each kind of OPCOND assertion supported by Cymbal. How to process an arbitrary OPCOND can then deduced by a process of structural recursion, i.e., in order to compute the TUPLEs satisfying a given top-level OPCOND, it will be sufficient to compute the TUPLEs satisfying certain implicit OPCONDs contained within the top-level OPCOND and then to combine those TUPLEs in a specified fashion so as to generate the answers to the top-level OPCOND.

In each case that follows, any outside variables present in the OPCOND are assumed to have values and hence, without loss of generality, they could just as well have had all of their occurrences replaced by their constant values. If \( T \) is a term, then the result of fully evaluating all variable dereferences, function calls, and other terms is written as \( T' \). For example, if \( T \) is \((.y-4)\) and the value of \( y \) is 10, then \( T' \) is 6.

The Daytona OPCOND satisfaction algorithm produces a satisfaction LIST for an OPCOND which is the LIST of TUPLEs of values for the OPCOND variables that satisfy the OPCOND’s assertion (or matrix).

9.4.1 Finitely Defining Variables

Daytona can only produce all TUPLEs that satisfy an assertion if there are but a finite number of such TUPLEs. In particular, Daytona must be able to determine a finite set of values for each declarative variable to take. (A declarative variable is one which is scoped within an assertion. A procedural variable is one which is scoped within an fpp.)

A defining (or generating) occurrence for a declarative variable occurs in a satisfaction claim that is capable of generating at most a finite set of values for it and is the lexically-first, non-scoping
appearance of this variable in the assertion with the proviso that if there is a defining occurrence for the variable in one disjunct, there must another one in each of the other disjuncts of the same disjunction. (Subsequent constraints on the variable may rule out some of these values when determining the set of values that the variable can take over its entire scope.) An assertion in Cymbal is well-formed if all of the variables scoped within are finitely defined on first non-scoping use, where for disjunctions, this stricture holds for each disjunct.

Also, by way of supporting terminology, for well-formed assertions, an occurrence of a variable that is not a defining occurrence and that is not in a quantifier (like there_exists .x) is said to be ground. Furthermore, all occurrences of procedural variables in an assertion are considered to be ground as well since a procedural variable always has a fixed constant value relative to any assertion it appears in. Terms which contain only ground occurrences of variables are also said to be ground; ergo, terms constructed solely from constants are ground. The term ground comes from the idea of being grounded, tied-down or fully instantiated in the sense that when the assertion processor comes to a ground term, that term can be evaluated to a constant since all of its variables have values at that point in the program execution. Furthermore, if all of the terms in a satisfaction claim are ground, then Daytona can evaluate the truth value of that satclaim. Consequently, an assertion is ground if all of its satclaims are ground. Furthermore, closed assertions (i.e., those with no free variable occurrences) are considered ground because all that can be done with them is to determine their truth value: they have no free variables to generate values for.\(^1\)

Although not all of the relevant constructs have been defined at this point in the manual, for completeness’ sake, what follows is the 12 (currently) supported ways to use a satisfaction claim to define a finite set of values for a declarative variable x. It’s understood here that any equality here can be rewritten with the LHS and RHS reversed and they are still generators in exactly the same way as they were before.

1. assert that \( .x = \text{term} \) where the term is ground. Note that this includes the cases where the term is a TUPLE or STRUCT or an intensional or extensional BOX or ARRAY(dynara) -- or elements thereof.

2. assert that \([ \ldots, .x, \ldots \)]\) is equal to a TUPLE where the element of the TUPLE corresponding to \(.x\) is ground.

3. assert that \([ \ldots, .x, \ldots \)]\) (or alternatively, \(.x\)) is equal to a LIST-valued function call (like one for tokens, aggregates, or read) where the RHS (right-hand-side) function call is ground. In the case of tokens, there may be more than one way to satisfy the equality, if the object being tokenized is able to produce a sufficiently long sequence of tokens that can be batched up to produce multiple LISTs of tokens. Likewise for read.

---

\(^1\) This definition of ground is different from the one in logic programming which says instead that a term is ground iff it has no variable occurrences. Also, Cymbal’s definition is just a (not bad) approximation of the real definition which involves model-theoretic interpretations and valuation functions and thus lives in a context beyond the scope of this document.
4. assert that \( .x \) is in a BUNCH, TUPLE, intensional BOX, bounded (i.e., finite) INTERVAL, or BUNDLE, where the RHS is ground. (Likewise for Is_Something_Where, Is_The_Next_Where, Is_The_First_Where, Is_The_Last_Where, Is_Selected_By, and likewise for Is_In_Again for intensional BOXes.)

5. assert that \([ ..., .x, ... ]\) is in a BUNCH, TUPLE or intensional BOX of TUPLES where the RHS is ground. (Likewise for Is_Something_Where, Is_The_Next_Where, Is_The_First_Where, Is_The_Last_Where, Is_Selected_By, and likewise for Is_In_Again for intensional BOXes of TUPLES.)

6. assert that \( .x \) is equal to a database FIELD value or is an element of a LIST/SET-valued database FIELD value. (This can only be done within a Cymbal description.)

7. assert that \( .x \) is (one of) the arguments to a macro PRED where that occurrence of \( x \) is a defining one.

8. assert that \( .x \) is (one of) the arguments to a view where that occurrence of \( x \) is a defining one.

9. assert that \( .x \) is in a path PRED with a ground term on the LHS as illustrated by:

   \[ .x \text{ Is A Royal Descendant Of } "William I" \]

10. asserting that a TUPLE of variable values satisfies an OPCOND, where any variables free in the OPCOND are ground. (This usage, while theoretically possible, almost never appears in practice.)

Furthermore, Daytona does not allow a variable to be defined directly or indirectly in terms of itself.

This concept of finitely-defining variables can be further clarified by this example:

\[ .x >= 2 \text{ and } .x <= 5 \]

Given that \( x \) is INTEGER-valued, there are clearly only a finite number of values that \( x \) can assume. However, Daytona cannot presently deduce this. It would be necessary to use instead:

\[ .x \text{ Is In } [ 2 -> 5 ] \]

In short, only the 12 cases listed above can be used to finitely define new declarative variables.

### 9.4.2 Satisfying Simple Satisfaction Claim OPCONDS

Each of the 12 different satisfaction claims for finitely defining the values of variables yields a case for defining the satisfaction of an OPCOND whose matrix is that satisfaction claim.
1. The simplest generating satisfaction claim (and the one upon which all others are based) is the scalar equality \( \sim[.x] \text{ such that } (.x = T) \) where \( T \) is a term whose variable dereferences, if any, are for outside variables. The corresponding satisfaction LIST is \([ [ T' ] ]\) where \( T' \) is the fully evaluated term. So, for example, if the outside variable \( z \) has value 4, then the satisfaction LIST for
\[ \sim[.x] \text{ such that } (.z * 4) = .x \]
is \([ [ 16 ] ]\).

2. LIST-equality OPCONDs like
\[ \sim[.x, .y, .z] \text{ such that } ( [ 1, .y, .a-1, .b ] = [ .x, 5, .z, .c ] ) \]
are processed by splitting the equality up in the natural way into a conjunction of scalar equalities and by processing that instead. In this case, the equivalent OPCOND would be:
\[ \sim[.x, .y, .z] \text{ such that } ( 1 = .x \text{ and } .y = 5 \text{ and } .a-1 = .z \text{ and } .b = .c ) \]
and the satisfaction LIST is \([ [ 1, 5, (.a-1) ] ]\) if \( .b = .c \) and is \([ ]\) otherwise. (How conjunctions are processed will be described shortly.)

3. LIST-equality OPCONDs like
\[ \sim[.x_1, \cdots, .x_k] \text{ such that } ( [ u_1, \cdots, u_n ] = \text{LIST–FUNCALL} ) \]
where each \( .x_i \) is some term \( u_i \) but if term \( u_i \) is not some \( .x_j \), then term \( u_i \) is ground. \( \text{LIST–FUNCALL} \) is an invocation of a FUN that returns a LIST, such as \( \text{aggregates} \) or \( \text{tokens} \). In the case of a FUN like \( \text{aggregates} \) the satisfaction LIST can be determined as in the preceding case (Case 2) once the \( \text{LIST–FUNCALL} \) has been materialized as a TUPLE, meaning that if the corresponding ground terms are not equal, then the satisfaction LIST is \([ ]\), whereas if they are all equal, then the the satisfaction LIST consists of the TUPLE of constants from the \( \text{LIST–FUNCALL} \) TUPLE that correspond to the \( .x_i \). In the case of \( \text{tokens} \) which is thought of as generating many TUPLEs, the satisfaction LIST consists of all TUPLEs of \( .x_i \) that correspond (as with \( \text{aggregates} \)) to \( \text{tokens–FUNCALL} \) TUPLEs when the corresponding ground terms are equal, with the generation ceasing when and if the corresponding ground terms become unequal.

4. The satisfaction LIST for a scalar \( \text{Is\_In} \) OPCOND like
\[ \sim[.x] \text{ such that } (.x \text{ Is\_In } T) \]
for TUPLE, BUNCH, bounded INTERVAL or intensional BOX \( T \) not containing any free occurrences of \( x \) is the LIST of bracketed elements of \( T' \). For example, the satisfaction LIST for
\[ \sim[.x] \text{ such that } (.x \text{ Is\_In } \{ 2, 3, .c, .d**2 \}) \]
is
(under the assumption that the values for the outside variables \( c \) and \( d \) are not such as to result
Actually, any scalar \texttt{Is\_In} OPCOND can be reduced to the more primitive form of a disjunction of equalities. (The processing of disjunctions will be discussed shortly.) For example, the OPCOND above is equivalent to:

\[
\sim [ \ .x \ ] \ \text{such\_that}( \ .x = 2 \ or \ .x = 3 \ or \ .x = .c \ or \ .x = .d**2 \ )
\]

(should all the proposed values for \( x \) be distinct). Since all of the elements of a bounded INTERVAL like \([ 1 \rightarrow .z \ by \ .q ]\) can in principle be enumerated explicitly, the same kind of equivalent OPCOND can be generated for the scalar \texttt{Is\_In} bounded INTERVAL case.

5. Consider the OPCOND

\[
\sim [ \ .x_1, \ \ldots, \ .x_k \ ] \ \text{such\_that}( \ [ \ u_1, \ \ldots, \ u_n \ ] \ \text{Is\_In} \ T )
\]

where each \( .x_j \) is some term \( u_i \) and \( T \) is a BUNCH, TUPLE, bounded INTERVAL or intensional box of \( n \)-TUPLEs whose definition contains no free occurrences of the \( x_j \). (Note that not every \( u_i \) need be a \( .x_j \).) Given that \( T' \) is a SET or LIST of \( q \) \( n \)-TUPLEs \( t_p \), the above OPCOND is considered equivalent to:

\[
\sim [ \ .x_1, \ \ldots, \ .x_k \ ] \ \text{such\_that}( \ [ \ u_1, \ \ldots, \ u_n \ ] = t_1 \ or \ \ldots \ or \ [ \ u_1, \ \ldots, \ u_n \ ] = t_q )
\]

which then reduces to other cases.

6. A dynamic associative ARRAY OPCOND like:

\[
\sim [ \ .x_1, \ \ldots, \ .x_k \ ] \ \text{such\_that}( \ .\text{ara}( \ u_1, \ \ldots, \ u_n \ ) = [ \ v_1, \ \ldots, \ v_m \ ] )
\]

where each \( .x_j \) is some term \( u_i \) or \( v_j \) but if term \( u_i \) or \( v_j \) is not some \( .x_j \), then that term is ground. The satisfaction LIST here is the LIST of all TUPLEs of values for the \( x_j \) such that there is some \( \text{ara} \) mapping of a domain TUPLE to a range TUPLE where the domain/range TUPLE elements that correspond to ground \( u_i \) or \( v_j \) are equal to their counterparts and the values for \( x_j \) are taken from their counterpart domain/range TUPLE elements. The scalar-valued ARRAY case is handled similarly.

7. Consider the following basic Cymbal description OPCOND, where the description is of a view or not:
\[ x_1, \ldots, x_k \] \text{ such that (}
\quad \text{there is a } \textit{REC\_CLS} \text{ where (}
\quad \quad A_1 = x_1
\quad \text{and } \cdots \text{ and}
\quad \quad A_k = x_k
\quad \text{)}
\text{)}

The understanding here is that a finite number of records \( R_1, \ldots, R_q \) of class \textit{REC\_CLS} are stored on disk, each having values for each of the \( k \) attributes mentioned. The description \textit{OPCOND} above is considered to be equivalent to:

\[ x_1, \ldots, x_k \] \text{ such that (}
\quad [ x_1, \ldots, x_k ] = [ R_1.A_1, \ldots, R_1.A_k ]
\quad \text{or } \cdots \text{ or}
\quad [ x_1, \ldots, x_k ] = [ R_q.A_1, \ldots, R_q.A_k ]
\text{)}

where \( R_i.A_j \) is the value that record \( R_i \) has for attribute \( A_j \). (Notice that records from record class \textit{REC\_CLS} that happen to fail to have a value for one of the attributes \( A_j \) are omitted from consideration.)

Descriptions having more complicated syntax can be rewritten so as to employ simpler cases. For example,

\[ [ x, y, z ] \text{ such that (}
\quad \text{there is a } \textit{SUPPLIER} \text{ named } x \text{ where (}
\quad \quad \text{Annual\_Revenues} > 1000000.
\quad \quad \text{and one\_of\_the Locations} = y \text{ for which (} x \text{ Matches } ^{\backslash}[AB] \text{ )}
\quad \quad \text{and Industry\_Code} = z \text{ which } > 2000 \& < 3000
\quad \text{)}
\text{)}

is equivalent to:
\[ \sim [ .x, .y, .z ] \text{ such that ( there_exists [ .u, .w ] such that (}
\text{there_isa SUPPLIER where (}
\begin{align*}
\text{Name} &= .x \\
\text{Annual_Revenues} &= .u \\
\text{Locations} &= .w \\
\text{Industry_Code} &= .z
\end{align*}
) \\
\text{and} &\ .u > 1000000. \\
\text{and} &\ .x \text{ Is_In .w} \\
\text{and} &\ .x \text{ Matches "^[AB]"} \\
\text{and} &\ .z > 2000 \\
\text{and} &\ .z < 3000
\)\]

(How to handle OPCONDS whose assertions begin with existential quantifiers will be discussed shortly.)

8. Consider the macro PRED OPCOND

\[ \sim [ .x\text{\textsubscript{1}}, \ldots, x\text{\textsubscript{k}} ] \text{ such that ( Some\_Macro\_Pred } [ u\text{\textsubscript{1}}, \ldots, u\text{\textsubscript{n}} ] ) \]

where each \( x\text{\textsubscript{i}} \) is some term \( u\text{\textsubscript{i}} \). Since a macro PRED is implemented by essentially replacing the macro PRED call with a suitably instantiated Cymbal assertion containing the \( u\text{\textsubscript{i}} \), the meaning of this OPCOND devolves to whatever that expanded assertion means.

9. Consider the path PRED OPCOND

\[ \sim [ .x\text{\textsubscript{1}}, \ldots, x\text{\textsubscript{k}} ] \text{ such that ( [ u\text{\textsubscript{1}}, \ldots, u\text{\textsubscript{n}} ] Some\_Path\_Pred } [ t\text{\textsubscript{1}}, \ldots, t\text{\textsubscript{n}} ] ) \]

where each \( x\text{\textsubscript{j}} \) is some term \( u\text{\textsubscript{j}} \) and where the terms \( t\text{\textsubscript{i}} \) and the definition of \( Some\_Path\_Pred \) contain no free occurrences of the \( x\text{\textsubscript{j}} \). The definition of a path PRED given in Chapter 16 implies that Daytona is able to produce a LIST of \( n\)-TUPLEs that satisfy the predicate in relationship to \( [ t\text{\textsubscript{1}}, \ldots, t\text{\textsubscript{n}} ] \). Consequently, this case reduces to the TUPLE-Is\_In-LIST OPCOND case.

10. Consider an OPCOND using an ancillary VBL like:

\[ \sim [ .x\text{\textsubscript{1}}, \ldots, x\text{\textsubscript{k}} ] \text{ such that ( MyPred[ } u\text{\textsubscript{1}}, \ldots, u\text{\textsubscript{n}} ] \text{ with\_anc\_vbl myvbl }) \]

where each \( x\text{\textsubscript{j}} \) is some term \( u\text{\textsubscript{j}} \) and one of them is \( myvbl \) but if term \( u\text{\textsubscript{i}} \) is not some \( .x\text{\textsubscript{j}} \), then that term is ground. Then the satisfaction LIST consists of all TUPLEs of values for the \( .x\text{\textsubscript{j}} \) such that the \( MyPred \) assertion is true if those values are substituted in for their counterparts.

11. The last case, that of an OPCOND satisfaction claim, is only of theoretical interest:

\[ \sim [ .x\text{\textsubscript{1}}, \ldots, x\text{\textsubscript{k}} ] \text{ such that ( [ u\text{\textsubscript{1}}, \ldots, u\text{\textsubscript{n}} ] Some\_Opcond }) \]

where each \( x\text{\textsubscript{j}} \) is some term \( u\text{\textsubscript{j}} \) and where \( Some\_Opcond \) contains no free occurrences of the \( x\text{\textsubscript{j}} \). Since \( Some\_Opcond \) has a satisfaction LIST of \( n\)-TUPLEs, this case reduces to the TUPLE-Is\_In-LIST OPCOND case.
9.4.3 Satisfying Conjunction OPCONDs

Understanding how to satisfy conjunction OPCONDs is fundamental. Consider an OPCOND like

\[ \sim[\ .x_1, \ldots, \cdot x_k ] \text{ such that } (C_1 \text{ and } \cdots \text{ and } C_n) \]

where the \( C_i \) conjuncts are logic assertions which may or may not contain the \( x_j \) or outside variables. Daytona processes these conjuncts from left to right by deciding for each one in turn whether it should be a generator or a test; if neither is possible, then an error condition is raised. As it processes the conjuncts, the algorithm collects the generators into a sequence of generators and the tests into a conjunction of tests. The generators are then started up and begin to generate in a cross-product fashion candidate TUPLEs for satisfying the OPCOND. Each one that satisfies the conjunction of tests is a solution. (Actually, the above process is equivalent to the more sophisticated one used by Daytona.)

Since the notation used in what follows gets somewhat involved, consider the following example which illuminates the basic principles involved.

\[ \sim[\ .x, \ .y, \ .z ] \text{ such that } \]

\[ .x \text{ Is_In } [4, 3, 2, 1] \]

\[ .x \% 2 = 0 \]

\[ .y \text{ Is_In } \{1, 2, .c\} \]

\[ .z \text{ Is_In } [.y \rightarrow .x + .c] \]

\[ .x + .y > 3 \]

Suppose that the value of the outside variable \( c \) is 2. Then \( y \) will only be assuming the values 1 and 2. The algorithm will consider the first conjunct to be a generating conjunct for \( x \) whereby \( x \) will assume the values 4, 3, 2, 1. \( x \) is the only variable appearing in the second conjunct and even if that second conjunct could be considered a generator of \( x \) values (which it can’t in this case), it would be and is considered a test in this example since it follows a generator for \( x \). In this case, it effectively cuts down the possible \( x \) values to 4 and 2. The third conjunct is taken to be a generator for \( y \) and since it has no dependence on \( x \), the two generators together produce the cross product of \( [4, 2] \) and \( [1, 2] \) as possible values for \( [.x, .y] \). The fourth conjunct is taken to a generator for \( z \); notice how for each \( [.x, .y] \), \( z \) has a different bounded INTERVAL to take its values from. Finally, the fifth conjunct is taken to be a test. Consequently, the TUPLEs that will satisfy this OPCOND can be computed by first, creating 3 nested loops for \( x, y \) and \( z \), respectively, where each of the variables is varying over its range and then second, accepting as a solution, a TUPLE \([.x, .y, .z]\) if it causes the conjunction of the two test conjuncts to be true. Here are the TUPLEs (formatted by Display) that satisfy this example OPCOND:
Returning now to an algorithm for satisfying the general case

\[ \text{\sim} [ .x_1, \ldots, .x_k ] \text{ such that ( C}_1 \text{ and } \ldots \text{ and } C_n ] \]

at each step \( i \) from 1 to \( n \), let \( G_i \) be the set of variables \( x_j \) that have values being generated for them by the conjuncts up to that step, with \( G_0 \) being the empty set. (Obviously, \( G_i \) does not include any outside VBLs relative to the OPCOND.) The algorithm goes as follows:

For each step \( i \) from 1 to \( n \),

1. If \( C_i \) can generate values for all of the \( x_j \) that appear free in \( C_i \) but do not appear in \( G_{i-1} \), then consider \( C_i \) a generator for these free \( x_j \) and let \( G_i \) be \( G_{i-1} \) augmented with these \( x_j \). This is the recursive step since what is happening here in effect is that an OPCOND is being formed:

\[ \text{\sim} [ .x_{j_1}, \ldots, .x_{j_q} ] \text{ such that ( C } C_i \]

where the \( x_{j_q} \) are the free \( x_j \) variables in \( C_i \) that are not in \( G_{i-1} \). The algorithm requires the production of all the TUPLEs that satisfy this new OPCOND in order to construct all the TUPLEs that satisfy the original conjunction OPCOND. Notice that, should any of them appear in \( C_i \), the variables in \( G_{i-1} \) are outside variables in so far as this subordinate OPCOND for \( C_i \) is concerned.

2. If all of the \( x_j \) that appear free in \( C_i \) are in \( G_{i-1} \) or are outside VBLs, then \( C_i \) is considered to be a test whose truth value can be ascertained once values for the \( x_j \) in \( G_{i-1} \) are known.

3. If neither 1 nor 2 hold, then raise an error condition since the OPCOND cannot be processed.

After all of the conjuncts have been categorized in this fashion into either generators or tests with no error condition having arisen, then, without loss of generality, in order to keep the subscripts
manageable, assume that the $C_i$ are relabelled so that first $p$ of the $C_i$ are the generators and that they appear in the order that the algorithm found them in the first place. Let $O_i(G_{i-1})$ be the OPCOND associated with a generating $C_i$ where the mention of $G_{i-1}$ emphasizes that the outside variables, if any, for $O_i(G_{i-1})$ may include the variables in $G_{i-1}$; consequently, for each set of new values for the $x_j$ in $G_{i-1}$, there is (potentially) a new OPCOND in effect associated with $C_i$ once those new $G_{i-1}$ $x_j$ values have been substituted in. (Of course, $C_i$ may not contain any outside variables, let alone any of the $x_j$, but in the general case, it will contain some occurrences of $x_j$ from $G_{i-1}$.) Without loss of generality, relabel the variables $x_j$ as needed so that $G_{i-1} = G_{i-1}$, i.e., the variables whose values are being generated by the OPCOND $O_i(G_{i-1})$, consists of the variables $x_{j+1}, \ldots, x_j$.

Then the TUPLEs generated by the $p$ generators are:

$$[ [ \cdot x_1, \cdot \cdot \cdot, \cdot x_k ] : [ \cdot x_1, \cdot \cdot \cdot, \cdot x_j ] O_1(G_0)$$
and $\cdot \cdot \cdot$ and
$$[ \cdot x_{j+1}, \cdot \cdot \cdot, \cdot x_j ] O_i(G_{i-1})$$
and $\cdot \cdot \cdot$ and
$$[ \cdot x_{j+1}, \cdot \cdot \cdot, \cdot x_k ] O_p(G_{p-1})$$

Notice how if none of the $O_i$ depend on the variables in $G_{i-1}$, then the TUPLEs generated are those of the cross-product of the TUPLEs generated by the $p$ OPCONds individually. In the general case though, the TUPLEs generated for $C_i$ can depend on the TUPLEs generated for $C_j$ for each $j < i$. A moment’s reflection reveals that the TUPLEs to be generated for a conjunction can easily be generated by a nested loops algorithm with one loop for each conjunct. Notice also that the linear, left-to-right nature of the algorithm precludes any circular situations where in order to compute values for $x_j$, one needs to know values for $x_j$. Recursive situations like this are handled in some logic-based systems but Daytona is not one of them. However, Daytona’s path PRED feature handles many of those inherently recursive logic specifications that arise in practice.

The best way to avoid the Step 3 error situation is to write all OPCOND assertions in such a way as to:

**Finitely define variables on first use.**

This means that the first lexical appearance of a variable in its scope should be associated with a sub-assertion that can serve as a generator for that variable’s values. By following this rule, the user can guarantee that Daytona will be able to process the user’s implicit or explicit OPCONds.

While Daytona will silently permute the order of various test assertions so as to cause test failure as early as possible during candidate TUPLE generation, there are two kinds of conjunct permutations which it does not do. First, it does not permute conjuncts in such a way as to change the defining occurrence(s) for a variable -- except in the box-of-key-field-values situation. Secondly, it does not change the relative order of generating conjuncts for different variables. In the latter case, perhaps it should but it doesn’t currently. This ordering is one of the primary ways that the user can influence the efficiency of the OPCOND satisfaction process: some orderings of the generators for different
variables can be quite a bit faster than others.

These procedural semantics for conjunctions are consistent with Prolog-style backtracking semantics. Without going into too much detail, for Prolog, when a generating conjunct is encountered for a variable, a new loop over the possible values for that variable is nested within the loops associated with previous generators, if any. When tests are encountered which disallow the current bindings for the loop variables, then backtracking occurs which means that the current attempt at satisfaction is abandoned in favor of a new attempt. Suppose that just after a new value for the top-level loop variable has been generated, a test invalidates that value. Then, when the next value for that loop is attempted immediately, quite a bit of work that would have been associated with the lower-level loops for that top-level loop variable value will be jettisoned as a result: in this case, backtracking will prune quite a bit of work from the search. Daytona moves tests as close to their generators as possible to achieve just this kind of efficiency.

9.4.4 Satisfying Disjunction OPCONDS

Fortunately, satisfying disjunction OPCONDs is a simpler matter than satisfying conjunction OPCONDs. Consider:

\[ \sim[ .x_1, \ldots, .x_k ] \text{ such that } ( D_1 \text{ or } \cdots \text{ or } D_n ) \]

where the set of \( x_j \) is the common set of all free VBLs appearing in each of the \( D_i \) that does not include any outside VBLs. The satisfaction LIST for the disjunction OPCOND is defined to be the concatenation of the satisfaction LISTs for the OPCONDs

\[ \sim[ .x_1, \ldots, .x_k ] \text{ such that } ( D_i ) \]

assuming that those subordinate satisfaction LISTs exist. (In the literature, this would be called a bag-union since it is a union that preserves duplicates.) For example, the answer to:

```c
set .outy = 6;
with_format _table_
do Display tuples_satisfying \sim[ .x, .y ] such_that( ( .x = 1 and .y = 2 )
or ( .x = 3 and .y = 4 )
or ( .x = 5 and .y = .outy )
or ( .x = 1 and .y = 2 )
);```

is
Here is an example of a disjunction OPCOND that Daytona cannot process:

\[ \sim[ .x, .y ] \text{such that}(\]
\[ .x \text{ Is_In } \{ 1, 2 \} \]
\[ \text{or } .y \text{ Is_In } [ 3\rightarrow4 ] \]
\[ ) \]

In this case, neither of the subordinate OPCONDs are well-formed, e.g., \[ [ .x, .y ] \text{such that}(\]
\[ .x \text{ Is_In } \{ 1, 2 \} \]
\[ ) \]
does not have \( y \) free in in the matrix. In fact, were the user to try to process:

\[ \text{do Display each}[ .x, .y ] \]
\[ \text{each_time}(\]
\[ .x \text{ Is_In } \{ 1, 2 \} \]
\[ \text{or } .y \text{ Is_In } [ 3\rightarrow4 ] \]
\[ ) \]

the following error message would be printed:

\text{error: unable to work with disjunctions which generate code that assigns values to different outside free variables in different cases (i.e., disjuncts). Check your ors and if-then-elses.}

\text{the best-guess corresponding Cymbal line number = 3}

\text{The list of outside free variables fixed in one case:}
\[ x \]

\text{The list of outside free variables fixed in another case:}
\[ y \]

Since

\[ 2 \text{ Is_In } \{ 1, 2 \} \]
\[ \text{or } 5034 \text{ Is_In } [ 3\rightarrow4 ] \]

it should be clear that the reason valid disjunction OPCONDs are defined so as to rule out different disjuncts generating values for different variables is that otherwise, there is potentially an unreasonably large number of solutions: just find a solution for one disjunct and then any values for the other variables are permissible for the other disjuncts and the disjunction as a whole remains true. Not only
is the number of solutions potentially infinite but they are unsatisfactory because they aren’t really
meaningful for the unconstrained variables.

It can be helpful to think of each disjunct as defining a parallel world for generating values for
variables and consequently, the (complete) rule is that every parallel world must generate values for
precisely the same non-outside variables, understanding that any of those variables is scoped and used
outside of the parallel world. In other words, it is actually OK for a parallel world to generate values
for distinct variables that are only known within itself. So, this criterion is actually more general than
saying, as was done at the start of this section, that every disjunct must generate values for the same
non-outside free VBLs.

As a technical note, recall that Daytona silently ensures that every disjunct begins with a somehow
or existential quantifier. For disjunction OPCONDS, if those somehows catch any variables, then
Daytona prior to evaluating the OPCOND will move the associated existentially quantified variables up
over conjunctions and disjunctions to become OPCOND_SOMEHOW variables, if possible. This can
result in duplicate answers (see subsequent discussion).

9.4.5 Satisfying If-Then-Else OPCONDS

If-then-elses are handled in a manner reminiscent of disjunctions. Consider:

\(~[ x_1, \ldots, x_k ] \) such_that( if \( A \) then \( B \) else \( C \) )

where no \( x_j \) is free in \( A \) and where all of the \( x_j \) are free in each of \( B \) and \( C \). Recall that all outside
variables relative to the OPCOND are assumed to have values, which means that either they are
procedural or that they have been finitely defined on first use elsewhere. This means that \( A \) is a test,
not a generator, since all of its free variables, if any, already have values. Consequently, if \( A \) is true,
then the if-then-else OPCOND has the same satisfaction LIST as:

\(~[ x_1, \ldots, x_k ] \) such_that( \( B \) )

whereas if \( A \) is false, then it has the same satisfaction LIST as:

\(~[ x_1, \ldots, x_k ] \) such_that( \( C \) )

In other words, an if-then-else OPCOND has the same effect as:

\(~[ x_1, \ldots, x_k ] \) such_that(\( A \) and \( B \))

or ( \( A \) and \( C \) )

except of course that the truth value of \( A \) is only evaluated once by Daytona.

As with disjunctions, any variables that \( B \) generates values for that are used outside of the if-then-
else must also have generators in \( C \) and vice-versa.

9.4.6 Satisfying Existential OPCONDS

Since Daytona inserts a somehow around every OPCOND matrix if an existential quantifier is
not already present, there is ample opportunity for the system to need to process existential OPCONDS
like:
where the $x_j$ and $y_i$ are all distinct and $A$ contains free occurrences of each of the $x_j$ and $y_i$. Recall that the $y_i$ are OPCOND_SOMEWHERE VBLs. Daytona processes this OPCOND by first obtaining the satisfaction LIST of this OPCOND instead:

$$\sim[ .x_1, \ldots, .x_k, .y_1, \ldots, .y_n ] \ such\_that( A )$$

Then it obtains the satisfaction LIST for the original existential OPCOND by projecting the elements of the derived satisfaction LIST onto their first $k$ components.

For example, given that $z$ is not an outside variable, Daytona processes:

$$\sim[ .x ] \ such\_that( .x \ Is\_In\ [ 1->3 ] \ and \ .z \ Is\_In\ [ 1->3 ] \ and \ .x > .z-1 )$$

as

$$\sim[ .x ] \ such\_that( \ there\_exists .z \ such\_that( .x \ Is\_In\ [ 1->3 ] \ and \ .z \ Is\_In\ [ 1->3 ] \ and \ .x > .z-1 ) )$$

which in turn is processed by computing the satisfaction LIST of:

$$\sim[ .x, .z ] \ such\_that( .x \ Is\_In\ [ 1->3 ] \ and \ .z \ Is\_In\ [ 1->3 ] \ and \ .x > .z-1 )$$

which is:

$$[ [1, 1], [2, 1], [2, 2], [3, 1], [3, 2], [3, 3] ]$$

When these are projected onto the first component, the result is what Daytona considers to be the satisfaction LIST for the original existential OPCOND:

$$[ [1], [2], [2], [3], [3], [3] ]$$

Note the presence of duplicate answers. Thus it is seen that existential OPCONDS in Cymbal’s (logic) calculus correspond to projections in the relational algebra.

### 9.4.7 Duplicates In Satisfaction LISTS

While the Daytona OPCOND satisfaction algorithm guarantees that every distinct answer TUPLE will be found, it promises nothing about how many times it will be found. In other words, several OPCONDS that are logically equivalent may generate satisfaction LISTS that are the same modulo the presence of duplicate answer TUPLES. For example, the following OPCOND:

$$\sim[ .x ] \ such\_that( .x \ Is\_In\ [ 1->3 ] \ and \ there\_exists .z \ such\_that( .z \ Is\_In\ [ 1->3 ] \ and \ .x > .z-1 ) )$$

is logically equivalent to the one just considered that produced duplicate answers. Yet the result of processing:
with_format _table_
do Display tuples_satisfying
-[ .x ] such_that( .x Is_In [ 1->3 ] and
    there_exists .z such_that( .z Is_In [ 1->3 ] and .x > .z-1 ));

is

-  
  X
  1
  2
  3
  -

Note the absence of duplicate answers. The reason for this is that Daytona considers the quantification over z as a test: consequently, for each value of x generated, the truth value of the quantification is computed to rule on the acceptability of that value of x as an answer and each value of x is generated exactly once.

In the future, Daytona optimization may be enhanced by permuting generators and by moving existential quantifiers around: these operations preserve logical equivalence among the different OPCONDs so generated but they may cause duplicate answers to come and go. So, the import of all this to the user is that unless the user has taken specific steps to avoid them, do not be surprised if duplicates appear in the answers to queries. The removal of duplicates can always be ensured by taking such steps as moving existential quantifiers in to have smaller scope or by creating a box which explicitly removes duplicates.

On the other hand, if the number of duplicates answers that Daytona produces has some meaning to the user, then the user should guarantee the preservation of that information by making the duplicates unique. One way to accomplish this is to make the OPCOND SOMEHOW variables OPCOND variables as was done above with z being moved in with x. There are other techniques that can be used; please see a Daytona developer for help if the presence or absence of duplicate answers becomes an issue.

A few comments on duplicate answers in the context of SQL are given in Chapter 4.

(Also, please note that Daytona does not guarantee a particular order for the answers of a query unless the particular query constructs being used are explicitly said in this manual to produce answers in a particular order.)

9.4.8 Varieties Of Test Assertions

There are some assertions that can never be generators: either they must be able to generate a test or the OPCOND satisfaction algorithm fails. For example, any satisfaction claim whose predicate is not equality or Is_In cannot be a generator and hence must be testable in order to be of any use at all. Likewise, negated statements and if-thens must always serve as tests.

Finally, with the exception of OPCOND SOMEHOWs, quantifications are always treated as tests.
These tests are implemented as sudden-death searches: the system treats the matrix of an existential quantification as an OPCOND and starts generating values: if one can be found which makes the matrix true, then the search stops immediately since the quantification itself must be true.

Universal quantifications are converted into existential quantifications using a standard logic identity. Nonetheless, it is a good idea to write universal quantifications so that the first or antecedent *Assertion* serves as a generating assertion that produces variable values for testing the truth of the second or consequent *Assertion*.

As a technical note, Daytona does try to rescue non-OPCOND SOMEHOW existential quantifications that contain generating occurrences for some free variables: it does this by moving the existentially quantified variables up to be OPCOND SOMEHOW variables, if there are only conjunctions and disjunctions between the two quantifiers. This may result in more duplicate answers (disju.1.Q). Perhaps in the future, other optimizations will be done instead, such as permuting conjuncts so that existential quantifiers can be moved in (instead of out) to their smallest scope.

### 9.4.8.1 Automatic Caching Of Ground Assertion Results

When Daytona compiles a query, it looks for opportunities to cache the results of processing assertions. This includes the truth values of ground *there exists, for each*, and *there isa* assertions as well as the values of declaratively defined BOXes, dynara, and aggregate function calls. Since the assertions involved in these constructs can contain outside variables, Daytona executables will recompute the cached value whenever the values for those outside variables change. Daytona executables will also recompute cached values when their process detects that a transaction (within the same process!) has run.

However, this caching optimization will not be deployed if Daytona discovers that a user-defined Cymbal function or predicate has been used in the assertion -- unless that user-defined fpp has been defined (or imported) with an *ignoring side effects* keyword. When caching is disabled for this reason, it is because Daytona is taking the conservative position that it cannot know whether or not the user has written potentially damaging side effects into the code of such fpps. Now, most likely, a side effect of just Writing strings to some CHAN is not going to change the assertion processing in a harmful way -- and so if that is desired and the caching is desired, then the fpp has to be defined/imported with the *ignoring side effects* keyword. Otherwise, if the fpp changes the values of variables that are outside of the assertion, then serious trouble (including, at the least, wrong answers) can ensue if the *ignoring side effects* keyword is used, whereas if it is not used, then Daytona will disable caching for the assertion -- as it should in this case. So, the user can get into serious trouble by using the *ignoring side effects* keyword when they should not be doing so.

Furthermore, any such user-defined PRED, whether using *ignoring side effects* or not, will be treated by Daytona like any other declarative PRED. Consequently, any satclaim using that PRED is subject to being moved around by Daytona’s optimizer, just like satclaims for other declarative PREDs are.

As another option for disabling caching, the *with_no_caching* keyword can be used when defining BOXes or dynara.

These concepts are illustrated in w_n_cach.1.Q as well as others.
9.4.9 A Final Procedural Semantics Example: Joins

This next example serves to review the definitely-define-on-first-use rule as it relates to conjunction OPCONDs and also illustrates how relational-type joins appear in Cymbal. In this query, for each supplier, the request seeks to identify every other supplier who supplied more of some part than the first one did during 1Q83 (compet.Q):

```plaintext
do Display
   with_title_line "Competitors with larger 1Q83 orders of a part"
   each[ .supplier, .competitor, .part, .qty1, .qty2 ]
   each_time(
      there_is_a SUPPLIER named .supplier where(
         Number = .supp_nbr1 )
      and there_is_an ORDER where(
         Supp_Nbr = .supp_nbr1
         and Part_Nbr = .part_nbr
         and Date_Placed >= '1-1-83' & < '4-1-83'
         and Quantity = .qty1 )
      and there_is_an ORDER where(
         Supp_Nbr = .supp_nbr2
         and Part_Nbr = .part_nbr
         and Date_Placed >= '1-1-83' & < '4-1-83'
         and Quantity = .qty2 )
      and .supp_nbr2 != .supp_nbr1
      and .qty2 > .qty1
      and there_is_a SUPPLIER named .competitor where(
         Number = .supp_nbr2 )
      and there_is_a PART named .part where(
         Number = .part_nbr
      )
   )
```

To understand this query, observe that the first conjunct establishes who the first supplier is and the second conjunct makes reference to one of its 1Q83 orders. The third conjunct identifies a 1Q83 order from another supplier of the same part (because a variable can have only one value at a time and .part_nbr appears in both ORDER descriptions). The 4th conjunct ensures that indeed the two suppliers are different and the 5th conjunct serves to restrict the orders of the second supplier to the ones with the greater quantities. The 6th and 7th conjuncts serve merely to get hold of the English names that correspond to the numbers of the part and second supplier.

Relative to the finitely-define-on-first-use requirement, the conjuncts of interest to us here are the
4th and 5th ones that assert \( \text{supp_nbr2} \neq \text{supp_nbr1} \) and \( \text{qty2} > \text{qty1} \). (By the way, description abbreviation conventions would allow both of these to be relocated inside of the second ORDER description.) The current implementation Daytona will exit with an error message if either one of these conjuncts is moved before the second ORDER description in the query. That is because, in either case, one of the variables has not been finitely defined on first use. For example, if the conjunct \( \text{qty2} > \text{qty1} \) were placed before the second ORDER description then \( \text{qty2} \) would not be finitely defined on first use since the first (non-quantifier) appearance of it in the query’s assertion is in the inequality satisfaction claim which does not satisfy any of the 10 cases in the definition. If this conjunct were moved before the first ORDER description, then both \( \text{qty1} \) and \( \text{qty2} \) would fail to be finitely defined on first use.

This query contains several examples of how relational joins are expressed in Cymbal. For example, there is a join between the first and second conjuncts, i.e., the first SUPPLIER conjunct and the first ORDER conjunct. This is indicated by the fact that the value of the Number FIELD of a SUPPLIER record is said to be equal to the value of the Supp_Nbr FIELD of an ORDER record by virtue of both FIELD values being said to be equal to the value of \( \text{supp_nbr1} \). This is equivalent to saying in SQL, \( \text{SUPPLIER.Number} = \text{ORDER.Supp_Nbr} \). Perhaps a closer parallel would be:

```csharp
there_is_a SUPPLIER named .supplier where(
    Number = .supp_nbr1 )

and there_is_an ORDER where(
    Supp_Nbr = .o_supp_nbr
    and Part_Nbr = .part_nbr
    and Date_Placed >= '1-1-83' & < '4-1-83'
    and Quantity = .qty1 )

and .supp_nbr1 = .o_supp_nbr
```

but the original expression is to be preferred because it is simpler (and besides, one of Daytona’s optimizations is to get rid of unnecessary variables and \( o_{\text{supp_nbr}} \) is surely one since it is always equal to \( \text{supp_nbr1} \)).

The joins here are called nested loop joins. Almost always for the sake of speed, nested loop joins should make use of indices, which would then make them indexed nested loop joins. Chapter 13 contains a discussion of indexed nested loop joins as well as a discussion of how to code the alternative hash joins in Cymbal. Definitely check this out because it contains the performance results of real world tests evaluating the various ways to use the two kinds of joins.

That concludes this section on procedural semantics for declarative Daytona queries. There is a lot of detail here but after a while, it can become second-nature and can make the logic part of Cymbal very pleasant to use because it is so concise and powerful.

### 9.5 The Nature Of Cymbal

Cymbal differs from many query languages in several dimensions. One such dimension has to do with Cymbal’s use of descriptions. These descriptions of records strongly encourage the user to group together in one place all of the assertions they are making about the fields in each given record. This contrasts with a language like SQL where it is possible to freely intermingle conditions on the
fields of different record types; in fact, for join conditions, it is even impossible to effect this kind of grouping because if one relation’s group gets the join condition (e.g., Y.ID = X.ID) then the other relation’s group does not. The advantage of this grouping is that it is easy to find out in a Cymbal query everything that is being said about each record type.

It is the independent existence of variables, as distinct from that of fields, that make grouping by description possible in Cymbal. This is because if it is desired to make reference to the value of a field outside of the description of its record then it is necessary to assert that a variable also has that value and to use that variable instead outside of the description. As it turns out, Cymbal’s use of variables constitutes another major dimension in which it differs from other query languages. Many query languages such as SQL avoid the use of independent variables by overloading field names with several different meanings in different contexts such as: (1) meaning the field itself, (2) meaning a variable which takes its values from the field’s values, and (3) meaning the value of such a variable. In Cymbal, there is no such overloading. For example, relative to the above query, Number is the analog of a field, part_nbr is the analog of a variable, and .part_nbr is the value of that variable.

One implication of this is that Cymbal queries may well be more verbose than queries in other languages since there is no point in associating a variable with a field if the system knows when to consider the field name as meaning either the field, a variable ranging over the field, or the value of that variable.

However, in addition to clarity, Cymbal gains in several ways by making these distinctions syntactically apparent:

1. Cymbal’s variables support bringing together in one place all that is said about each described record.

2. Variable names can be chosen so that they are suggestive of their roles in the particular query involved (cf., competitor above); this is of great value in writing understandable Cymbal for complicated queries.

3. If a variable’s value is to be output, it is the name of that variable that Display uses to label that value: since the meaning of the output record fields is not necessarily conveyed by the name of any of the queried records’ fields, the freedom of selecting variable names appropriately enables the user to select meaningful names for the columns in the new output table.

4. Also, since variable names are not tied down to be field names, Cymbal is free to introduce new variables that are not directly related to any database field, virtual or otherwise. (SQL, for example, allows new field variables only if they are the columns of new relations.)

5. No complications arise when the same name is used for fields in different records (like Number above).

6. The use of variables makes unnecessary any Cymbal analog for the relation name aliases required by languages like SQL as would be required here to reference the ORDER table twice.

7. Cymbal’s concept of variable is equally at home in the declarative and procedural setting. Of course, variables are absolutely essential in the procedural setting and as illustrated by parameterized queries, Cymbal variables provide an elegant method of communicating
information from procedural settings down into declarative settings.

8. Just from a syntactic standpoint, the dots make it very clear in Cymbal queries just where variables are being used and for that matter, they help the parser so much in distinguishing variables from keywords and functions that explicit variable definitions can almost always be avoided.

Cymbal differs from languages like SQL along yet a third dimension. This dimension is that of the size of the primitives. SQL is a language that has big primitives that are very powerful. There are essentially two primitives in SQL: the SELECT-FROM-WHERE statement and the GROUP-BY/HAVING modifiers used for aggregation. Every SQL query is built up by connecting together several of these big primitives. However, just because they are big, they are not as flexible as one would want and they can well lead to awkward and incomprehensible constructions for the more involved queries.

SQL is to Cymbal as a statistical package is to a programming language. The statistical package has powerful functions which take only a few arguments and produce, after much computation, sophisticated results. Of course, you can get the same results by writing a program; it’s just that it takes more keystrokes and more thought to do it. On the other hand, there are many useful and desired programs that you can write using the programming language that you will never get the statistical package’s big functions, applied in whatever combinations, to do for you. Cymbal’s smaller and more numerous primitives give it the flexibility and power of expression to easily write requests that could never be written in SQL. Cymbal was not designed to express average questions with the fewest number of characters; Cymbal was designed to express the most complicated of queries in as clear a fashion as possible.
10. Tokens

One of the primary tasks of a computer program is to read information in from its environment. When that information is presented as a sequence of bytes, *tokenizing* is the process of grouping the bytes into subsequences (called *tokens*), each of which can generate a typed object for the program. To this end, Cymbal offers the sophisticated and powerful *tokens* function.

*tokens* provides a single interface for tokenizing that:

- is able to handle input from any I/O CHANNEL or STRING
- supports all known tokenizing capabilities including regular expression, delimiter-based, message-based, and offset-based.
- automatically converts tokens to objects of specified types.
- offers both procedural and declarative tokenizing, the latter providing concise and powerful ways to tokenize well-behaved I/O CHAN. (Declarative tokenizing is fairly much unique to Daytona and constitutes the main value-added that *tokens* has over *read*.)
- offers full exception handling capability for procedural tokenizing.
- supports various missing value handling protocols for delimiter-based and message-based tokenizing.
- works with previously opened I/O CHAN or opens its own CHAN automatically, optionally using locking and/or user-specified buffer sizes.

It may come as no surprise then that *tokens* is a (non-trivial) synthesis of *new_channel* and *read*. This is clearly indicated by the keyword arguments that it inherits:
10.1 Procedural Tokenizing

At its simplest, `tokens` can be used to tokenize a string with a regular expression:

```plaintext
set [ .base, .ext ] = tokens( for "myprogram.c" matching "\([\^\.*]\)\.(\[^\]*\)" );
```

Since the default `via` is `_string_`, this `tokens` call in principle tokenizes a `CHAN(_string_)` containing the value of the `for` argument and produces a TUPLE as its result, which is then assigned to the manifest TUPLE of VBLS on the left-hand-side. (In practice, Daytona dispenses with the overhead of creating a `CHAN(_string_)` in this case and just applies the tokenizing to the `STR` `for` argument directly.) Unless otherwise specified, the variables that receive `tokens` values are assumed to be STRINGS.

`tokens` can also be used to assign values to TUPLES or conventional ARRAYS:

```plaintext
local: TUPLE[ FLT, DATE ] .tu
      INT ARRAY[ 4 ] .ara
set .tu = tokens( for "1234:3-31-98" upto ";" );
set .ara = tokens( for "1 2 3 4" upto " ");
```

Daytona implements these cases by using `new_channel()` to create `CHAN(_string_)` which it then uses `read()` to read. Notice also that `tokens` is handling all the type conversions (to INT and to DATE) automatically. It’s important to remember that `tokens` produces typed objects as tokens when asked to, not just strings. It is also important to remember that since `tokens` returns a LIST, it has to be assigned to a LIST, even if the LIST is a singleton:
local: STR .base
set [ .base ] = tokens( for "myprogram.c" matching "\([\^\.]\)*\).\([\^\.]\)*" );

Daytona will not accept the left-hand-side being .base alone.

10.2 Declarative Tokenizing

Procedural tokenizing is characterized by assigning the values returned by tokens to variables. True to the generate-or-test paradigm, declarative tokenizing is characterized either by asking the system to find all ways to tokenize given input, each one of which provides values for variables, or alternatively, just to test that given input can be tokenized in a specified way. In these situations, it is the equality satisfaction claim that is the Cymbal construct that invokes the functionality (whereas it is the assignment statement that invokes it for procedural tokenizing).

Here is a canonical example that uses an associative array to compute how many times each UNIX shell is the default login shell for the users of a system (tok.7.Q):

```
local: INT .shell_count[ STR : with_default @ => 0 ]
for_each_time .shell is_such_that(
    [ 6?, .shell ] = ftokens( for "/etc/passwd" upto ":\n" )
){
    set .shell_count[ .shell ]++;
}
```

(Associative arrays are discussed in Chapter 11 but understanding their use here should easy.) Here ftokens is the variant of tokens for which via \_file\_ in other words, the above also could be written using:

```
tokens( via _file_ for "/etc/passwd" upto ":\n" )
```

Conceptually, the /etc/passwd file is being thought of as an indefinitely long sequence of lines consisting of 7 tokens each. By using the satisfaction claim

```
[ 6?, .shell ] = ftokens( for "/etc/passwd" upto ":\n" )
```

with VBL shell free, the user is requesting that the system generate all possible ways to get 7 tokens, one such sequence after the other, so that the 7th token in each sequence can be given as the value for VBL shell. Obviously, this generation ends when the end of the file is encountered. Note the use of skolems ? to skip over tokens of no interest to the user in this example. Furthermore, any use of but_if_absent mirrors that of Read regarding skolem targets: the but_if_absent values are not used for skolems!

Compare this with the same expression in Perl:
open F, "/etc/passwd" or die ;
while(<F>) {
    chop ;
    split /:/ ;
    $sh_cnt{$7}++;
}

Perl is typically more terse in any event but the point is that the declarative Cymbal version merely names the file, the delimiters, and the pattern to be matched whereas the Perl (and corresponding procedural Cymbal) need to explicitly open the file, presumably close it at some point, and at least implicitly, use an assignment to pull out the 7th token and put it in a variable. It’s a difference in paradigm. In the context of Cymbal, when it is possible to use the declarative paradigm, the reward is that there are fewer characters to type than for the procedural paradigm and there is also an attendant simplicity of expression, which can be heightened all the more when additional declarative constructs are used in the same assertion.

The other half of the coin is that tokens satisfaction claims can also be tests:

```perl
set .x = 57;
when( [ .x, "abc" ] = tokens( for "57 abc" upto " " ) )
{
    do Write_Line( "Successful Test" );
}
```

Of course, hybrid tests where some portions of the TUPLE are ground and others free are also supported:

```perl
for_each_time INT .x is_such_that(
    [ .x, 8 ] = tokens( for "1 8 2 7 3 8" upto " " )
){
    do Write_Line( .x );
}
```

The output of this program is 1 and 3 on separate lines. The rule is that every TUPLE of tokens is considered in turn and if there are no _failed_comparison_ s, then whatever values are generated are taken as an answer to the satisfaction claim. (If INT is omitted above, then x will be inferred to have type STR(*) -- and the output will look the same.)

To further explore these ideas, consider this variant of the preceding query:

```perl
for_each_time [ INT .x, INT .y ] is_such_that(
    [ .x, .y ] = tokens( for "1 8 2 7 3 8" upto " " )
and [ .x, .y ] = tokens( for "1 8 2 7 3 8" upto " " )
){
    do Write_Words( .x, .y );
}
```

The output is:
The first conjunct serves to generate values for \( x \) and \( y \) and the second conjunct (rather pointlessly in this case) verifies that they are indeed tokens that can be found in that sequence of pairs of INTS. (This is a typical declarative testing paradigm used in Daytona’s test suite: conjoining two identical satisfaction claims should leave the query answer invariant over just having one of them.)

A generating tokens satisfaction claim stops generating if it runs into \_instant_eoc\_, \_missing_value\_, or other exception condition indicating an inability to tokenize the input. (As described later, the \_missing_value\_ exception cannot occur during declarative tokenizing except when \texttt{with~no~default~bia} has been explicitly mentioned.) Other than \_instant_eoc\_ and \_missing_value\_ (as well as \_failed_comparison\_ when doing a declarative test), exception conditions are considered unrecoverable and the program will abort with an error message. \_instant_eoc\_ and \_missing_value\_ will also (quietly) terminate a search for tokens when encountered when performing a test.

In order to control how many tokens are created during the processing of any declarative tokens satisfaction claim, the user may introduce a value for the \texttt{max\_times} argument. By default, this value is \_all\_ for declarative processing and it is trivially and necessarily 1 for procedural processing.

```plaintext
for_each_time [ INT .x, INT .y ] is_such_that(
    [ .x, .y ] = tokens( for "1 8 2 7 3 8" upto " "
    and [ .x, .y ] = tokens( for "1 8 2 7 3 8" upto " " max_times 2 )
)
    do Write_Words( .x, .y );
)
```

The output is:

```
1 8
2 7
```

10.3 Different Ways To Tokenize

Since tokens inherits its ability to tokenize from read, it necessarily supports all of read’s capabilities. These are the delimiter-based tokenizing provided by \texttt{upto} and \texttt{upthru}, line-constrained, regular expression-based tokenizing provided by \texttt{matching}, message-based tokenizing provided by \texttt{ending~with}, and offset-based tokenizing provided by \texttt{with~stated~sizes}. Just use the keywords as they would be understood by read. (This includes the "non\_-\texttt{string}\_\_CHAN, line-constrained" caveat for matching with read!)

10.4 Different Sources For Tokens

A tokens call can either get its tokens from a previously opened CHAN by means of read’s from argument or else, by means of a new\_channel call’s for and via arguments, it can (implicitly) ask new\_channel to open a new CHANNEL (and eventually Close to close it). Examples of the latter appear
above.

As a convenience, Cymbal offers stokens, ftokens, ptokens, ctokens to stand in for a tokens call where via is _string_, _file_, _pipe_, _cmd_line_, respectively. ftokens was illustrated above.

Here is an example of tokens using the CHAN associated with a TENDRIL(_spawn_) (See Chapter 19 for more about TENDRILS).

```cymbal
set .uten = new_tendril( with_downlink _receive_
    spawning `cat /etc/passwd`SHELLP );
set .line_count = count( over .ln each_time( 
    [.ln] = tokens( from .uten upto "\n" ) ));
set ? = next_waited_for_tendril( namely .uten ) otherwise do Exit(4 );
do Write_Line(.line_count);
```

(Read further to see a nicer way to do this.)

Now, while this illustrates the fact that tokens() can work off of a previously open CHAN, it does suggest how convenient it would be to have a ttokens() that would, behind the scenes, take care of opening the TENDRIL and waiting for it to end so that it can be closed. Happily, such a ttokens() exists:

```cymbal
LIST FUN: ttokens(
    /* from new_tendril */
    (0->1) for_bundle BUNDLE = _default_process_bundle_ ,
    (0) with_downlink _3GL_TEXT = _receive_,
    (0->1) executing DO,
    (0->1) spawning CMD|SHELLP,
    /* from read() */
    ( 0 ) from CHAN(?),
    ( 0->1 ) upto STR,
    ( 0->1 ) upthru STR,
    ( 0->1 ) matching STR|RE|CRE,
    ( 0->1 ) ending_with STR,
    ( 0->1 ) with_stated_sizes,
    ( 0->1 ) but_if_absent manifest TUPLE,
    ( 0->1 ) with_default_bia,
    ( 0->1 ) with_no_default_bia,
    ( 0->1 ) max_times INT
"
)
```

So, this ttokens() is the analog using new_tendril() and read() of ptokens() which is based on new_channel() and read(). Here is the equivalent ttokens() Cymbal code that accomplishes precisely what the much longer tokens()-based code does immediately above:

```cymbal
do Write_Line( count( over .ln each_time( 
    [.ln] = ttokens( spawning `cat /etc/passwd`SHELLP upto "\n" ) )));
```

Note that ttokens() also works with clones as well as spawn.
Here is a very useful example that shows how to implement a robust Perl-equivalent "file glob":

```perl
fet [.file] = ttokens( upto "\n"
    spawning
    (SHELLP)"for ii in .file_paths
do
    echo $ii
done"ISTR
) { ... }
```

Here `.file_paths` is shell regular expression like `/etc/*/*`. See tok.b.o for more examples.

10.5 Handling Missing Values For Delimiter-Based Tokens

Since `tokens` is based in part on `read`, it inherits the same missing value handling primitives. However, whereas for procedural `tokens` handling, the default is `with_no_default_bia`, for declarative `tokens` handling, the default is `with_default_bia`. The rationale is that since declarative `tokens` handling has no mechanism for handling exceptions like `_missing_value_`, appropriate default behavior is put in place to prevent the termination of tokenizing by the occurrence of missing values. Of course, the user can specify their own unique defaults by including a `but_if_absent` keyword argument. Consider:

```perl
for_each_time [ INT .x, DATE .y ] is_such_that(
    [.x, .y] = tokens( for "1234:4-14-01:::5678:6-14-01" upto ":" )
){
    do Write_Words( .x, .y );
}
```

The answers are:

```
1234 2001-04-14
0 9999-12-31
5678 2001-06-14
```

For contrast, consider a procedural alternative:

```perl
{ local: INT .x; DATE .y;
    set .schan = new_channel( via _string_ for "1234:4-14-01:::5678:6-14-01"
    otherwise do Exit(3);
    loop {
        set [.x, .y] = tokens( from .schan upto ":" ) otherwise break;
        do Write_Words( .x, .y );
    }
    do Close( .schan );
}
```

In this case, the answers illustrate that the missing values cause the `tokens` call to fail and the loop to terminate.
Of course, \texttt{with\_default\_bia} is not assumed for declarative tokenizing of \texttt{cmd\_line}, since then it would tokenize forever.

### 10.6 Procedural tokens Exception Handling

As illustrated in the preceding example, tokens used in a procedural setting sets a \texttt{tokens\_call\_status} variable which supports using an \texttt{otherwise} keyword argument. The following example shows how exception handling for tokens using \texttt{otherwise\_switch} can handle unruly input:

```plaintext
{ local: STR .x
  set .schan = new\_channel( via _string_ for "a:b::c:::d" )
  otherwise do Exit(3);
  loop { try\_for\_tokens:
    set [ .x ] = tokens( from .schan upto ":" )
    otherwise\_switch {
      case( = _missing\_value_ )
      {
        do Exclaim\_Words( "skipping a missing value!" );
        go try\_for\_tokens;
      }
      case( = _instant\_eoc_ )
      {
        break;
      }
      else {
        with\_msg "surprise error" do Exit( 6 );
      }
    }
    do Write\_Words( .x );
  }
  do Close( .schan );
}
```

Here is the answer:

```
a
b
skipping a missing value!
c
skipping a missing value!
```

The values for \texttt{tokens\_call\_status} are the same as for \texttt{read\_call\_status} because when tokens are being
processed, it is a read behind the scenes that is doing the work. Indeed, in the procedural setting, Daytona supports multiple tokens and read calls against the same open CHAN.

10.7 Miscellaneous Options To tokens

Last and least, tokens inherits these special-purpose keyword arguments from new_channel:

( 0->1 ) with_patience INT,
( 0->1 ) with_bufsize INT,
( 0->1 ) with_locking,

They can be of value when tokenizing via the appropriate CHANNELS.
11. Associative Arrays

A function maps each object in its domain set to a (unique) object in its range set. Functions are defined either algorithmically by giving the sequence of operations needed to compute the range object or extensionally just by identifying all the pairs of domain and associated range objects. In the latter case, the function is commonly known as an associative array or in some circles, as a finite map. Cymbal supports associative arrays that map TUPLES of scalars to scalars or alternatively, map TUPLES of scalars to TUPLES of scalars. One of the ways Daytona’s associative array implementation is particularly powerful is in its ability to map TUPLES to TUPLES: its competitors are often more comfortable mapping scalars to scalars.

The key point about the implementation of associative arrays is that a hash function is used to find the range object associated with any given domain object. This is a constant time lookup that is much faster at scale than using a Cymbal box for a similar purpose since that would take \( O(\log n) \) time due to the use of skip-lists. (Of course, in fairness to boxes, they also offer many services associative arrays cannot provide, including sorting, duplicate elimination, and the multiple, simultaneous indexing of the same collection of TUPLES.)

On the other hand, conventional arrays use offset arithmetic to compute which range object is associated with which INTEGER(S) from the domain lattice of the array -- or the Cartesian product of the INTEGER lattices making up the domain of the array. Consequently, conventional arrays are faster than associative arrays. But in favor of associative arrays, the domain of an associative array is much more general since it can be any collection of TUPLES of (writable) scalar objects. Consequently, associative arrays are perfect for representing functions that map a sparse (non-lattice) set of INTS to other objects. By definition, a writable scalar is a scalar that can be offered to the Write procedure call, which is effectively any scalar.

However, due to the vagaries of rounding, FLTS and types based on FLOATS like MONEY can be problematic when they appear in the domain TUPLE of an associative array, whereas of course, there is no problem if they are used in the range. The tricky situation here is this: suppose 8.33333333333333 is allowed to index an associative array element, and then the user evaluates several function calls resulting in computing the value 8.33333333333334, while presuming that it is 8 1/3, and then Daytona is asked to look up the element mapped from 8.33333333333334, whereupon sadly none will be found, thus leading to user surprise and disappointment. If one really needs to have a FLT index, then a good protocol is to apply round_to_nearest() to all FLT values before using them to index an associative array mapping or to compute an array element mapping as in the likes of .my_ara[ rtn(.myflt, .001) . Since the Daytona implementation converts all FLT-based indices to STRINGs, another option to just change the type to STR and use str_for_dec. (There is no penalty for doing this type conversion to STR because the Daytona implementation converts associative array indices to STRINGs anyway as needed.) See dynara.b.Q for examples of both approaches. See also the discussion on FLT in Chapter 5.

Cymbal supports two kinds of associative arrays: dynamic and static. The dynamic variant is the one that will be the most familiar to people and will be discussed first. For the static variant, the exact domain elements of the array are specified statically at compile time and cannot be changed.
subsequently, thus limiting the usefulness of this static variant.

11.1 Dynamic Associative Arrays

Cymbal’s dynamic arrays can have their extension modified during program execution, meaning that domain-range object pairs can be added and deleted -- and updated.

11.1.1 Defining/Declaring Dynamic Associative Arrays

Here are some examples of specifying a dynamic associative array type for a VBL:

```
export: DATE .birthday_for_person[ STR ]
import: IP .ip_for[ STR .zip, DATE_CLOCK .time ]
local:
    TUPLE[ INT .counts, FLT .totals, TIME .duration ] ARRAY[ INT, CLOCK ] .summary
    INT .z = 4
    FLT ARRAY[ INT ] .x = { 100 => 2.2, 1000 => 3.3, 10 <\* .z => .z + 0.4 }
    TUPLE[ INT, STR ] ARRAY[ STR, INT ] .y =
        ( [ "a", 1 ] => [ 26, "z" ], [ "b", 2 ] => [ "y", 25 ] )
define INT FUN( DATE ARRAY[ INT, STR ] .x ) date_calc {}
```

A number of concepts are illustrated here. The first is that a dynamic associative array (dynara) is defined/declared by using scalar class names to indicate where its domain elements are taken from. For example, the domain for the `birthday_for_person` dynamic associative array is `STRING`; the domain for `ip_for` is the cross-product of `STRING` with `DATE_CLOCK`. The range of `ip_for` is `IP`. Note that the range of the `summary` VBL is the space of `TUPLE[ INT, FLT, TIME ]`. Tag variables like `duration` are optional; their only utility is to serve as mnemonics for the user (except when using a dynara to dynamically cache expensive-to-compute function values as described later in this chapter). As is customary for ARRAY definition/declarations, the type can be factored out completely in front of the VBL name, as is the case for `summary`, or the definition/declaration can be written in the embedded C-style, as illustrated by the one for `ip_for`.

The binary infix `=>` operator is used to identify domain/range pairs (called map elements) for use in initializing dynamic associative arrays. For example, in the definition of `x` above, `x` is initialized with 3 of these map elements, including `10 <\* .z => .z + 0.4`. The latter indicates that function calls and references to previously defined variables are supported in map elements. A SET of map elements is a manifest dynara. (Recall that `<\*` is the infix exponentiation operator.)

Formally, definitions of (dynamic and static) associative (and conventional) ARRAYS are contained in the following grammar productions:
MultiVblDefs ::= [ vblad j | Type [ VBL | P | VblDBase | , VblDBase | ]* ]
;
VBL ::= VBL | VARIABLE
;
vbladj ::= static | dynamic | constant | alias | copy | C_external | manifest
;
VblDBase ::= [ . | P | lower | DimSpecTuple | P | | Initializer | P ]
;
Type ::= Type ARRAY DimSpecTuple
;
DimSpecTuple ::= [ DimSpec | , | DimSpec | P | | : | ArrayQualifiers | P ]
;
DimSpec ::= integer | Interval | [ SubjectSeq ] | Type
;
ArrayQualifiers ::= [ ArrayKeywdArg | P ]
;
ArrayKeywdArg ::= with_init_size_in_K Subject
| with_init_max_nbr_elts Subject
| with_growth_factor manifest INT|FLT = 2
| with_no_deletions
| with_deletions_ok
;
with_deletions_ok is the default for dynara.

11.1.1.1 Managing Huge Dynamic Associative Arrays In Minimal Space

(This special topic assumes some knowledge of declaratively defined dynamic associative arrays which are discussed at the end of this chapter.)

Since the process image of Daytona executables is constrained to 4GB (but only when working with 32-bit Daytona; not so for 64-bit Daytona), when a dynamic associative array becomes large enough, it can be necessary to use special constructs to minimize the effects of malloc allocation fragmentation so as to ensure that it will fit in the memory space available. To this end, all four of the keywords with_init_max_nbr_elts, with_init_size_in_K, with_growth_factor, and with_no_deletions can be utilized as in:
local: INT ARRAY[ INT : with_no_deletions with_init_size_in_K 1024000
   with_init_max_nbr_elts 30000000 with_growth_factor 1 ] .x
   // this assignment defines a dynara declaratively
set .x = {.a => .b :
   .a Is_In [ 1->2000000 ] and .b = .a+1 :
   with_no_deletions with_init_size_in_K 1024000
   with_init_max_nbr_elts 30000000 with_growth_factor 1 };
do Write_Line( .x.Malloc_Stats() );
do Write_Line( base_malloc_stats() );

The value for with_init_size_in_K is in kilobytes and represents the initial size of the space dedicated
exclusively to the array including its hash table and all elements it points to. As needed and if allowed
by the user, the array can be grown beyond this size but the point is that nothing else can use this space.
By far the greatest storage efficiency is achieved with with_init_size_in_K if with_no_deletions can be
specified. This would mean that array elements can only be added or updated but not deleted. If
with_growth_factor is 1.0, then the initial size is the final one; otherwise any with_growth_factor has no
effect since any growth in kilobytes will be done by the system in multiples of the with_init_size_in_K value. with_init_size_in_K cannot be used with shared memory arrays.

(Technical aside: with_init_size_in_K causes vmalloc to make a new and separate region; if
with_init_size_in_K is not used, then there is no creation of a new vmalloc region and the array will take
up space in the default region. However, in either case, once a dynara has been allocated, nothing
short of program termination (or using free on return when) will cause its hash table and other
infrastructure to be freed, even if it is a local non-static variable in an fpp that is called repeatedly.
However, the objects being pointed to by the dynara, i.e., the contents, will come and go as the
program leaves and returns to the scope of the variable (actually that happens just when the program
returns to the scope). And of course, those objects are freed when the user explicitly assigns {} to the
array -- but as just mentioned, the hash table infrastructure will still remain in that case.)

The value for with_init_max_nbr_elts causes the hash table to be initialized to be able to point to
elements numbering up to the least power of two greater than that keyword's value. Performance and
storage efficiency will degrade if the value for with_init_max_nbr_elts is exceeded and since there is little
penalty for a not-too-excessive over-estimate, a conservative value should be chosen. If
with_growth_factor is 1.0, then there will never be more than the initial number of slots in the dynara
hash table but the number of objects in the hash table can grow beyond that, resulting of course in
longer and longer collision chains but not failure to allocate. If not 1.0, then the with_growth_factor
value should be 2 or a multiple of 2.

For non-shared-memory dynara, both with_init_size_in_K and with_init_max_nbr_elts can be
specified. If so, then any with_growth_factor value is used for both. Also, with_growth_factor can be used
without either of with_init_size_in_K or with_init_max_nbr_elts.

For experts, the STR FUN .<array_vbl>.Malloc_Stats collects statistics on the memory region
holding the array whereas the STR FUN base_malloc_stats collects statistics on the (containing)
memory region for the entire program, i.e., the heap.

Copyright © 2013 AT&T
All Rights Reserved.
September 15, 2013
11.1.2 Creating/Updating/Deleting Elements Of Dynamic Associative Arrays

Cymbal assignment statements can be used to create/update/delete elements of dynamic associative arrays:

```c
local: FLT ARRAY[ STR ] .salary
set .salary[ "John" ] = 5000.0;
set .salary[ "John" ] += 1000.0;
set .salary[ "John" ] ++;
set .salary[ "John" ] = $ + $;
set .salary[ "John" ] = __absent__;
set .salary[ "John" ] = - ;
```

The latter two assignments, which delete the salary entry for "John" are equivalent since ~ is just an abbreviation for __absent__. Notice that it is not an error to remove the same domain element twice, or in other words to remove an element that is already not there in the array is OK. Also, the removal is being done by specifying the domain element alone, since that necessarily determines what the corresponding range element is.

The user can use the STRUCT member dereference .Elt_Count to obtain the number of elements in any dynara as in:

```c
do Write_Words( "number of array elts =", .salary.Elt_Count );
```

Since Daytona in general considers it to be a fatal error to use the value of an associative array for a domain element when such does not exist, some care must be taken when updating associative array elements. Consider, for example, one way to achieve the task of counting up the number of times various shells are used as the default shell by the logins in /etc/passwd:

```c
local: INT ARRAY[ STR ] .shell_count
for_each_time .shell is_such_that( 
   [ 6?, .shell ] = ftokens( for "/etc/passwd" upto ":\n" ) 
){
   when( .shell_count[ .shell ] = ? )
   { 
      set .shell_count[ .shell ]++; 
   } else {
      set .shell_count[ .shell ] = 1;
   }
}
```

The use of the skolem here is nothing out of the ordinary, since the rule for skolems says that the when clause above is equivalent to:

```c
when( there_exists .x such_that( .shell_count[ .shell ] = .x ) )
```

It is also equivalent to:
when(.shell_count[.shell] Is _present_)

This in fact constitutes one of the three exceptions to the rule that associative array elements cannot be
used if they don’t exist. The first exception is where the element appears as the LHS of an assignment
statement and the other two are when presence or absence is being tested for. So, with this
understanding, the following simpler do-group would not work in this setting:

```
set .shell_count[.shell]++;
```
because it is the same as:

```
set .shell_count[.shell] = .shell_count[.shell] + 1;
```
and .shell_count[.shell] on the RHS does not exist the first time a shell is assigned to VBL
shell. Using this do-group will result in a fatal run-time error. Some languages like Perl rather
boldly assume a default value for (some) expressions that have not be explicitly defined but Cymbal is
not one of them. If that value were 0, then that would be perfect here, were Cymbal like Perl.

### 11.1.2.1 @-defaults for dynara

However, Cymbal does provide the user the ability to explicitly define the default range value
for an associative array so that simpler updating becomes possible:

```
local: INT ARRAY[STR : with_default @ => 0 ] .shell_count
for_each_time .shell is_such_that(
  [ 6?, .shell ] = ftokens( for "/etc/passwd" upto ":\n" )
){
  set .shell_count[.shell]++;
}
do Write_Words("total number of shells =", .shell_count.Elt_Count);
```

By using the map element @ => 0 as the argument to the with_default keyword in defining the array,
the user is specifying a rule that the system create a default range value of 0 (in this case) whenever an
associative array element is referenced that doesn’t exist yet for given domain values. The @ token is
read as anything or everything which distinguishes it from ? which is read as something and from ~
which is read as nothing. So, if .shell_count[.x] does not exist when this expression is evaluated,
the @-default causes two things to happen: 1) the creation of a new dynara map element .x => 0 for
shell_count and 2) the evaluation of the term .shell_count[.x] to be 0.

This @-default value creation mechanism is operative in declarative settings (i.e., assertions) just as
it is in procedural ones. While this next example uses concepts to be defined later in this chapter, it is
included here in the @-default section for local completeness (dynara.demo.Q).
{ local:
    INT ARRAY[ INT : with_default @ => 4567 ] .xxx
    set .xxx[ 4 ] = 3;
    fet .y ist( .xxx[ 5 ] = .y ){ _Show_Exp_To(.y) }
    when( .xxx[ 6 ] = 8 ){  
        do Write_Line( "error: default = 8 !" );
    }
    when( .xxx[ 7 ] = 4567 ){ 
        do Write_Line( "default used" );
    }
    when( .xxx[ 8 ] = ? ){  
        else {
            do Write_Line( "default not used" );
        }
        when( .xxx[ 8 ] = ~ ){  
            do Write_Line( "default not used" );
        } else {} 
    when( xxx[ 8 ] = ? ){ 
        else { 
            do Write_Line( "default not used" );
        }
        when( xxx[ 8 ] = ~ ){  
            do Write_Line( "default not used" );
        } else {}
    }
}

The output is:

    .y = 4567
    default used
    default not used
    default not used
    default not used
    default not used
    default not used

It is very important that any @-default specification in the definition for a dynara be included in any import for that dynara because that is essential to everyone’s understanding of how the array behaves in use. Indeed the @-default specification is part of the type. All exports and imports must agree as to the type of shared objects as is also the case with arguments and parameters to fpps.

### 11.1.2.2 Deleting elements of dynara

To delete every element in a dynara in one easy step, simply set it to (), as in:
set .shell_count = {};

This does free the storage associated with the elements of the dynara while nonetheless leaving the smallish dynara infrastructure itself still allocated. See also free_on_return_when.

When it is necessary to loop over a dynara and delete a portion of it, care must be taken so as to avoid trying to modify the array in the body of the for_each_time loop while iterating over the array in the assertion for the for_each_time. This is because as a rule, data structures do not like being modified while they are being iterated over. Here is one way to accomplish this goal (dynara.demo.Q):

```q
local: INT ARRAY[ INT ] .ararand
   // this assignment defines a dynara declaratively (as discussed later)
   set .ararand = { .i => rand_int(1000) : .i Is_In [ 1 -> 1000 ] };
   fet .i ist( .i Is_The_Next_Where( .j = .ararand[ .i ] and .j > 444 ))
   { set .ararand[ .i ] = ∼; }
```

The reason why this works is because Is_The_Next_Where creates a box of keys specifying the elements to delete before any dynara elements are actually deleted by the for_each_time. See also the two-item list of suggestions for updating dynara at the end of this next subsection "Creating/Updating/Deleting TUPLE-based Elements".

### 11.1.2.3 Creating/Updating/Deleting TUPLE-based Elements

Special considerations come into play when updating TUPLE-valued dynamic associative arrays. As illustrated in the next example (dynara.k.Q), TUPLE-valued dynamic associative arrays are ideal for computing group-by queries:

```q
local: TUPLE[ INT .order_cnt, FLT .sum_qty, INT .max_days_open ]
       ARRAY[ INT(_short_) .supp_nbr ] .supp_stats
   for_each_time [ .supp_nbr, .qty, .days_open ]
      is_such_that( 
        there_is_an ORDER where( Supp_Nbr = .supp_nbr
        and Quantity = .qty and Date_Placed = .dp and Date_Recd = .dr )
        and .days_open = .dr - .dp
      ) 
      when( .supp_stats[ .supp_nbr ] = ? ) 
      { set .supp_stats[ .supp_nbr ] = [ $#1+1, $#2+.qty, max( .days_open, $#3 ) ]; 
      } else { 
        set .supp_stats[ .supp_nbr ] = [ 1, .qty, .days_open ]; 
      }
```

Copyright © 2013 AT&T
All Rights Reserved.
September 15, 2013
An ORDER record describes an order for a given part for a specified quantity from a specified supplier. This query makes one pass through the data using the (in-memory) associative array \( \text{supp\_stats} \) to accumulate summary information about each of the suppliers that it encounters, whatever those might be, since they are not in principle known in advance.

As discussed in Chapter 14, Daytona’s SET-based group-by syntax provides a more concise way to compute these statistics:

\[
\text{set } .\text{supp\_stats\_box} = \{ \\
[ .\text{supp\_nbr}, \text{count()}, \text{sum( over .\text{qty})}, \text{max( over .\text{days\_open})} ] : \\
\text{there\_is\_an ORDER where( Supp\_Nbr = .\text{supp\_nbr}} \\
\text{and Quantity = .\text{qty} and Date\_Placed = .\text{dp} and Date\_Recd = .\text{dr}} \\
\text{and .\text{days\_open} = .\text{dr} - .\text{dp} } \\
\}
\]

Not surprisingly, Daytona implements SET-based group-bys by using dynamic associative arrays. Nonetheless, there are group-by’s that can be computed directly using dynamic associative arrays that cannot be computed using the SET-based group-by syntax.

Anyway, the updating assignment statement for \( \text{supp\_stats} \) above could have been written in at least these three other ways:

// one way
set .\text{supp\_stats}[ .\text{supp\_nbr} ]#1 = .\text{supp\_stats}[ .\text{supp\_nbr} ]#1+1;
set .\text{supp\_stats}[ .\text{supp\_nbr} ]#2 = .\text{supp\_stats}[ .\text{supp\_nbr} ]#2+.\text{qty};
set .\text{supp\_stats}[ .\text{supp\_nbr} ]#3 = \text{max( .\text{supp\_stats}[ .\text{supp\_nbr} ]#3, .\text{days\_open})}.

// another somewhat better way
set .\text{supp\_stats}[ .\text{supp\_nbr} ]#1 += 1;
set .\text{supp\_stats}[ .\text{supp\_nbr} ]#2 += .\text{qty};
set .\text{supp\_stats}[ .\text{supp\_nbr} ]#3 = \text{max($#3, .\text{days\_open})}.

// an even better way but still not great
set [ .u, .v, .w ] = .\text{supp\_stats}[ .\text{supp\_nbr} ];
set .\text{supp\_stats}[ .\text{supp\_nbr} ]
\quad = [ .u+1, .v+.\text{qty}, \text{max( .\text{days\_open}, .w )} ];

These however turn out to be inefficient compared to the way ultimately chosen. Consider the first of the alternatives. Each of the assignments will result in one look-up of an array element to get the value to update and then another look-up to put the updated element back. This makes a total of 6 look-ups. Daytona implements the second alternative by using VBL VBLs to result in one look-up per assignment for a total of 3 look-ups. The third alternative way is better still since it looks up the array element once and stores all three TUPLE values in temporary variables, computes the updated quantities and then stores them using another look-up, for a total of two look-ups.

Incredibly though, the entire updating operation can be accomplished with just one array look-up! The use of $-assign in the single original assignment:

\[
\text{set } .\text{supp\_stats}[ .\text{supp\_nbr} ] = [ \$#1+1, \$#2+.\text{qty}, \text{max( .\text{days\_open}, \$#3 )} ];
\]

is translated by Tracy into Cymbal code equivalent to the following using a new VBL VBL named \( q \):
Note that single array look-up occurs when \( q \) gets what amounts to be a pointer to the range element that needs updating. Clearly, it is important to use the $-assign when updating TUPLE-valued dynamic associative arrays -- or else to use VBL VBLs directly if for some reason, the $-assign is not possible.

As illustrated by the next example, there are some interesting uses of skolems that come into play when using TUPLE-valued dynamic associative arrays.

```
set .supp_stats[ .supp_nbr ] = [ 42, 2? ];
set [ .u, ? ] = .supp_stats[ .supp_nbr ];
when( .supp_stats[ .supp_nbr ] = [ 42, 2? ] )
do Write_Line( "The change is made" );
when( .supp_stats[ ? ] = [ 42, 2? ] )
do Write_Line( "Got one" );
when( .supp_stats[ .supp_nbr ] = [ (3) ? ] )
do Write_Line( "Got one" );
when( .supp_stats[ .supp_nbr ] = ? )
do Write_Line( "Got one" );
```

Skolems are used in the dynara-range specification of a TUPLE assignment to indicate that the corresponding TUPLE elements should be ignored. Hence, in the assignment above, it is only the first component of the .supp_stats[ .supp_nbr ] TUPLE that is changed. The next assignment shows how to get just one element out of the range TUPLE of a map element. The other examples show how skolems can be used to test for the existence of map elements satisfying various conditions. The last two \texttt{when} examples are considered the same by Daytona.

Of particular note is the use of dynara to concisely and automatically support the caching of expensive-to-compute functions:

```
TUPLE[ DATE, DATE ] ARRAY[ INT .iii, DATE .ddd
    : with_default @ => [ ggg(.iii+2,.ddd,.zzz), ggg(.iii+4,.ddd,.zzz) ] ] .xx
```

Here \texttt{ggg} is expensive to compute, thus making the value of \texttt{xx} at some [ \texttt{INT, DATE} ] pair also expensive to compute. However, clearly, the default value mechanism will cause the value of \texttt{xx} at any particular [ \texttt{INT, DATE} ] pair to be computed just once -- automatically -- and thereafter, read from the cache that is the dynara. Note that it is the use of the tag variables on the dynara domain that enable the user to express how the corresponding range element is to be computed (the first time). (In the above example, \texttt{zzz} is a global VBL.)

Map elements can also be used in assignments to set and update the values of dynara:
local: INT .x[ INT ]
set .x = { 1 => 2, 3 + .z => .z };
// adds more elements
set .x += { 4 => 6, 8 => fff(9,.z) };

Some care must be taken when writing Cymbal that loops over elements of a dynara with the intent of updating them or adding/deleting elements. Cymbal is generally constrained by the rule that a for_each_time loop cannot modify the variables that are used in its assertion. This prevents various anomalies associated with modifying data structures while they are being read, not the least of which is being able to easily predict what the answer will be. The next example illustrates two ways to update all elements in a dynara without violating this rule:

```cymbal
for_each_time [ .supp_nbr, .order_cnt, .sum_qty, .max_days_open ]
is_such_that(
    [ .supp_nbr, .order_cnt, .sum_qty, .max_days_open ] Is_The_Next_Where(
        .supp_stats[ .supp_nbr ] = [ .order_cnt, .sum_qty, .max_days_open ]
    )
    set .supp_stats[ .supp_nbr ] = [ 10* .order_cnt, 10 * .sum_qty, 0 ];
)
for_each_time .elt
is_such_that( .elt = supp_stats[ .supp_nbr ] )
{
    set .elt = [ 10*$#1, 10*$#2, 0 ];
}
```

Actually, to truly understand this query, it is necessary to know how to work with dynara declaratively, which is the subject of the next section, and to know about boxes, which is the subject of Chapter 12. The above updates are allowed because they fall into the two exceptional categories:

1. The for_each_time assertion asserts that the for_each_time variables are in a box.

   Since this effectively caches the array (which can be space-intensive) before any changes are made, it is safe to make any changes because then the dynara will be modified during a time when it is not being read. This includes adding/deleting/updating elements.

2. Use a VBL VBL to point to the range elements of interest and then modify those range elements via the VBL VBL.

   This is safe because the structure of the dynara itself is not being modified while it is being read, rather just the contents of some of its storage locations. This strategy does not work for additions and deletions of map elements.

Dynara can be passed as arguments to fpps, either by alias or by value. They can also be imported and exported.
Dynara storage, as is typical, is freed up when the program execution leaves the scope of the variable.

11.1.3 Working With Dynamic Associative Arrays Declaratively

The reader may have noticed that the preceding examples did not contain any Cymbal code that professed to find out what was in a dynara, once created. This is because the only way to do that is declaratively. The declarative paradigm provides a very powerful way to retrieve information about what is in a dynara. Here is how to print out (some of) the answers from the group-by query above that computed the dynara .supp_stats (dynara.k.Q):

```plaintext
with_col_labels[ "Supplier", "Total Orders", "Avg Qty Ordered", "Max Days Open" ]
with_format_table_
in_lexico_order
do Display
each[ .supp_nbr, .order_cnt, str_for_dec(.sum_qty/.order_cnt,3), .max_days_open ]
each_time(
    .supp_stats[ .supp_nbr ] = [ .order_cnt, .sum_qty, .max_days_open ]
    and .supp_nbr % 10 = 5 // used to eliminate 90% of the output
);
```

True to the usual declarative semantics, the user is asking Daytona to produce the values for the free variables supp_nbr, order_cnt, sum_qty, and max_days_open that when substituted into the assertion, result in a true assertion. Note that the dynara is used here in an equality satisfaction claim that states the relationship between elements of its domain and range. Since dynara are implemented using hash tables that do not support any notion of ordering, there is no way for the user to predict the order in which the answer tuples will be generated: consequently, the in_lexico_order keyword is used to order the output for this query.

Also true to form, the use of ground terms in an assertion about a dynara serves to suitably restrict the output. Here is how to find the information about all suppliers with a total of 4 orders:
with_col_labels["Supplier", "Total Orders", "Avg Qty Ordered", "Max Days Open"]
with_format _table_
in_lexico_order
do Display
each[ .supp_nbr, .order_cnt, str_for_dec(.sum_qty/.order_cnt,3), .max_days_open ]
each_time(
    .supp_stats[ .supp_nbr ] = [ 4, .sum_qty, .max_days_open ]
    and .supp_nbr % 10 = 5  // just to cut down on the amount of output);

Any grounding pattern is fine in that any portions of the domain and range can be specified using
ground terms. However, remember that since dynaras are implemented as hash tables, that fast O(1)
"constant time" retrieval will only be possible if the entire domain of the dynara element reference is
ground; otherwise, Daytona has to loop through the entire dynara in order to apply the specified
ground constraints and generate what is asked for. Here's how to find out which of a select list of
suppliers has a total of 4 orders:

with_col_labels [ "Supplier", "Max Days Open" ]
with_format _table_
in_lexico_order
do Display
each[ .supp_nbr, .max_days_open ]
each_time(
    .supp_nbr Is_In [ ~415~INT(_short_), 425, 465, 485, 495 ]
    and .supp_stats[ .supp_nbr ] = [ 4, ?, .max_days_open ]
);

Note that the domain element is ground as is one of the range elements. (~415~INT(_short_) is
used above to get Tracy to infer the associated type for supp_nbr, which otherwise would be inferred as
a INT(_long_), leading to a type mismatch with the domain of supp_stats. This nuisance could have
been avoided entirely by declaring the Supp_Nbr FIELD in ORDER to be INT(_long_) but it was
declared INT(_short_) just for testing purposes.)

Skolems in the domain and range specifications can also be used in the usual way to indicate a
disinterest in something whose existence at least needs to be acknowledged but no more.

with_col_labels [ "Supplier", "Max Days Open" ]
with_format _table_
in_lexico_order
do Display
each[ .supp_nbr, .max_days_open ]
each_time(
    .supp_nbr Is_In [ ~415~INT(_short_), 425, 465, 485, 495 ]
    and .supp_stats[ .supp_nbr ] = [ 4, ?, .max_days_open ]
);

Notice how the previous .sum_qty has been replaced by ? because the user is just not interested in
that quantity and doesn’t want to bother coming up with a name for a dummy variable.

To just list all of the elements of the domain, execute:

```plaintext
with_col_labels [ "Supplier" ]
with_format _table_
in_lexico_order
do Display
each[ .supp_nbr ]
each_time( .supp_stats[ .supp_nbr ] = ? );
```

Note that ? can be used instead of the more accurate [ ?, ?, ? ] or [ 3? ]. An analogous construction can be used to loop over and produce all the range elements.

Here is a way to print out the elements of a dynara using a skolem and the fact that Write can write out a TUPLE:

```plaintext
for_each_time .supp_nbr
is_such_that( .supp_stats[ .supp_nbr ] = ? and .supp_nbr % 10 = 5 )
{
    do Write_Words( .supp_nbr, .supp_stats[ .supp_nbr ] );
}
```

If one wants to get fancy and only look up range elements once, VBL VBLS can be used:

```plaintext
for_each_time [.supp_nbr, .rng ]
is_such_that( .rng = supp_stats[ .supp_nbr ] and .supp_nbr % 10 = 5 )
{
    do Write_Words( .supp_nbr, ..rng );
}
```

Skolems can also be used in tests, as distinct from generating assertions:

```plaintext
when( .supp_stats[ ? ] = [ 4, 2? ] )
{
    do Write_Line( "Success #1" );
}
```

This tests to see if there is some supplier with a total of 4 orders.

Of course, absence (or presence) can also be explicitly tested for:

```plaintext
when( .supp_stats[ 315 ] = - )
{
    do Write_Line( "Success #2" );
}
```

Here is an unorthodox way to use a procedural VBL VBL to capture the output of an existence test so that with just one lookup attempt, if the value exists then it can be worked with, otherwise not (dynara.assgn.Q).
In contrast to the \texttt{fet} VBL VBL use above, this paradigm works with procedural VBL VBLs whose values are known outside of assertions. Since this violates the principle of the truth valuation of assertions not changing state, the same restrictions presented in Chapter 6 for the \texttt{set()} function are implicitly repeated here. A preceding explicit definition of \texttt{vv} is necessary to provide the type information needed by the Cymbal parser to handle the \texttt{set} call.

11.1.3.1 Failing, Not Aborting, When Absent

In general, the declarative portion of Cymbal is designed so that assertion failure, not program abort, occurs when a term turns out to be uncomputable at run time due to some necessary quantity failing to exist. The raison d'être for this policy is flexibility: the ability to test and/or otherwise react programmatically to assertion failure is clearly far more flexible than program termination.

Cymbal's behavior in this regard can be seen by running this:

\begin{verbatim}
set .box1 = [ 1, 2, 3 ];
when( 2 Is_In .box1 with_selection_index 241 ){ do Write_Line( "Success" ); }

fet .x ist( .x Is_In .box1 with_selection_index 241 ){ _Show_Exp_To(.x) }
when( [ "ABC" ] = stokens( for "adsf" matching "->\([a-z]+\)<"RE ) )
{ do Write_Line( "error" ); } // these stokens() calls have no return value

fet .x ist( [.x] = stokens( for "adsf" matching "->\([a-z]+\)<"RE ) )
{_Show_Exp_To(.x)}
\end{verbatim}

This program runs without error and produces no output! Regardless of whether referring to a box element or to a token for the purpose of doing a test or generating a value, the absence of such simply causes a satisfaction claim involving the term to evaluate to false in the former case and to cause backtracking in the latter. Likewise for handling missing FIELD values. For consistency and for the benefits of flexibility, the analogous situation is true for dynara. Consider what happens below when there is no value for dynara at index 400 (dynara.denote.1.Q):
local: INT ARRAY[ INT ] .dynara

when( 15 = .dynara[400] ){ do Write_Line( "Surprise!" ); }
for_each_time .x is_such_that( .x = .dynara[400] ){ do Write_Line( "Surprise!" ); }
set .y = .dynara[400];
// do Change so_that( .dynara[400] = 12 );

In the first example, the condition is false and so no Surprise! will be written; in the second, x does not get a value in the for_each_time loop and so no Surprise! will be written; in the third, the program aborts at runtime with an error message; if the fourth is uncommented, the program will not even compile because Daytona prefers that dynara assignments take place using the simpler syntax shown in the third case.

In summary, for ARRAY(_dynamic_,) .my_ara, if either .my_ara[.tu] or my_ara[.tu] (no initial dot) are evaluated when the dynara is not defined for .tu, then if this is in a testing situation, the enclosing satisfaction claim is considered false and if it is in a generating situation, then backtracking occurs.

Here is an example where this behavior is most useful. To compute the intersection of two sets as represented by .dynara1 and .dynara2 simply execute:

```c
for_each_time [.y] is_such_that( .dynara1[y] = true_ and .dynara2[y] = true_ ) {
    do Write_Line( .y );
}
```

The fact that .dynara2[y] may not exist for some .y does not cause the program to terminate, just to backtrack. The result is that the intersection of the two sets is printed.

Note that the presence or absence of a given map element can still be tested by using ? and ~ as in:

```c
when( .dynara[400] = ~ ) { do Write_Line( "yep, it’s gone." ); }
when( .dynara[400] = ? ) { do Write_Line( "surprise! it’s here." ); }
```

Also, it is important to note that the use of the @-default as in @ => 0 completely overrides this fail-no-abort behavior inasmuch as it causes a new map element to be created when one is found not to exist. Consequently, it is very important that any @-default specification in the definition for a dynara be included in any import for that dynara, so as to maintain consistent behavior.

Here’s an example that illustrates these concepts in the context of dynara VBL VBLs. Suppose two different survey teams have gone out and made estimates of the total population in different towns but that the surveys have not necessarily covered all the same towns. For the towns that have been surveyed by both, the following query prints out the population estimates of both and the average of the population estimates. Then it sets the sources of these estimates to have the average value instead.

(nul_dynara_objp.1.Q):
import: FLT ARRAY[ STR ] .popest1, .popest2
for_each_time [ .town, .est1_vbl, .est2_vbl ]
is_such_that(
    .est1_vbl = popest1[ .town ] and
    .est2_vbl = popest2[ .town ]
){
    do Write_Words( .town, ..est1_vbl, ..est2_vbl, (..est1_vbl+..est2_vbl)/2 );
    set ..est1_vbl = (..est1_vbl+..est2_vbl)/2;
    set ..est2_vbl = ..est1_vbl;
}

By using VBL VBLs (est1_vbl, est2_vbl), the query is able to restrict the number of hash table
lookups per common town to two, one for each of the arrays, which is clearly the minimum.

Here is another example where, by using a VBL VBL, the number of hash lookups can be
minimized to be exactly one. The goal in this example is to set a variable y to have a dynara value if it
exists but otherwise to generate a new value for y and store it. The straightforward way of doing this
would result in a maximum of two hash lookups:

import: INT ARRAY[ STR ] .int_id_for
when( .int_id_for[ .supplier ] = ? )
{
    set .y = .int_id_for[ .supplier ];
} else {
    set .y = new_int_id_for( .supplier );
    set .int_id_for[ .supplier ] = .y;
}

However, by using the principles and techniques of this section, the number of lookups can be reduced
to exactly one always:

fet .val_vbl ist( .val_vbl = int_id_for[ .supplier ] )
{
    set .y = ..val_vbl;
} else {
    set .y = new_int_id_for( .supplier );
    set .int_id_for[ .supplier ] = .y;
}

This idiom can also be adapted to work with boxes.

11.2 Declaratively-Defined Dynamic Associative Arrays

There are two ways to think of a dynara as being declaratively defined. The first is if the
dynara VBL is a declarative VBL, i.e., is a VBL defined in an assertion. Here are two examples of that
(decl.dynara.1.Q):

Copyright © 2013 AT&T
All Rights Reserved.
September 15, 2013
local: INT .z = 4
when( .x = { 1 => 2, 3 + .z => .z*.z } and .x[5] = 2 )
{ do Write_Line( "Success for x" ); } 

fet [ .x, .y ]

ist( .a = { 1 => 2, 3 + .z => .z } 
and .y = .a[.x] 
){
        do Write_Words( .x, "=>", .y );
}

The second is if the dynara VBL is defined by an assertion, which is to say by means of a "dynara-former" where the VBL’s value is said to equal the set of all map elements that make an assertion true. Schematically, that looks like this with two dynara-formers:

```
set .dara1 = { .x => .y : some assertion with free x and y };
for_each_time [ .x, .y ] is_such_that(
    .dara2 = { .a => .b : some assertion with free a and b }
    and .dara2[.x] = .y
){
    do Write_Words( .x, .y );
}
```

The first dynara VBL dara1 is a procedural VBL being defined declaratively; the second dynara VBL named dara2 is a declarative VBL being defined declaratively. Here are two specific examples of this (decl.dynara.1.Q):

```
fet [ .x, .y ] ist(
    .a = { .r => .s : .r Is_In [ 1 -> 10 ] and .s = .r*.r }
    and .y = .a[.x]
){
    do Write_Words( .x, "=>", .y );
}

do Write_Line( 25*"=" );
set .two = 2;
set .a2 = { [ .x, 1000 + .y ] => [ .x*.x, .two*.x*.y, .y*.y ] : 
            .x Is_In [ 1 -> 4 ] and .y Is_In [ 5 -> 9 ] };
fet .tu ist( .tu = .a2[?,?] )
{
    do Write_Words( .tu );
}
```

Note the use of expressions in the dynara-former for a2.

Clearly then, a dynara VBL can both be defined in an assertion as well as by an assertion. And dynara VBLS defined by an assertion may or may not be defined in an assertion. On the other hand, the first examples in this section show how a dynara VBL can be defined in an assertion but not by an assertion (decl.dynara.1.Q): note the use of a sequence of explicit map elements in the dynara
definitions. So, all possibilities are possible.

By the way, Daytona supports incrementally adding map elements to a dynara by adding declaratively-defined map elements (decl.dynara.1.Q):

```plaintext
set .a = { 1=>2, 3=>4 };  
set .a += { 10=>20, 30=>40 };  
set .a += { .x => .y : .x = 5 |=. 6 and .y = .x+10 };  
fet [ .x, .y ] ist( .a[.x] = .y )  
{ do Write_Words( .x, .y ); }
```

As will be discussed in complete detail in Section 12.1, the same homonym VBL scoping conventions used for boxes, Displays, and aggregate function calls are in force for dynara-formers. Here is an example (decl.dynara.1.Q):

```plaintext
fet [ .x, .y ]  
ist(  
 .sno = 404 and  
 .dya = [ [ .name, .sno ] => _true_ :  
     there_isa SUPPLIER where( Name = .name and Number = .sno ) ]  
)  
{ do Write_Words( .x, .y ); }
```

According to these rules, the \( \text{sno} \) VBL whose value is 404 is simply not the same as the \( \text{sno} \) VBL used in the dynara-former for VBL \( \text{dya} \), i.e., they are homonyms, different VBLs with the same name. Consequently, this query will print out dynara element values for every single SUPPLIER.

Alternatively, here is how to convey the first \( \text{sno} \) VBL’s values into the dynara-former, if such is desired:

```plaintext
fet [ .x, .y ]  
ist(  
 .sno = 404 and  
 .dya = [ [ .name, .sno2 ] => _true_ :  
     there_isa SUPPLIER where( Name = .name and Number = .sno2 which = .sno ) ]  
)  
{ do Write_Words( .x, .y ); }
```

The sole output of this is:

```
Webley Rentals 404
```
11.3 Static Associative Arrays

As illustrated by the following two definitions, a static associative array is characterized by an explicit listing of the domain elements in the definition, with the implication that the domain cannot be subsequently changed, either by additions or deletions.

```plaintext
local:
    STR .stara[ 2, [ "this", "that" ] ] = [ "a", "b", "c", "d" ]
       = [ [ 1, "a" ], [ 2, "b" ], [ 3, "c" ], [ 4, "d" ] ]

    set .stara[ 1, "this" ] = "e";
    set .stara[ 1, "where" ] = "e"; // runtime error!
```

Static associative arrays are largely vestigial at this point. Their primary distinguishing advantages are that they are faster (although probably not observably so) and that (static) associative dimensions can be used along with conventional array dimensions in the same array, as illustrated by `stara` above.
12. Boxes: More Than Just In-Memory Tables

One of Cymbal’s more powerful constructs is that of the box. A BOX is a finite collection of scalars or TUPLES of scalars, all elements being of the same type, that is stored totally within memory. These elements can be stored in such a way that duplicates are never present (i.e., as a SET) or in such a way that the stored items will be accessible in the order in which they were entered (i.e., as a LIST). In addition, multiple user-defined order specifications may be used to further define the order of access for elements of a BOX on different occasions. These order specifications are implemented using skip-list indices which provide $O(\log n)$ access to BOX elements. Finally, for intensional BOXES, the user may specify conditions for stopping the construction of a BOX and for determining on the basis of arrival order, what goes in the BOX. Since certain ways of using a BOX make use of what is essentially an index, BOXES are in fact in-memory tables.

Boxes can be defined either extensionally by explicitly listing their elements or intensionally by providing a membership assertion that characterizes what objects do or do not belong in the BOX. The extensionally defined BOXES are defined using BUNCHES, TUPLES or INTERVALS, although not every BUNCH, TUPLE, or INTERVAL is implemented using a BOX. Thus, it is seen that a BOX is an implementation data structure, just as a C-struct is an implementation data structure. Daytona reserves the right to implement syntax like:

```
[ 4, .x, .y+.z ]
```

in whatever is arguably the best way consistent with the user’s intent. Sometimes, the implementation is a BOX, other times, a C-struct, and still other times, a C-switch statement, or as an internal list that never really makes it to the C level as a single unit. So, in the case of TUPLE syntax, it is not a simple matter to predict how Daytona will implement it in C code. (However, every SET or type-homogeneous LIST VBL is implemented as a BOX.) Since the extensional forms were defined in Chapter 5, the rest of this chapter will focus overwhelmingly on intensionally defined BOXES.

The mandatory ingredient of every intensional BOX definition is an assertion with free variables: every element of the BOX will satisfy this assertion (but not necessarily conversely).

Here are two correct and semantically equivalent assertions that INT .x is a member of a SET BOX.

```
.x Is_In { .z : .z Is_In [ 4 -> 8 ] }
.x Is_Something_Where( .x Is_In [ 4 -> 8 ] )
```

(These examples are presented in a pedagogical, if not realistic, spirit since they are both equivalent to

```
.x Is_In [ 4 -> 8 ].)
```

The first is the standard mathematical way of saying these things which uses a so-called set-former and appears in text books as:

```
x \in \{ z : z \in [ 4 \to 8 ] \}
```

The second expression is a convenience that dispenses with the need for creating local variables like $z$. 

Copyright © 2013 AT&T
All Rights Reserved.
September 15, 2013
Here are a couple of semantically equivalent LIST boxes being used in assertions:

\[
[.x, .y] \text{ Is}_\text{In} \left[ [.z, .z \ast .z] : .z \text{ Is}_\text{In} \left[ 4 \rightarrow 8 \right] \right]
\]

\[
[.x, .y] \text{ Is}_\text{The}_\text{Next}_\text{Where} (.x \text{ Is}_\text{In} \left[ 4 \rightarrow 8 \right] \text{ and } .y = .x \ast .x)
\]

Note the use of brackets instead of curly braces in the first assertion to distinguish between SETS and LISTS. In this example, the LIST that is being created is a LIST of pairs of INTS with their squares.

The great thing about boxes is that their defining assertion can be absolutely any assertion as long as Daytona is able to figure out, in the general case, the finite collection of TUPLES of values for the free variables which make the assertion true.

Here is the basic syntax for defining and using boxes:
Cymbal syntax for boxes is fairly extensive but each syntactic form has its own area of convenient use. There are 3 primary ways to specify the construction of a box:

1. SET-formers / LIST-formers
2. box function calls
3. Box-former PREDICATES

Copyright © 2013 AT&T
All Rights Reserved.
September 15, 2013
Since all three of these forms reduce to the use of a box function call, the box function is a good concept to explain first. As the reader has probably surmised from the above discussion of box assertions with free variables, at the root of every box is an OPCOND. In fact, the one mandatory argument to the box function is an OPCOND, with the understanding that, were that to be the only argument, then the function would return the satisfaction LIST for the OPCOND. The optional BuildBoxKeywdArgs specify how to organize and/or trim down that satisfaction LIST in order to produce the box that is the result of the call. A box function call can produce sets, lists, bags/multisets, ordered sets, indexed sets, and many other structured collections of TUPLES that don’t even have commonly accepted names.

For example, with no_duplicates and with_default_arbitrary_order are two BuildBoxKeywds which together serve to create a box which is a SET as illustrated by these two equivalent box specifications:

\[
\{ \text{.color : there is a PART where( Color = .color )} \}
\]

\[
\text{box( for } \sim[ \text{.color } \text{such that( there is a PART where( Color = .color )} \text{with_no_duplicates with_default_arbitrary_order )})} \]

Clearly, it is more convenient to write the curly-brace set-former specification. But nonetheless, the basic and most general construct underlying and unifying all boxes is the box function.

12.1 Reducing Set-Formers/List-Formers To box Calls

Before describing in detail all of the BuildBoxKeywdArgs and their relation to the box function, the precise translation of a SetFormer/ListFormer into a box function call will be presented. Consider the following generic SetFormer:

\[
\{ [ t_1, \ldots, t_n ] : \text{Assertion : BuildBoxKeywdArgs } \}
\]

where the \( t_i \) are terms that may or may not include occurrences of the free variables of \( \text{Assertion} \) and may in fact include occurrences of variables defined outside the \( \text{SetFormer} \). Daytona considers this to be equivalent to the following box call:

\[
\text{box( for } \sim[ \text{.}v_1, \ldots, .v_n \text{ such that( }
\begin{align*}
&\text{Assertion} \\
&\text{and } .v_1 = t_1 \\
&\text{and } \ldots \\
&\text{and } .v_n = t_n \\
\text{with_no_duplicates} \\
\text{with_default_arbitrary_order} \\
\text{BuildBoxKeywdArgs } \end{align*}
\)
\]

where for each \( i \), \( v_i \) is a new, unique, system-generated variable, unless \( t_i \) is a simple variable dereference, in which case, \( .v_i \) is exactly \( t_i \). In other words, a new variable is introduced only if \( t_i \) is a constant, array element, tuple/structure member, or a function call; otherwise, the simple, original \( t_i \) variable itself is used as the OPCOND variable. Of course, in this latter case, the equality appended is of the form \( .x = .x \) for some \( x \) and consequently has no effect; furthermore, the use of this \( x \) as an OPCOND VBL will create a separate scope for this VBL name which will be distinct from that of any
homonym \( x \) appearing outside of the box-former.

Consider, for example, the following program (contained in box.demo.Q):

```plaintext
set .d = 4;
with_format _table_
do Display each[.x, .y, .z]
each_time( [.x, .y, .z] Is_In { [.w, .w * .d, 16] : .w Is_In [ .d -> 8 ] } )
```

Notice that the outside variable \( d \) appears both in the *Assertion* and in the target TUPLE \([ .w, .w * .d, 16 ]\). This program is equivalent to the program:

```plaintext
set .d = 4;
with_format _table_
do Display each[.x, .y, .z]
each_time( [.x, .y, .z] Is_In
  box( for \~[.w, .v1, .v2] such_that(
    .w Is_In [ .d -> 8 ]
    /* and .w = .w */ and .v1 = .w * .d and .v2 = 16 )
  with_no_duplicates with_default_arbitrary_order )
);```

The answer is:

```
X  Y  Z
---------
4  16 16
5  20 16
6  24 16
7  28 16
8  32 16
---------
```

When needed, the best way to specify the exact types of the elements of TUPLES in boxes is to cast as necessary the elements of the subject TUPLE in the box-former. Consider:

```plaintext
set .bbb = { [ (INT(_long_)).num, substr(.city,1,30) ] : there_is_a SUPPLIER where(
  Number = .num and
  City = .city and
  Telephone Matches "^602"RE ) };
```

Another alternative is to use a box() call and use the ability to specify types explicitly for OPCODES:
set .bbb = box( for ~[ FLT .nbr, STR .city ] such_that(  
    there_is_a SUPPLIER where(  
        Number = .nbr and  
        City = .city and  
        Telephone Matches "~602"RE )
    with_no_duplicates with_default_arbitrary_order );

The box function call corresponding to a ListFormer is produced in the same way as that for a SetFormer with the exception that the defining BuildBoxKeywordArgs are with_duplicates_ok and with_default_selection_order instead of their opposites, with_no_duplicates and with_default_arbitrary_order.

As a technical note, Daytona considers boxes of scalars such as:

\{ .color : there_is_a PART where( Color = .color ) \}

to be just convenient abbreviations of boxes of unary TUPLES such as:

\{ [ .color ] : there_is_a PART where( Color = .color ) \}

Thus, boxes really are always collections of TUPLES of scalars.

12.1.1 Technical Digression: An Undesirable Alternative

It would be simpler if this equivalence transformation always introduced a new variable to be equal in value to each term \( t_i \) regardless of whether it was a constant, variable value or function call. However, an unfortunate pitfall could arise if such a translation algorithm were employed as illustrated by:

local: INT .w
set .d = 4;

with_format _table_
do Display each[.x, .y]
each_time( [.x, .y ] Is_In { [.w, .u ] : .w Is_In [.d -> 8 ] and .u = 2*.w } )

This would transform to:

local: INT .w
set .d = 4;

with_format _table_
do Display each[.x, .y]
each_time( [.x, .y ] Is_In
    box( for ~[.v0, .v1] such_that(  
        .w Is_In [.d -> 8 ] and .u = 2*.w  
        and .v0 = .w and .v1 = .u )
    with_no_duplicates with_default_arbitrary_order )
    );

Now, the definition of \( w \) says that \( w \) is a procedural INT variable that has default value 0 and is therefore an outside variable in so far as the box assertion is concerned. Consequently, the query
would be processed as if it were:

```plaintext
local: INT .w
set .w = 0;
set .d = 4;
    with_format _table_
do Display each[.x, .y]
each_time( [.x, .y] Is_In
    box( for ~[.v0, .v1] such_that( // v0, v1 are new
0 Is_In [ .d -> 8 ] and .u = 2 * 0 // note: 0
and .v0 = 0 and .v1 = .u ) // note: .v0 = 0
    with_no_duplicates with_default_arbitrary_order )
);

Hence the introduction of an apparently innocent variable definition (i.e., for \( w \)) has completely
changed the definition of the box to something that is probably not desired. This unfortunate
communication of outside variable values inside occurs when what are intended to be different
variables from different portions of the query happen to share the same name (i.e., are homonyms); this
can be a very subtle problem to identify and resolve when the query is big. Note however that this
pitfall is removed if the OPCOND VBL for the box() call is left with the same name as its correspondent
had in the set-former expression -- in this case, Daytona will treat it as a new variable with a new scope
and will (internally) give it a new unique name, thus preventing the introduction of the outside
variable into the query. (Note that this is consistent with the way variables appearing in for_each_time
variable lists are treated.)

In the rare case where the user would actually like one of the \( t_i \) terms in the subject TUPLE for a
set-former assertion to be identical to an outside variable value (in which case that component of the
TUPLE will be constant for all members of the set), then the user can accomplish that by introducing a
new variable of their own as done with \( v0 \) below:

```plaintext
set .z = 8;
set .bb = { [ .i, .v0 ] : .i Is_In [ 2 -> .z ] and .v0 = .z };
```
or relying on the sensitivity of the translation to function calls, another alternative is the more elegant
but also more subtle and tricky:

```plaintext
set .z = 8;
set .bb = { [ .i, same(.z) ] : .i Is_In [ 2 -> .z ] };
```

### 12.1.2 Outside Variables And Box-Formers

Just as with Display calls, box-formers can be protected against unanticipated influence from
outside variables by explicitly quantifying any OPCOND_SOMEHOW variables:
set .bb = { [ .x, .y ] : 
  there_exists INT(_short_) .z such_that( 
    .x Is_In { 1, 2 } 
    and .y Is_In [ 3 -> 4 ] 
    and .z = (.x + .y) % 2 );

With the quantification of \( z \), the set-former is completely protected from the influence of outside variables.

### 12.2 Building Boxes

There are essentially two groups of box-associated keywords: **BuildBoxKeywds** and **UseBoxKeywds**. The former are used when defining a box so as to specify how to construct it, i.e., how to specify what properties it should have; the latter are used when using a box in order to specify how it is to be used this time: i.e., how will elements of an already constructed box be accessed. The term **Build Box** is used to denote the context of box construction; the term **Use Box** is used to denote the context of box use, i.e., where assertions are made about the elements of constructed boxes.

This section discusses the Build Box context by explaining in detail how to employ the various **BuildBoxKeywds** in the following syntax for box function calls and SetFormers/ListFormers:

\[
\begin{align*}
\text{Aggregate} & ::= \text{Bunch} \mid \text{Tuple} \\
& \quad \mid \text{SetFormer} \mid \text{ListFormer} \\
& \quad \mid \text{box}( \text{for} \ \text{OpCond} \ [ \text{BuildBoxKeywdArg} \ |^* ] ) \\
& \quad \mid . \ \text{BoxVbl} \\
& \quad \mid [ \text{SqlQuery} ]$
\end{align*}
\]

\[
\begin{align*}
\text{Tuple} & ::= [ [ \text{AggItemSeq} ] ] \\
\text{Bunch} & ::= ( [ \text{AggItemSeq} ] ] \\
\text{AggItemSeq} & ::= \text{AggItem} | \text{AggItem} [ \text{AggItem} ]^* \\
\text{AggItem} & ::= \text{Subject} | \text{IntervalSpec} | \text{Tuple} \\
\text{SetFormer} & ::= ( \text{SomeSubjects} : \text{Assertion} [ : \text{BoxQualifiers} ] ) \\
& \quad | ( \text{AggItemSeq} : : \text{BoxQualifiers} ) \\
\text{ListFormer} & ::= [ \text{SomeSubjects} : \text{Assertion} [ : \text{BoxQualifiers} ] ] \\
& \quad | [ \text{AggItemSeq} : : \text{BoxQualifiers} ] \\
\text{BoxQualifiers} & ::= [ \text{BuildBoxKeywdArg} ]^*
\end{align*}
\]
12.2.1 Build_Box Keywords

Duplicate TUPLES within a box may be forbidden by using the no-argument with_no_duplicates Build_Box keyword. The default is with_duplicates_ok which may be explicitly given or not.

There are many keywords that have to do with creating boxes that maintain simultaneously multiple sortings of the elements according to different sorting criteria.

Selection order corresponds to the order in which TUPLES were selected for inclusion in the box, i.e., the order in which they were generated and passed any additional tests (see, e.g., selecting_when below). Selection order becomes inaccurate and increasingly so however if elements are inserted after elements are deleted because the new elements go into the holes left by the deleted ones. The no-argument keyword with_default_selection_order may be used to specify that if in the use of a box, no order is explicitly asked for, then selection order is to be used. For LIST boxes, with_default_selection_order is implied. The default however for a box call is with_default_arbitrary_order, which means that the system is free to select an order of presentation which is the most efficient or convenient -- when the user does not explicitly specify one. For SET boxes, with_default_arbitrary_order is implied. The nature of these two keywords will be clarified later.

When the keyword with_lexico_order is used in a box definition, then Daytona will cause indexing structure to be created which will support accessing the TUPLES of the box in lexicographic order. Lexicographic order for Daytona TUPLES corresponds to sorting the TUPLES first on their first component, second on their second component, and so forth. This means that when two TUPLES have equal first components, their relative order is decided by considering the relative ordering of their second components and so on down to the last component if needed. The sorting done on a given dimension is done according to the natural <= ordering corresponding to the type of the values allowed to appear in that dimension. So, lexicographic ordering for TUPLES of type TUPLE[ INT, STR, DATE, IP ] corresponds to sorting the quadruples first on their INTEGER components according to the INTEGER <=, next on their STR components according to the STR <=, thirdly, on their DATE components according to the DATE <=, and lastly, (if needed) on their IP components according to the IP <=. For emphasis, recall "192.205.31.226"IP > "192.205.31.32"IP, which is not the way the obvious STR version of these quantities would be sorted by the STRING ordering. (Note that IP addresses are sorted lexicographically themselves by INT octet.) If with_reverse_lexico_order is used in a box definition, then index structures will be created which support accessing the box elements in reverse lexicographic order, which means that >= is used instead of <= for each dimension of the sort.

sort_specs enable the user to create index structures that sort the elements of a box according to any desired subsequence of the dimensions, either ascending or descending. For example, a [ 3, -2, 5 ] sorts the elements of the box first by ascending values of their third component, then by descending values of their second component, and lastly, by ascending values of their fifth component; values in the other components are ignored. Here is an example that illustrates specifying this kind of sorting in the construction of a box:
set .bbb =
    box( for ~[ .x, .y, .z ] such_that(
        .x Is_In [ 10 -> 1 by -1 ] and .y Is_In [ .x - 3 -> .x + 3 ]
        and .z = .x * .y )
    with_sort_spec [ -3, 2 ] );

If there is more than one sort specification, then the with_sort_specs keyword, whose argument is a
tUPLE of sort specs, must be used as in:

set .bbb =
    box( for ~[ .x, .y, .z ] such_that(
        .x Is_In [ 10 -> 1 by -1 ] and .y Is_In [ .x - 3 -> .x + 3 ]
        and .z = .x * .y )
    with_sort_specs [ [ -3, 2 ], [ 1 ], [ -2, 1 ] ]
    with_lexico_order
    );

This call causes Daytona to support 4 different sort orderings on the elements of the box in addition to
selection order, which is always available.

One of the advantages of boxes is that they grow automatically as needed. By default, they start
out able to hold 128 elements and when they expand, they grow by a factor of 1.5 at each reallocation.
In some situations, the user may wish to control how a box grows. In particular, if the final size of the
box is known, then the box can be created with that final size by using an appropriate argument for the
keyword with_init_max_nbr_elts. In this way, the box doesn’t need any memory reallocations while
being filled, which helps to conserve memory by increasing the efficiency of memory management, and
especially so if a final 1.5 growth increment would result in too much memory being taken. In fact, the
keyword with_growth_factor (by default, 1.5) will tell Daytona how fast to grow a box.

Now, unfortunately, for a box to grow more than, say 1.7 GB, Daytona cannot accomplish that by
this iterative reallocation process. The reason is that reallocation requires that the process image hold
both the original area and the new larger area (so that the former can be copied into the latter) and
after a point, they both just don’t fit into the maximum address space for 32-bit programs, which is 4
GB. So, the workaround is to use with_init_max_nbr_elts to allocate the final size of the box in the first
allocation. This can result in a box taking up close to 4GB in space, modulo the fact that other parts of
the program also need their own segments of the address space.

All supported Build_Box keywords are given in the prototype for box() in sys.env.cy.

12.2.2 Generalized Bunches And Tuples

BoxQualifiers can be added to BUNCHES and TUPLES so as to give indices to these
extensionally-defined boxes. Here is a BUNCH with two indices defined:

    { [ 1, 2 ], [ 2, 1 ], [ 3, 2 ],
      [ 2, 3 ], [ 3, 1 ], [ 1, 3 ]
    :: with_sort_specs[ [ 1 ], [ -2 ] ] }

The double colon may seem a little unusual until it is remembered that in general there are three
parts to a LIST/SET: the first is a listing of the elements, the second is some assertion, if any, they must
satisfy, and the third is any sequence of keyword arguments that further characterizes the LIST/SET.
In this case, there is no assertion to satisfy and so that is indicated by having two colons with nothing
inbetween. So, when defining the contents of a BOX, there can be at most 2 colons. On the other hand,
when specifying the type of a BOX, there can be at most one colon (since there cannot ever be any
assertion):


See box.demo.Q for more examples.

12.2.3 Box Ancillary Variables And Assertions

In general, an ancillary variable is a variable that the user supplies to an fpp call so that while the
fpp goes about its primary mission, it can assign an interesting value to the ancillary variable
concerning the course of the fpp’s computation for future reference by the user.

For example, associated with each element of a box are two integers: the candidate index, i.e., the
ordinal position of the element in the satisfaction LIST for the box’s OPCOND and the selection index,
i.e., the ordinal position of the element in the selection order of the box. These two ordinal positions
can only differ when the user uses a selecting_when, stopping_when, or backtracking_when ancillary
assertion to select which elements from the satisfaction LIST make it into the final box. In the following
pedagogical example contained in box.demo.Q, the box contains only the first 10 even integers taken
from [50->80]:

```
set .bb = box( for ~[ .w ] such_that( .w Is_In [ 50 -> 80 ]
   with_candidate_indices_stored
   with_selection_index_vbl si
   with_selection_indices_stored
   selecting_when( .w % 2 = 0 )
   stopping_when( .si = 10 )
 );
with_format _table_
do Display tuples_satisfying ~[.x,.y,.z]such_that(
   .x Is_In .bb
   with_candidate_index_vbl y
   with_selection_index_vbl z
 );
```

Here, in the box function call, the keyword-argument with_selection_index_vbl si serves to
cause the selection index for each TUPLE in the OPCOND satisfaction LIST to be assigned to the VBL
si so that it can be used in the stopping_when ancillary assertion. (Note that since the user is
identifying a VARIABLE of their own (not the value of some VBL) to be used by the system, the proper
argument syntax for with_selection_index_vbl is the likes of si, not .si.)

The purpose of the stopping_when ancillary assertion is to provide a condition for terminating the
generation of TUPLES in the OPCOND satisfaction LIST: when this condition becomes true, generation
stops. In general, ancillary assertions are fully general Cymbal assertions which are used during box
construction to modify either the search or the selection of elements for the box. They may contain free occurrences of the box’s OPCOND variables or of the box’s ancillary variables. The selecting_when assertion does just what the name suggests: if the assertion is true for a given candidate TUPLE, then it is selected for inclusion in the box. (The utility and power of selecting_when assertions is most apparent when they include free occurrences of ancillary variables; otherwise, they could just be conjoined to the OPCOND assertion.) (The backtracking_when assertion is only used for path predicates.)

Although of more use in the path PRED setting, the PRED Candidate_Selected_Before can be used in the stopping_when assertion to take appropriate action on the occasion that an element previously selected for the box appears again as a candidate.

The keyword with_candidate_indices_stored causes the candidate index for a given box element to be stored in such a way that in a corresponding Use_Box_Vbl situation (like that of the Display above), the value may be retrieved into a user-supplied candidate index variable by making use the with_candidate_index_vbl keyword. The with_selection_indices_stored keyword is handled similarly in a Use_Box_Vbl situation.

The answers to the preceding query are:

```
X  Y  Z
---
50 1  1
52 3  2
54 5  3
56 7  4
58 9  5
60 11 6
62 13 7
64 15 8
66 17 9
68 19 10
---
```

In defining a box, the with_sort_indices_stored keyword causes the sort indices for all sort specifications (with the exception of selection order) to be stored for future retrieval in Use_Box_Vbl situations by using with_sort_index_vbl; otherwise, they will not be available.

Ancillary variables are not allowed to appear in their box’s OPCOND assertion.

If there is no need to have the values cached in a box VBL for multiple purposes, then the following simple Display will do the same job.
12.2.4 Extended Set-Former/List-Former Syntax

Since it is inherently less verbose, more conventional and more aesthetic to use the SetFormer/ListFormer Syntax, this syntax has been extended so as to allow the use of \texttt{BuildBoxKeywdArgs}. Here is an example (from box.demo.Q):

\begin{verbatim}
set .box2 = [ [.a, .b, (.a *.a)+1] :
    .a Is_In [ 1->4 ] and .b Is_In [.a -1 -> .a + 2 ]
    : with_reverse_lexico_order
    with_sort_spec[ -2, 1 ] ];
\end{verbatim}

Notice that after the assertion, a colon introduces a sequence of \texttt{BuildBoxKeywdArgs}. (The syntax productions for SetFormers were given earlier.)

12.3 Box Variables

In several examples above, variables were defined whose values were boxes. In general, both assignments and value-generating uses of equality may be used to define box variables. For example:

\begin{verbatim}
set .inter = [ 1->4 ];
with_format _table_
do Display each [ .x, .y ] each_time(
  .box1 = [ [.a, .c+1] : .a Is_In .inter and .c = .a *.a ]
  and [ .x, .y ] Is_In .box1
);
\end{verbatim}

Daytona keeps track of how many elements are stored in a box. This INT is available at the Cymbal level by using the \texttt{Elt_Count} structure member as in: \texttt{.box1.Elt_Count}.

As of this time, box variables are not completely implemented. While boxes may be passed as parameters to fpps, they may not yet be returned as values from functions. Boxes cannot be used in equality tests. Unless the left-hand-side of an assignment is for an alias BOX VBL, it is not possible to assign the value of one BOX VBL to another.
12.3.1 Explicitly Typing Box Variables

Previously in this chapter, BOX VBLs were defined implicitly simply by saying that a BOX constant was equal to or was now assigned to be the value of a variable. The type of the BOX VBL was quietly inferred in the background. Here are some examples showing how to explicitly declare the type of a BOX VBL:

```plaintext
export:
  SET{ INT } .bb
  LIST[ TUPLE[ STR(25) .name, INT .age, FLT .salary ] ] .folks
  BOX[ TUPLE[ DATE, IP, HEKA ] : with_no_duplicates with_lexico_order ] .yy

import:
  LIST[ TUPLE[ STR, DATE, FLT ]:
    with_sort_spec[ 2 ] with_candidate_indices_stored
    with_deletions_ok ] .cc

local:
  static SET{ TUPLE[ INT(_short_), INT(_short_) ] } .lu_box
  SET( INT ) VBL dd
```

In short, a BOX type specifies either a SET or LIST of some scalar type or a SET or LIST of TUPLES of scalar types. The Build_Box keywords may be added to further define a BOX type. Note that variable names like `name`, `age`, and `salary` above may be added to give further meaning to a component of a BOX TUPLE. Here are some relevant productions from the Cymbal grammar:

```
Type ::= TUPLE [ ArgSlotDefSeq ]
| LIST [ Type [ BoxQualifiers ] ]
| SET { Type [ BoxQualifiers ] }

BoxQualifiers ::= : [ BuildBoxKeyIdArg ]
```

**Important:** When a BOX VBL is defined implicitly by assigning a constant BOX to be its value and when the same BOX VBL is explicitly defined as a local, imported, or exported variable, then the inferred type must be exactly the same as the type explicitly provided.

A BOX VBL may be used as a parameter variable (box.parm*Q). If the BOX it refers to will have elements added or deleted by the associated fpp, then it must be an alias VBL.

12.4 Using Boxes

This section describes `UseBoxKeyIdArg` that are used in `BoxAsn` in order to work with previously constructed boxes. Recall the relevant syntax:

```
BoxAsn ::= SomeSubjects Is_In Aggregate [ UseBoxKeyIdArg ]
```

Copyright © 2013 AT&T
All Rights Reserved.
September 15, 2013
12.4.1 Use_Box Keywords

The Use_Box keywords have to do with either

1. specifying the order in which the elements of the box are accessed (i.e., used) or
2. gaining access to candidate, selection or sort indices or
3. accessing a particular element of a box or
4. adding, deleting or updating elements of a box.

There are three different ways to access elements of a box: in selection order, in sort order, and in random order.

The keywords in_selection_order and in_reverse_selection_order are used to specify that selection order is to be used in the indicated manner to access the elements of a box. (Clearly, the last specification requires that the elements be accessed in the reverse order to which they were selected.) For example, this query from box.demo.Q:

```plaintext
with_format_table_

do Display each .x
each_time( .x Is_In [ 4, 2, 0, 1, 3, 5 ] in_reverse_selection_order );
```

produces the output:

```
- 
X
-
5
3
1
0
2
4
```

As this example makes clear, the UseBoxKeywds can be used for Is_In claims applied to BUNCHES, TUPLES and INTERVALS in addition to set/list-formers, box() function calls and dereferenced box VBLS.

The keywords in_lexico_order, in_reverse_lexico_order and sorted_by_spec are used to specify which of possibly several orderings are to be used for the given use of the box. As mentioned above, if no choice of sorting criterion is explicitly stated, then the default ordering for the box will be chosen (cf., with_default_selection_order). Here are three uses of a box defined above (box.demo.Q):
set .bbb =
  box( for \( x, y, z \) such that
    \( x \) Is_In \[ 10 \rightarrow 1 \) by -1 \] and \( y \) Is_In \( x - 3 \rightarrow x + 3 \)
    and \( z = x \times y \)
  with_sort_specs \[ [ -3, 2 ], [ 1 ], [ -2, 1 ] \]
  with_lexico_order
  with_selection_index_vbl si
  stopping_when( si > 10 )
);
with_format _table_
do Display tuples_satisfying \(~ x, y, z \) such that
  \[ x, y, z \] Is_In .bbb ; /* default here: arbitrary order */
with_format _table_
do Display tuples_satisfying \(~ x, y, z \) such that
  \[ x, y, z \] Is_In .bbb in_lexico_order);
with_format _table_
do Display tuples_satisfying \(~ x, y, z \) such that
  \[ x, y, z \] Is_In .bbb sorted_by_spec \[ -2, 1 \] );
with output:

<table>
<thead>
<tr>
<th>X_Scp2</th>
<th>Y_Scp2</th>
<th>Z_Scp2</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>7</td>
<td>70</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>80</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>90</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>110</td>
</tr>
<tr>
<td>10</td>
<td>12</td>
<td>120</td>
</tr>
<tr>
<td>10</td>
<td>13</td>
<td>130</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>54</td>
</tr>
<tr>
<td>9</td>
<td>7</td>
<td>63</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>72</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>81</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X_Scp3</th>
<th>Y_Scp3</th>
<th>Z_Scp3</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>6</td>
<td>54</td>
</tr>
<tr>
<td>9</td>
<td>7</td>
<td>63</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>72</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>81</td>
</tr>
<tr>
<td>10</td>
<td>7</td>
<td>70</td>
</tr>
</tbody>
</table>

Copyright © 2013 AT&T
All Rights Reserved.
September 15, 2013
The keyword `with_default_arbitrary_order` informs Daytona that the user does not care what order of presentation is used by default, i.e., when none is explicitly specified. This is different from `with_random_indices_stored` which is where the user instructs Daytona to construct a random permutation of the elements as the last step in constructing the box; this is like another "sort" order and therefore does not interfere with specifying and using selection order or some other orders based on sort specs. In order to access those elements using that random order, one uses `in_random_order` in a Use_Box situation. Here's an example:

```plaintext
set .bbb = [ .i : .i Is_In [ 1 -> 100 ] : with_random_indices_stored ];
fet .j Is_In .bbb in_random_order { _Show_Exp_To(.j) }
```

The implementation of random indices is incompatible with allowing deletions and so, deletions are not allowed for boxes with `random_indices_stored`. Additions are allowed; the true randomness of the permutation is not guaranteed subsequent to procedurally-specified element additions unless the Randomize() PROC for the box is called as in:

```plaintext
do .bbb.Randomize();
```

Transitive closure predicates can also be randomized in this fashion (since their elements are stored in a box). See box.rand.1.Q.

The keywords `with_candidate_index_vbl`, `with_selection_index_vbl` and `with_sort_index_vbl` are used to gain access to the corresponding attribute of the box element. In the previous section, an example was
given of a Display call which printed out the candidate_index and selection_index of box elements by using this mechanism.

Finally, by using the keywords with_selection_index and with_sort_index, particular elements in a box can be identified by their ordinal index according to a specified ordering. For example, for the box .bbb above, the following query:

```plaintext
with_format_table_
do Display each [.x,.y,.z]
each_time(
    [.x,.y,.z] Is_In .bbb with_selection_index 5
);

with_format_table_
do Display each [.x,.y,.z]
each_time(
    [.x,.y,.z] Is_In .bbb with_sort_index 5 sorted_by_spec [ -2, 1]
);
```

produces output:

```
-----------
X  Y  Z
-----------
 10 11  110
-----------
------
X  Y  Z
------
   9  9  81
------
```

The argument to both with_selection_index and with_sort_index cannot contain defining occurrences for any variables; in other words, these occasions cannot be used to generate values for variables: that in fact is the purpose of their cousins with_selection_index_vbl and with_sort_index_vbl. Either of these keywords may be given the argument _last_ which will cause the keyword-argument pair to refer to the last element in the box according to the designated sorting.

The as_quantile keyword, which takes an argument \( p > 0.0 \) and \( \leq 1.0 \), is used to access the \( p \)th quantile of the box according to the designated sorting criterion. For the statisticians in the audience, the \( p \)th quantile that Daytona computes is the value of left-continuous inverse of the sample cumulative distribution function at \( p \). At the lay level, the \( p \)th quantile is that point in the sample which is the closest (in some sense) to having \( 100*p% \) of the sample less than its value and \( 100*(1-p)% \) of the sample greater than its value; to be precise, that point is the \( \text{ceil}( p * n ) \)-th order statistic. For example, quant.1.c.Q shows how to get various quantiles out of a single box:
for_each_time[ .wt, .quant ]
is_such_that(  
    .quant Is_In [ .25, .5, .75, -.4, 0.0, 2.4, 1.0, .66 ]  
    and .wt Is_In { .z : there_is_a PART where( Weight = .z ) }  
as_quantile .quant  
)do {  
do Write_Line( "The ", .quant, " weight quantile = ", .wt );  
}

Note that if a quantile is undefined (as would be the case with the -.4 and 2.4 quantiles above), then nothing is generated.

All supported Use_Box keywords are given in the prototype for Is_In[] in sys.env.cy.

### 12.4.2 Skolems/Placeholders

Since Daytona uses positional syntax for gaining access to boxes, users may find themselves having to think up names for variables whose values they don’t care about. For example, for box2 below, when summing up the values of the first two components of the TUPLES in the box, the values of the third component are of no interest and yet the syntax requires that something be offered in the third component of the subject to the Is_In satisfaction claim.

```cyl
set .box2 = [ [.a, .b, (.a * .a)+1] :  
    .a Is_In [ 1->4 ] and .b Is_In [ .a -1 -> .a + 2 ]  
 : with_reverse_lexico_order  
    with_sort_spec[ -2, 1] ];

do Write_Line( "Total first two components = ",  
    sum( over .qq each_time(  
        [.x, .y, .who_cares] Is_In .box2 and .qq = .x + .y )) );
```

For boxes with many dimensions, this can be a real nuisance. Fortunately, Cymbal provides ?, a skolem constant or placeholder. This is analogous to the underscore anonymous variable from Prolog and is used just like other variables:

```cyl
set .box2 = [ [.a, .b, .c, .d, .e, .f] :  
    .a Is_In [ 1->4 ] and .b Is_In [ .a -1 -> .a + 2 ]  
    and .c Is_In [ .b -1 -> .b + 2 ] and .d Is_In [ .c -1 -> .c + 2 ]  
 : with_with_sort_spec[ -2, 1] ];

do Write_Line( "Total first two components = ",  
    sum( over .qq each_time(  
        [.x, .y, .z, .w] Is_In .box2 and .qq = .x + .y + .z + .w )) );
```

Feel free to use as many ?s as desired. In fact, as a convenience, an integer prefixed to a ? is considered an abbreviation for a comma-separated list of ? of length equal to the integer. For example, 3? is shorthand for ?, ?, ?. Optionally, one can use the multiplicity syntax used for types as in (3)?.

(The name comes from that of the logician T. Skolem. In logic, skolem constants are abbreviations for an occurrence of some new existentially quantified variable. That is precisely what they are for Cymbal as well.)
12.4.3 Using Box Indices For Searching

The sorting indices that Daytona maintains for boxes may be used not only for generating box elements in a specified order but also for supporting the fast, efficient searching of boxes for certain elements.

For example, when testing to see if a particular TUPLE is in a box, if there is a (non-selection) ordering which involves all TUPLE dimensions (such as `with_lexico_order`), then by specifying that ordering in the Use_Box context, Daytona will conduct a \( \log n \) search to perform the test (where \( n \) is the number of TUPLES in the box); otherwise, the search will be done in linear time. Consider, for example:

```plaintext
set .box2 = [ [.a, .b, (.a *.a)+1] :
    .a Is_In [ 1->4 ] and .b Is_In [ .a -1 -> .a + 2 ]
    : with_reverse_lexico_order
    with_sort_spec[ -2, 1 ] ];

with_format _table_
do Display each .x each_time(
    .x Is_In [ 0 -> 10 ]
    and [ .x, 3, (.x * .x)+1 ] Is_In .box2 in_reverse_lexico_order
);
```

The `Is_In` satisfaction claim for `.box2` in the Display call is a test, not a generator for any values, because `x` has its generating occurrence in the preceding conjunct. Consequently, the term `[ .x, 3, (.x * .x)+1 ]` is ground. (see Chapter 9 for a definition of ground). Since the subject TUPLE is ground and `with_reverse_lexico_order` sorts all of the dimensions, this `Is_In` test will be done in \( \log n \) time.

So far, this chapter has presented `Is_In` satisfaction claims where either all components of the subject TUPLE are defining occurrences for variables or none of them are (i.e., all are ground). Of course, it is certainly permissible for `Is_In` satisfaction claims to contain subject TUPLE components that are a mixture of generating variable occurrences and ground terms.

The rule is that Daytona will use efficient \( \log n \) access to search a box if the ground subject TUPLE components correspond to an initial subsequence of the specified (non-selection) sort order. In other words, efficient search results if, for some \( k \) \( > 0 \), each dimension referenced in the first \( k \) elements of the sort_spec is ground in the subject TUPLE. For example, \( \log n \) access will be used for this query in box.demo.Q which seeks to look up TUPLES in the box based on given values for the second component:

```plaintext
with_format _table_
do Display each[ .x ]
each_time(
    .y Is_In [3,4] and
    [ .x, .y, ? ] Is_In .box2 sorted_by_spec[ -2, 1 ]
);
```

Copyright © 2013 AT&T
All Rights Reserved.
September 15, 2013
Note that the second component of the subject TUPLE is ground and corresponds to the first component of the sort_spec. If the first component of the subject TUPLE was also ground, then efficient $\log n$ access would also be used and since it is more constrained, the resulting list of answers would be contained in the first. However, if only the first component of the subject TUPLE were ground, then since that does not supporting using any sort_spec for searching, Daytona will instead scan the box in linear time but sorted by spec $[-2, 1]$ and honoring the constraint on the first component of the subject TUPLE.

The preceding discussion concerned box indexing when sort orders were explicitly given in the Use_Box situation. One situation where Daytona will use box indexing when no Use_Box sort order is explicitly given occurs when with_default_arbitrary_order is specified during Build_Box and there is some sort_spec for the box present that matches the grounding pattern. This situation will most commonly occur when using a SET because with_default_arbitrary_order is implied and since Daytona removes duplicates by using a lexic_order sort_spec, that order is available for indexing use. As a result, for example, membership tests for SETS are done in $\log n$ time. (For Build_And_Use_Box (hybrid use as discussed later), for SETS, $\log n$ search is done for any grounding pattern with no need for explicitly giving a sort order.)

So, it is useful to remember that selection order is the default for a LIST with_default_selection_order, which is a default characteristic of a LIST. Consequently, if no sorted_by_spec is explicitly provided, then an $Is\ In$ assertion on that LIST will proceed by selection order, regardless of any grounding pattern of the subject TUPLE.

When Tracy is running with -FYI, Tracy will print out which sort spec is being used for a given box search. Here is a sample message:

```
fyi: recognizing that Daytona can add, subtract, merge and extend
    sort specs during optimization, Daytona is using
    sort spec #4 (starting with #0) for the box use near this line nbr

    The approximate or best-guess corresponding Cymbal line number = 25
    in the file `dum.Q'
    The fpp being processed is `Begin'.

fyi: The sort spec chosen is [ 7, -1 ].
```

This mechanism for using the subject grounding pattern to determine index choice is mirrored in Daytona’s indexing capabilities for database (disk) tables, as discussed in Chapter 13.

When a BOX TUPLE element has a component that is of FLT or MONEY type and which is used in indexing the BOX, then the same considerations hold as they do for B-tree indexing and associative arrays: see Chapter 5 and Chapter 11.

### 12.5 BoxFormer Predicates And Hybrid Build/Use

As a convenience, Cymbal provides the *BoxFormerPreds* which support a briefer, consolidated way to specify the construction and use of a box. Recall from before the following two semantically equivalent assertions:
The second assertion uses the `BoxFormerPred Is_Something_Where` and exemplifies the following syntax:

```
BoxAsn | SomeValCalls BoxFormerPred BoundedAsn HybridBoxKeywordArg
Is_Something_Where Is_The_Next_Where Is_The_First_Where Is_The_Last_Where
```

Notice that the subject to a `BoxFormerPred BoxAsn` must be either a single `ValCall` or a TUPLE of such. This is because, in part, the nature of this particular abbreviation is to create a TUPLE of new variables and to substitute those for their corresponding subject variables in the `BoundedAsn`.

12.5.1 Is_Something_Where And Is_The_Next_Where

In particular,

```
[ .x_1 , . . . , .x_n ] Is_Something_Where( A )
```

is considered to be an abbreviation for:

```
[ .x_1 , . . . , .x_n ] Is_In [ [ .v_1 , . . . , .v_n ] : A' ]
```

where the \( v_i \) variables are new and \( A' \) is the result of substituting \( v_i \) for \( x_i \) in \( A \).

Likewise,

```
[ .x_1 , . . . , .x_n ] Is_The_Next_Where( A )
```

is considered to be an abbreviation for:

```
[ .x_1 , . . . , .x_n ] Is_In [ [ .v_1 , . . . , .v_n ] : A' ]
```

`Is_Something_Where` is particularly convenient for getting rid of duplicates in Cymbal queries. The following query is an attempt to discover, for each color, all suppliers which have orders of some part with that color. When this is run on the data from the Daytona test suite, it produces many, many duplicate answers because there several parts with the same color and several orders of the same part from the same supplier.
do Display
   with_title_line
   "for each part color, all suppliers supplying parts of that color"
   each[ .color, .supplier ]
   each_time(
      [ .color, .supplier ] Is_Something_Where(
         there_is_a PART where(
            Number = .part_nbr
            and Color = .color )
         and there_is_a SUPPLIER named .supplier where(
            Number = .supp_nbr )
         and there_is_an ORDER where(
            Supp_Nbr = .supp_nbr
            and Part_Nbr = .part_nbr )
      )
   )

(Also, efficiency-wise, this illustrates in general a poor way to write a query: the consecutive PART and SUPPLIER generating assertions have no ‘join’ condition, i.e., there are no shared variables to cause the SUPPLIER generator to be constrained by the PART generator: a cross-product results. It is probably more efficient to order the conjuncts PART/ORDER/SUPPLIER, although it wouldn’t be if the cardinalities of PART and SUPPLIER are small and there are many ORDERS per PART.)

A quick Is_Something_Where can be thrown in to eliminate the duplicates (cf., colorsupp.*.Q):

do Display
   with_title_line
   "for each part color, all suppliers supplying parts of that color"
   each[ .color, .supplier ]
   each_time(
      [ .color, .supplier ] Is_Something_Where(
         there_is_a PART where(
            Number = .part_nbr
            and Color = .color )
         and there_is_a SUPPLIER named .supplier where(
            Number = .supp_nbr )
         and there_is_an ORDER where(
            Supp_Nbr = .supp_nbr
            and Part_Nbr = .part_nbr )
      )
   )
By throwing in Is_Something_Wheres in a different way to create two boxes (and re-ordering the conjuncts in a more join-oriented way), a more efficient query results that prints out all the suppliers for one color in a single consecutive group, as opposed to several groups as the preceding query will do.

```plaintext
do Display
   with_title_line
     "for each part color, all suppliers supplying parts of that color"
   each[ .color, .supplier ]
   each_time(
      .color Is_Something_Where( 
         there_is_a PART where( Color = .color ) )
      and .supplier Is_Something_Where( 
         there_is_a PART where( 
            Number = .part_nbr
            and Color = .color )
         and there_is_an ORDER where( 
            Supp_Nbr = .supp_nbr
            and Part_Nbr = .part_nbr )
         and there_is_a SUPPLIER named .supplier where( 
            Number = .supp_nbr )
      ) )
```

As explained in Chapter 14, SET boxes are frequently used in aggregation queries to define the initial partitioning into groups. The above query is using one to group-by Color.

*Is_The_Next_Where* and *Is_Something_Where* are both predicates that take keyword arguments. The next (pedagogically motivated) query (allcolors.8.Q) illustrates many of them in action:
do Display
    with_title_line "some colors in the PART table"

    each[ .color, .nbr ]

    each_time(  
        .color Is_Something_Where(  
            there_is_a PART where( Color = .color ) )  
        with_sort_directions[ _desc_ ]  
        with_candidate_index_vbl nbr2  
        selecting_when( .nbr2 % 2 = 0 )  
        stopping_when( .nbr2 > 14 ) 
    
    and  
    .color Is_The_Next_Where(  
        there_is_a PART where( Color = .color ) )  
        with_no_duplicates  
        with_candidate_index_vbl nbr  
        selecting_when( .nbr < 12 ) 
)

The keyword arguments that are allowed for each \texttt{BoxFormerPred} are given in the file \texttt{sys.env.cy}. They include some \texttt{BuildBoxKeywds} and some \texttt{UseBoxKeywds}.

The next query (supporders4.IQ) illustrates using boxes as in-memory (which they always are) tables with what amount to be indices. The point of this query is to prompt the query repeatedly for a Supplier and then to print all the order numbers associated with that Supplier. To speed up the execution of the query, a keyed \texttt{Is_Something_Where} box is used.
locals: STR: .supplier

loop {
  skipping 1 do Exclaim( "Enter supplier: " )
  do Read_Line( supplier );
  when( At_Eoc ) leave;
  skipping 2
  do Write_Line( "Supplier = \", .supplier, \"" );
  for_each_time .order_nbr is_such_that(
    there_is_a SUPPLIER named .supplier where( Number = .supp_nbr )
    and [ .supp_nbr, .order_nbr ] Is_Something_Where(
      there_is_an ORDER where(
        Number = .order_nbr
        and Supp_Nbr = .supp_nbr ) )
  )
  do { do Write_Line( rb( .order_nbr, 9) ); }
  do Exclaim_Line;
}

The reason why this box use is keyed is because the supp_nbr variable is already finitely defined on first use when the box is first encountered. The value of the given supp_nbr is used in a search tree lookup within the box to retrieve only the order_nbrs associated with that supp_nbr.

### 12.5.2 Is_The_First_Where And Is_The_Last_Where

Sometimes, even though an assertion can be satisfied in many ways, it is desired to work just with one of the ways, in fact, the very first one. Is_The_First_Where provides an easy way to do this:

\[
[ .x_1, \ldots, .x_n ] \text{Is_The_First_Where}( A )
\]

is considered to be an abbreviation for:

\[
[ .x_1, \ldots, .x_n ] \text{Is_In}
[ [.v_1, \ldots, .v_n ] : A' \text{with_candidate_index_vbl z stopping_when( z = 1 ) }]
\]

and

\[
[ .x_1, \ldots, .x_n ] \text{Is_The_First_Where}( A ) \text{in_lexico_order}
\]

is considered to be an abbreviation for:

\[
[ .x_1, \ldots, .x_n ] \text{Is_In}
[ [.v_1, \ldots, .v_n ] : A' \text{with_sort_index 1 in_lexico_order }]
\]

allcolors.9.Q provides an example of its use where the first Is_The_First_Where assertion serves to generate a value for color and the second one serves (unnecessarily) to test it, thus illustrating both and all possible kinds of uses of Is_The_First_Where.
Cymbal also supports the opposite of Is_The_First_Where, namely, Is_The_Last_Where.

\[ [ .x_1, \ldots, .x_n ] \text{ Is_The_Last_Where( } A ) \]

is considered to be an abbreviation for:

\[ [ .x_1, \ldots, .x_n ] \text{ Is_In } [ [ .v_1, \ldots, .v_n ] : A' with_selection_index _last_ ] \]

and

\[ [ .x_1, \ldots, .x_n ] \text{ Is_The_Last_Where( } A ) \text{ in_lexico_order} \]

is considered to be an abbreviation for:

\[ [ .x_1, \ldots, .x_n ] \text{ Is_In } [ [ .v_1, \ldots, .v_n ] : A' with_sort_index _last_ \text{ in_lexico_order} ] \]

### 12.6 Incremental Additions And Deletions To Boxes

It is often useful to be able to incrementally add and delete elements of a box one by one. Here are some examples of box adds and deletes (box.del.1.Q):

```plaintext
{ local: SET{ TUPLE[ INT, INT ] : with_deletions_ok } .box4

   do Change so_that( [4,3] Is_In .box4 );
   do Change so_that( ! [4,3] Is_In .box4 );
   do Change so_that( [2,1] Is_In .box4 );
   do Change so_that( [2,1] Is_Not_In .box4 );
   do Change so_that( [9,9] Is_In .box4 );

   do Write_Words( "box4 has", .box4.Elt_Count, "elements." );
   set .box4 = {}; /* quick emptying of entire box */
   do Write_Words( "box4 has", .box4.Elt_Count, "elements." );
}
```

Note that the use of = {} causes the box to become empty but it does not free the associated space at all! See free_on_return when for that purpose. Note that = {} is used for emptying SETs and = [] is used for emptying LISTS.

Cymbal’s Change procedure is used to specify changes to boxes (and record classes) in a
declarative way. The idea is that Change will do the least work possible to ensure that the do Change assertion is true after the call returns. So, for the first invocation above, if \([4,3]\) is not in .box4 before the invocation, then it will be afterwards.

The assigning of {} to be .box4 is the quickest way to empty a box. If the box were a LIST, [] would have to be used instead as the assigned value in order to empty the LIST.

As another example, here is one way to delete the smallest element in a box (top-k.2.Q):

```plaintext
for_the_first_time [ .date_placed, .nbr, .qty ] ist(
    [ .date_placed, .nbr, .qty ] Is_In .ord_box
    with_sort_index _last_ sorted_by_spec[ -1 ]
})
    do Change so_that(
        [ .date_placed, .nbr, .qty ] Is_Not_In .ord_box );
```

`for_the_first_time` is needed here because of the stricture that `for_each_time` loops cannot change the value of a variable that is defined outside the `for_each_time` and is used within it. Fortunately, since there can only be one _last_ element, using `for_the_first_time` does not eliminate any iterations of interest.

Actually, for boxes that contain TUPLES with a large number of components, it is tedious to identify each TUPLE component in order to delete a TUPLE from the BOX. Fortunately, element VBL VBLs can eliminate this tedium (top-k.2.Q):

```plaintext
for_the_first_time .ptu ist(
    ? Is_In .ord_box with_sort_index _last_ sorted_by_spec[ -1 ]
        with_elt_vbl_vbl ptu
})
    do Change so_that( ..ptu Is_Not_In .ord_box );
```

12.6.1 Is_In_Again For LIST additions

Since LISTS allow duplicate entries, the `Is_In_Again` predicate is needed as the basis for the Change assertion in order to get an element in the LIST more than once (box.del.1.Q):
{ local: LIST[ TUPLE[ INT, INT ] ] .box4
    do Change so_that( [4,3] Is_In .box4 );
    do Change so_that( [4,3] Is_In .box4 );
    do Change so_that( [4,3] Is_In_Again .box4 );
    do Change so_that( [4,3] Is_In_Again .box4 );
    /* now .box4.Elt_Count = 3 */
    with_sudden_death
    do Change so_that( [4,3] Is_Not_In .box4 );
    /* now .box4.Elt_Count = 2 */
    do Change so_that( [4,3] Is_Not_In .box4 );
    /* now .box4.Elt_Count = 0 */
}

The rationale is that Cymbal is intended to read like (mathematical) English: if [4,3] is already a member of the LIST, then the assertion:

[4,3] Is_In .box4

is already true and hence, there is nothing to change. Likewise, with the last statement above, if Daytona is asked to change .box4 so that [4,3] is not in the LIST, then all occurrences of [4,3] will be removed.

However, when ties are possible and if the user would like to guarantee that exactly one is deleted, then the Change PROC keyword with sudden death can be used to cause the Changing to cease as soon as one TUPLE is deleted.

Also, it is important to note that both selection and sort indices get recomputed from scratch after any of these Change procedure box adds or deletes (box.sel.1.Q).

### 12.6.2 Updating A Box Element By Deletion Followed By Insertion

A generally useful but slow way to update an element of a box is to delete it and then insert its updated value. It is also possible by means of using VBL VBL box elements to update box elements in place very quickly but the former, simpler way is discussed first. The primary advantage of this delete+insert paradigm is that there is no restriction that forbids changes to components of box element TUPLES that are being sorted on.

Here is a PROC which takes a BITSEQ .z_val and a BOX .z_box of [BITSEQ, INT] pairs as arguments and updates the BOX so that if .z_val is not the first member of some TUPLE in the BOX, then [.z_val,1] will be inserted in the .z_box, whereas if otherwise, then the INT count associated with .z_val will be bumped by one.
define PROC(
    alias SET{ TUPLE[ BITSEQ(30), INT] : with_sort_indices_stored
        with_sort_specs[ [-2], [1] ] with_deletions_ok } .z_box,
    BITSEQ(30) .z_val )
Update_Box
  do {
    for_the_first_time .z_cnt
    is_such_that(
        [ .z_val, .z_cnt ] Is_In .z_box
    ) do {
      set .tmp_cnt = .z_cnt+1;
      do Change so_that( [ .z_val, .z_cnt ] Is_Not_In .z_box );
      do Change so_that( [ .z_val, .tmp_cnt ] Is_In .z_box );
    } else {
      do Change so_that( [ .z_val, 1 ] Is_In .z_box );
    }
  }

  Note that we have to use for_the_first_time here (instead of for_each_time) because the do-group of the loop changes the value of z_box, which is an outside variable for the for_the_first_time assertion. Fortunately, by the construction, we know that there is at most one element in the box for any given BITSEQ.

  Note also that since Update_Box changes .z_box, z_box must be an alias VBL parameter to this PROC. Also, since .z_box is being deleted from, with_deletions_ok must be a part of the type specification. The reason behind this is that boxes that support deletion require more overhead in code and space than boxes that are used for adds alone: consequently, the default configuration for a box does not support deletions. Daytona will try to infer when a box needs to support deletions but nonetheless, when defining parameter BOX variables, if the associated BOXES need to support deletion, then with_deletions_ok needs to appear explicitly in the type specification for the parameter BOX variable.

  The following queries are illustrative: box.demo.Q, box.add.1.Q, box.del.1.Q, box.export.1.Q, box.parm.1.Q. Also, box.group.2.Q contains a nice example of using boxes to compute two group-by queries simultaneously in one pass through the data. (In general though, when dynamic associative arrays can be used, they are much faster than using boxes.)

12.6.3 Updating Box Elements In Place

  Instead of updating a box element by reading it, deleting it, and re-inserting a modified version of it, it is much faster to find a target box element TUPLE, get what amounts to be a pointer to it, and change the box element in place by accessing it directly through its pointer.

  In general, in-place-box-element-updating can only be done if modifying the element will not invalidate any sort that exists on the box elements. This can be accomplished in one of two ways. If
the BOX is a LIST and there are no sorts specified at all, then box elements can be updated in place either by looping over all of them or by identifying one or more by its selection index. Otherwise, if the BOX is a SET or has any sorts defined at all, then two conditions must be met before such in-place updates can be supported by Daytona. First, there must be at least one sort spec that is designated as a unique key, meaning that the box can never contain two TUPLES that agree on the components specified by any unique key sort spec. Second, no component of a box element TUPLE can be updated in-place in this fashion if it is sorted on by any sort spec, unique or not. When these conditions hold, the user can identify the element (or elements) to be updated in place in any fashion, either by their selection index, or by looping over all of them, or by accessing them by a unique or non-unique index (sort). If all these conditions cannot be met, then the delete-followed-by-insert paradigm must be employed.

There are two principles operative here. The first is that for simplicity and predictability for the semantics of duplicates, Daytona requires that for any BOX that has a sort (which includes a SET) that either there is a sort on all the components of a TUPLE or that there is a unique sort on some of the components of a TUPLE. The second principle is that no update-in-place can result in the invalidation of any existing sort.

The following example, taken from box.update.1.Q, begins with showing how to define a sort spec to be a unique key:

```
set .bbb = [ [1,2,3,4], [2,3,4,5], [3,4,5,6], [4,5,6,7] :
   with_no_duplicates with_sort_specs[ [2], [1 :: as_unique_key ] ] ];
for_each_time .bbb_elt_p is_such_that(
   .i Is_In [ 1 -> 4 ]
   and [ .i, 3? ] Is_In .bbb with_elt_vbl_vbl bbb_elt_p
) {
   set ..bbb_elt_p = [ ?, ?, ..bbb_elt_p#4+1, $#4+1 ];
   set ..bbb_elt_p#3 += 1;
   set ..bbb_elt_p#4 = ..bbb_elt_p#3 + 1;
}
```

The first double colon and then the double colon in [1 :: as_unique_key] may seem a little unusual until it is remembered that in general there are three parts to a LIST: the first is a listing of the elements, the second is some assertion, if any, they must satisfy, and the third is any sequence of keyword arguments that further characterizes the TUPLE. See the discussion above.

The in-place box element updating is accomplished by using the with_elt_vbl_vbl ancillary VBL keyword to define a VBL VBL which will serve as a pointer to the current box element. Daytona appropriately determines the type of the VBL VBL, which in the case of bbb_elt_p in the example above is:

```
TUPLE[ INT, INT, INT, INT ] VBL VBL
```

Consequently, for example, the way to refer to the 4th component of the current element of .bbb in the for_each_time loop above is to write ..bbb_elt_p#4.
In order for there to be a unique key, either a \( \text{LIST[ with_no_duplicates ]} \) must be used or else a \( \text{SET{}} \).

Note the use of two skolems in the right-hand-side of the first assignment. They indicate that the corresponding elements in the LHS are not to be changed.

The following example, taken from box.sel.1.Q, shows how an element VBL VBL can be used to update a \( \text{LIST of INTs} \) by selecting out a couple elements in particular by means of their selection index:

```plaintext
local: \text{LIST[ INT ] .bbb}
set .bbb = [ 1, 3, 6, 9, 12, 15, 18 ];
set eltvv = (.idx = 4 |\#5) and ? Is_In .bbb with_elt_vbl_vbl eltvv with_selection_index .idx
set ..eltvv#1 = 0;
```

Of course, \( \text{LISTs of TUPLES} \) can also be updated in this fashion but this example serves to highlight that the value of an element VBL VBL is always a \( \text{TUPLE} \). The \( \text{LIST} \) may look like a \( \text{LIST of INTs} \) but in reality, that is just a pleasant abuse of notation -- it is implemented as a \( \text{LIST of singleton TUPLES} \) and so therefore \( \text{eltvv} \) has type \( \text{TUPLE[ INT ]} \).

### 12.6.4 Using A for each time Loop To Add/Delete/Update Some Box Elements

A \( \text{for each time} \) loop provides a natural paradigm for the task of deleting every element of a set that satisfies some condition. However, one has to take care to honor the stricture of modifying outside VBLS (in this case, the BOX VBL itself) in the do-group that also appear in the \( \text{for each time} \) assertion. A general way to escape this stricture is in effect to confine one’s read access to the target box by constructing a temporary box to hold what is of interest (box.del.3.Q):

```plaintext
fet .a = (.a Is_The_Next_Where (.a Is_In .bbb and .a \% 5 = 0 ))
do Change so_that (.a Is_Not_In .bbb )
```

This is an exception to the rule that is supported by Daytona because by the time \( \text{bbb} \) can receive the first change, all read access of \( \text{bbb} \) has taken place (due to the construction of the box associated with \( \text{Is_The_Next_Where} \)). This same exception applies to any other box modifications that may be desired in a loop context. Note that the temporary box is not a copy of the original box; indeed, it simply identifies only those TUPLES in the original box that are going to be changed. That may only be a few TUPLES compared to the size of the original box.

### 12.6.5 Deleting Box Elements At Given Positions

The following query shows how easy it is to delete elements of boxes at given selection_index
or sort_index positionsi (box.del.5.Q)

\[
\begin{align*}
\text{local: } & \text{SET( TUPLE[ INT, INT ] ) } .\text{bbb} \\
& \text{set } .\text{bbb} = \{ \{ .x, .y \} : .x \text{ Is_In } [ 10 \rightarrow 1 \text{ by } -1 \} \text{ and } .y = .x^*.x \}; \\
& \text{fet .si Is_In } [ 6, 3 ] \\
& \{ \text{do Change so_that_previous( ? Is_Not_In .\text{bbb} with_selection_index .si )}; \\
& \text{fet .tu Is_In .\text{bbb} ( do Write_Words( .tu ); )} \\
& \}\text{fet .si Is_In } [ 7, 2 ] \\
& \{ \text{do Change so_that_previous( ? Is_Not_In .\text{bbb} with_sort_index .si )}; \\
& \text{fet .tu Is_In .\text{bbb} in_lexico_order ( do Write_Words( .tu ); )} \\
& \}
\end{align*}
\]

The Change statements should be read as they sound, i.e., change the situation so that the previously existing something is not in the box at the stated position.

### 12.7 Caching And Recomputing The Values Of BOX VBLs

When the definition of a BOX variable includes the static keyword or if it is an exported variable, then the value of the BOX variable is maintained from one invocation of its enclosing fpp to the next, which is just the behavior expected of static and exported variables. The only exception is if the free_on_return_when or free_on_begin_when mechanism is also being used. On the other hand, if a BOX variable is local and not static, then the value of the variable is lost on exit of the enclosing fpp. So, this is one form of caching the value of a BOX VBL so as to reuse it and thus avoid recomputing it. Furthermore, when a BOX VBL is defined (declaratively) by an assertion, Daytona will implement it in a way that will cache its value until it has a good reason to discard the cached value and recompute it.

So, related to but still distinct from the notion of caching BOX variable values is the criteria by which boxes are rebuilt, i.e., recomputed, even within a single execution of their enclosing fpp. A box is always fully constructed once to begin with and is then re-used without rebuilding zero or more times until one of the following happens:

1. Some of its outside variables (if any) change their values.
2. Some transaction within the process has run in the interim.
3. The box building occurs within a transaction and the transaction is invoked again or, within a given execution of the transaction, the transaction saves data to disk after building or rebuilding the box.
4. The variable is neither static (nor export-ed, which implies static) and the flow of control exits the enclosing fpp.

When any of these events occur, the box is rebuilt in the new environment. Consider, for example, the
following helper function definition from box.cache.1.Q:

```plaintext
define INT FUN( INT .key ) look_up
do {
    import: DATE .first_date
    local: static SET{ TUPLE[ INT(_short_), INT(_short_) ] } .lu_box
    set .lu_box = { [ .s_nbr, .qty ] :
        there_is_a SUPPLIER where( Number = .s_nbr ) and
        there_is_an ORDER where(
            Supp_Nbr = .s_nbr
            and Quantity = .qty
            and Date_Recd =
            min( over .dt
                 each_time(
                     there_is_an ORDER where(
                        Supp_Nbr = .s_nbr
                        and Date_Recd = .dt
                        which_is >= .first_date )))
    )
    set .ret = -1;
    for_each_time .q is_such_that(
        [.key, .q ] Is_In .lu_box
    )
    { set .ret = .q;
    }
    return( .ret );
}
```

The purpose of this helper is to look up for a given supplier number, that quantity associated with
the first ORDER whose Date_Recd is >= .first_date. This is accomplished by storing the
appropriate quantity for each supplier in a box. Note the min aggregate function nicely nested within
the there_is_an ORDER. Also, note that the for_each_time search on .lu_box will occur in \( \log n \)
time since the box is a SET and lexico ordering is quietly used by Daytona to remove duplicates: hence
it is available for suitable index lookups as well since with_default_arbitrary_order is implied for SETS.

Relative to the box, first_date is an outside variable: whenever look_up has been invoked and
the value of first_date has changed since the last use of the box, the box will be recomputed.
Similarly, recomputation will occur if a transaction within the same process runs between invocations of
look_up (and this is because such a transaction could change the contents of the ORDER table).
Otherwise, the contents of box will be re-used from its cache. So, even though on the surface, the
Cymbal looks like it is going to assign a box to lu_box every time look_up is called, in fact, it does
not since the system uses caching to minimize box construction.
12.8 Sorting By v10sort Instead Of Boxes

It turns out that using a BOX just to sort TUPLEs in memory can be very time-consuming. Alternatively, and somewhat paradoxically, it can be much, much faster to use v10sort to sort TUPLEs externally and then read them back in. The following query shows how to do this by using a bipipe to v10sort the contents of a dynara (sort_ara_with_v10sort.Q):

```plaintext
INT .sorted_onbr
for_each_time[ .ord, .supp, .part, .quant] ist(
    there_is_an ORDER where(
        Number = .ord and
        Supp_Nbr = .supp and
        Part_Nbr = .part and
        Quantity = .quant )
    ){                            
            set .orders[.ord] = [.supp, .part, .quant];
    }
// sorting on Part_Nbr, Supp_Nbr, ORDER Number but NOT Quantity
set .sort_chan =
    new_channel( via _bipipe_ with_mode _update_ for "v10sort -n -t'|'")
otherwise
    with_msg "Failed to open sort channel" do Exit(1);
for [.ord, .supp, .part] ist(
    .orders[.ord] = [.supp, .part, ?]
    ){
            to .sort_chan with_sep "|" do Write_Line( .part, .supp, .ord );
    }
with_mode _write_ do Close(.sort_chan);
loop {
    set [?, ?, .sorted_onbr] = read( from .sort_chan upto "|\n")
otherwise{ break; }
    set [s, p, q] = .orders[.sorted_onbr];
    when( .sorted_onbr % 10 = 2 )
        with_sep "|" do Write_Line( p, s, .sorted_onbr, q );
}
do Close(.sort_chan);
```
13. Advanced Topics For Cymbal Descriptions

This chapter explains in detail a number of useful properties and extensions of the Cymbal description construct. The first section describes how to write descriptions so that Daytona will automatically use indices to speed up the retrieval of data. Partial match indexed queries are described in the second section. The third section describes various miscellaneous capabilities such as Cymbal’s machinery for working with missing values, the this_is_a construct for reusing data buffers, and dirty reads. The fourth and last section describes various ways to access data by position in the data file and index.

This chapter introduces a number of keyword arguments that can be added to Cymbal descriptions. The precise syntax permissible for these keyword arguments is given in the import for Any_There_Isa in sys.env.cy.

```plaintext
import:
PRED[
( 0->1 ) using_siz,
( 0->1 ) using_reverse_siz,
( 0->1 ) using_no_index,
( 0->1 ) ignoring_failed_opens,
( 0->1 ) with_ignoring_failed_opens_handler BOOL FUN( STR ) = _null_fun_,
( 0->1 ) with_dirty_reads_ok,
( 0->1 ) keyed_for_index manifest INT|STR|LIT|THING,
( 0->1 ) sorted_by_index manifest INT|STR|LIT|THING,
( 0->1 ) at_bin_cluster_elt_pos INT(_huge_),
( 0->1 ) at_bin_pos INT,
( 0->1 ) with_bin_pos_vbl alias INT(_huge_),
( 0->1 ) at_bin_offset INT(_off_t_),
( 0->1 ) with_bin_offset_vbl alias INT(_off_t_),
( 0->1 ) from_a_bin_sample_of_size INT,
( 0->1 ) from_a_bin_sample_of_frac FLT,
( 0->1 ) from_version INT|STR|LIT|THING|FLT|DATE,
( 0->1 ) using_source STR|LIT,
( 0->1 ) with_lock_mode manifest _3GL_TEXT, /* LOCK_MODE */
( 0->1 ) skipping INT,
( 0->1 ) from_section manifest TUPLE[ INT, INT ]
] Any_There_Isa
```

13.1 Keys, Indices And Descriptions
13.1.1 Finding Out What Indices Daytona Has Chosen, If Any

By running Tracy with the -FYI option, the user can easily find out what indices Daytona is using for the Cymbal descriptions in a given query. If query performance is disappointing, this should be the user’s first diagnostic step. Indices can make the difference between now and never in databases and so, it is important to make sure that they are being used appropriately. As an example of how to find out what indices are being used, when this command is run:

    Tracy -FYI -r Q/hparti.2.Q >dum.out 2>&1

Then the contents of dum.out contain:

```
fyi: the access to HOPPER near Cymbal line `9'
   is using a horizontal partitioning KEY for Condition `='
   with FIELDS: Hopper_Nbr
fyi: using the box-of-key-field-values rewriting
fyi: the access to HOPPER near Cymbal line `9'
   is using file access method `Unique_Btree_Key_Fam'
   with INDEX `1' for KEY `1'
   with FIELDS: Status, Entry_Nbr
fyi: using the box-of-key-field-values rewriting
fyi: the access to HOPPER near Cymbal line `22'
   is using a horizontal partitioning KEY for Condition `>='
   with FIELDS: Hopper_Nbr
fyi: the access to HOPPER near Cymbal line `22'
   is using file access method `Next_By_Offsets_Fam'
   with INDEX ` ' for KEY ` '
   with FIELDS: Hopper_Nbr
fyi: the access to HOPPER near Cymbal line `33'
   is using a horizontal partitioning KEY for Condition `<='
   with FIELDS: Hopper_Nbr
fyi: the access to HOPPER near Cymbal line `33'
   is using file access method `Next_By_Juxta_Fam'
   with INDEX ` ' for KEY ` '
   with FIELDS: Hopper_Nbr
```

If reference is made to Btree_Key_Fam (Unique or NonUnique), a Btree index is being used for that access. On the other hand, Next_By_Offsets_Fam and Next_By_Juxta_Fam indicate sequential access. hparti KEYS are used when searching in memory for which horizontal partitioning bin or bins are to be accessed. The concept of box-of-key-field-values is discussed later in this chapter.

If Daytona thinks that creating an INDEX could help, then Tracy will print out a warning like this:
warning: this query may run faster if an INDEX is specified in the rcd for RECORD CLASS 'ORDER' so that the INDEX uses as few of the following FIELDS
as needed to minimize the __Avg_Reach for the INDEX:

   Quantity

The approximate or best-guess corresponding Cymbal line number = 1
   in the file `-' .
The fpp being processed is `Begin’.

Remember however that indices are no panacea -- there are plenty of queries, especially 'group by' queries, that are best run with sequential access. In our experience, if the query wants information out of more than, say, 15% of the rows of a table then for sure sequential access will result in faster query execution wall clock times (although they will eat up a lot more CPU time) than going in through an INDEX, were that possible. But anyway, for doing joins, indices are often the best way when using Daytona -- and certainly the easiest to cause Daytona to use, if they are available.

Daytona’s process for choosing an index is not sensitive to the order of the conjuncts in a query there_isa.

Remember that Daytona does not yet do join order optimization. So, the speed of a join query can be increased if the order of appearance of the tables of a join is permuted. A good rule of thumb is to order the joined table accesses by most restrictive to least restrictive in terms of the number of rows that satisfy them.

13.1.2 Hard-coding Index Choice

Also, Daytona provides the keyed_for_index manual-override keyword for use in queries. A typical usage would be:

   there_is_an ORDER keyed_for_index `sp` where( ... )

This usage would mandate that an sp index exist, that a key for it be given in the corresponding Cymbal description and that the sp index be used for accessing the data records. Since this defeats Daytona’s index selection mechanism, its use is only recommended in special circumstances.

To mandate that Daytona visit all the data file records in the order that keys appear in a specified index, use the sorted_by_index keyword as in:

   there_is_an ORDER sorted_by_index `sp` where( ... )

This will require that Daytona use the sp index to visit all data file records in the order of the keys stored in INDEX sp. Since there is no point in doing this if the data file records are stored in the order of this INDEX's keys (in which case, a sequential .siz scan is indicated), the necessary implication is that sorted_by_index scan will involve lots of (slowish) seeks to widely separated data file locations. Also, since unless terminated prematurely by user directive, all index entries will be visited, there is no point in using sorted_by_index if there are constraints on the corresponding KEY fields that could result in fewer records being (directly) accessed by virtue of regular keyed B-tree access (perhaps by using
keyed_for_index explicitly).

To mandate that Daytona use the .siz file to sequentially scan records in the record class in the order of their appearance in the data files, simply use the no-argument keyword using_siz as in

there_is_an ORDER using_siz where( ... ). Using the keyword using_reverse_siz will cause the records in the record class to be accessed in the reverse order of their appearance in the corresponding data files.

To mandate that Daytona use no index (including the .siz file) when scanning the records in a record class in the order of their appearance in the data files, simply use the no-argument keyword using_no_index as in there_is_an ORDER using_no_index where( ... ).

When Tracy is run with the -FYI option, fyi messages will be run indicating which indices are being used. For example:

```
fyi: the access to ORDER near Cymbal line '7'
  is using file access method 'NonUnique_Btree_Key_Fam'
  with INDEX 'dp' for KEY 'dp'
  with FIELDS: Date_Placed
```

### 13.1.3 Boxes Of Key Field Values

Daytona is always on the lookout to use indices to answer queries. There is however one particular situation where it may or may not be advantageous to use an index and where Daytona needs help in determining whether or not to use available indices. This is where the user is asserting that one or more FIELDS in some KEY has values in some BOX. Sometimes it is faster (or the output will appear in a more desirable order) if the related description is processed without using any indices. Other times, it will be better to use an index.

The Cymbal description writer can choose either one by employing special syntax. Specifically, if one or more KEY FIELDS is asserted to have values in a BOX, then an INDEX for the associated (possibly multi-field) KEY will be considered for use (along with whatever other INDICES are available).

For example, the following Display query asserts that the value of the Unique KEY FIELD Number is in a LIST and consequently, the INDEX on Number will be used instead of a linear scan through the table.

```
do Display each [ .order_nbr, .qty ]
each_time(
    there_is_an ORDER where(
        Number = .order_nbr which Is_In [ 1, 5, 7 ]
        and Quantity = .qty )
);```

In fact, what Tracy does when it recognizes a box of key field values situation and decides to use the associated INDEX, is to rewrite the query by moving the BOX membership assertion in front of the there_isa assertion, as in:
do Display each .order_nbr
each_time(
   .order_nbr Is_In [ 1, 5, 7 ]
   and
   there_is_an ORDER where(
      Number = .order_nbr
      and Quantity = .qty )
);

This reduces the query to a fundamental structure that recommends the use of keys, i.e., where the query asserts that a KEY field is equal to some previously computed (or constant) value. For example, in the above, when Tracy is considering Number = .order_nbr, it has already processed the previous conjunct and therefore it knows that order_nbr will be taking values in a TUPLE. Those values will be available to the program when it wishes to find the associated ORDER and the best way to find that ORDER is by means of the INDEX on Number.

Here are several more examples of box-of-key-field-values patterns in queries:

set .box_a = [ 1, 5, 7 ];
do Display each[ .order_nbr, .qty ]
each_time(
   there_is_an ORDER where(
      Number = .order_nbr which Is_In .box_a
      and Quantity = .qty )
);

/* Supp_Nbr is the first FIELD in a 2 FIELD KEY */
do Display each[ .order_nbr, .qty ]
each_time(
   there_is_an ORDER where(
      Number = .order_nbr
      and Supp_Nbr Is_In { 400, 402, 403 }
      and Quantity = .qty )
);

/* Supp_Nbr is the first FIELD in a 2 FIELD KEY */
do Display each[ .order_nbr, .qty ]
each_time(
   there_is_an ORDER where(
      Number = .order_nbr
      and Supp_Nbr = .s_nbr which Is_Something_Where(
         there_is_a SUPPLIER where( Number = .s_nbr and City ~= "New" ))
      and Quantity = .qty )
);

Here is a special case that employs the INTERVAL syntax for boxes:
do Display each [.order_nbr, .qty]
    each_time(
        there_is_an ORDER where(
            Number = .order_nbr which Is_In [300 -> 400]
            and Quantity = .qty)
    );

This query will be handled using the box-of-key-field-values transformation. Please note that the box-of-key-field-values optimization will not be done if inequalities are used:

do Display each [.order_nbr, .qty]
    each_time(
        there_is_an ORDER where(
            Number = .order_nbr which_is >= 300 & <= 400
            and Quantity = .qty)
    );

Here is another special case where no BOX is used but the transformation takes place anyway (as would be desired):

    set [.z] = read(from _cmd_line_ but_if_absent[0]);
do Display each [.order_nbr, .qty]
    each_time(
        there_is_an ORDER where(
            Number = .order_nbr which = .z
            and Quantity = .qty)
    );

Fortunately, Tracy will use the Number KEY even though there is no BOX in this query and even though the query’s assertion about Number is a variant of the fundamental one Number = .z.

Here are two examples where a FIELD value is being asserted to be one of several values but where BOX syntax is not being employed; consequently, no INDEX will be used.
do Display each[ .order_nbr, .qty ]
each_time(
    there_is_an ORDER where(
        Number = .order_nbr
        for_which( .order_nbr = 1 or .order_nbr = 5 or .order_nbr = 7 )
        and Quantity = .qty )
); 
do Display each[ .order_nbr, .qty ]
each_time(
    there_is_an ORDER where(
        Number = .order_nbr
        and Number = 1 | = 5 | = 7
        and Quantity = .qty )
); 

In summary, the rule is that when a Cymbal description asserts that a KEY FIELD value is a
member of a BOX, then an INDEX associated with that KEY might be used (i.e., it is available for use
but maybe it won’t be used because some other INDEX is better). However, when an explicit or’ing is
employed or inequalities are used, then no INDEX associated with that FIELD will be used.

In general, performance will be improved if the box of key field values is sorted in key order so
that the index can be searched sequentially.

13.2 Partial Match Indexed Retrieval

Daytona offers several ways to retrieve records based on partially matching their key values.

13.2.1 Starts_With: Indexing On Prefixes

The Starts_With PREDICATE can be used in conjunction with a regular or cluster B-tree index
to quickly produce all records whose keys start with the given quantity.

Here’s an example where there is a btree index built on Telephone:

do Display each[ .name ]
each_time(
    there_is_a SUPPLIER where(
        Name = .name
        and Telephone Starts_With "212"
        and Telephone Contains "9" )
); 

Daytona will use the Telephone btree index to quickly produce all SUPPLIER records whose Telephone
numbers begin with “212”.

This prefix search feature can only be used with the following types of FIELDS: STR, LIT, HEKA,
HEKSTR, SAFE_STR, ISO8859, BASE64(utf_), IP, and IPORT. It can also be used to select records
whose DATE_CLOCK-valued FIELDS start with a DATE. It also works with LIST- or SET-valued
FIELDS of the aforementioned types by using syntax like:

```
one_of_the Phones = .ph which Starts_With .prefix
```

In the case of IP and IPORT, the heka-encoding of these types is such that the user can always search for octet prefixes, as in (prefixsrch.1.Q):

```
do Display each[ .nbr, .ip ]
each_time(
  there_is_a SMORGAS4 where(
    Number = .nbr and Local_Ip = .ip which Starts_With ^135.205.80^IP
  )
);
do Display each[ .nbr, .iport ]
each_time(
  there_is_a SMORGAS4 where(
    Number = .nbr and
    Local_Iport = .iport where( .iport Starts_With ^135.205.80^IP ))
);
```

However, both range queries and subnet queries offer more flexible and powerful ways to use indices to support querying of records by IP address.

Here from prefixsrch.2.Q is an example of using Starts_With to search for records whose Local_Date_Clock Plain_12 DATE_CLOCK values begin with a certain DATE:

```
fet there_isa SMORGAS3 where( Local_Date_Clock Plain_12 Starts_With ^9/8/96^ )
do {
  where this_isa SMORGAS3 do Describe;
}
```

Somewhat more arcane, either a DATE or a DATE_CLOCK can be said to start with a STR where that STRING would be a prefix to the data as it is stored in its canonical form in the DC data file (prefixsrch.2.Q). (As an example, 1990060100 is the canonical form of 1990-0601@0.

Please note that Starts_With can only be used to work with one FIELD from a KEY and in the case of a multi-field key, the Eq PRED must be used in the there_isa on all KEY FIELDS prior in the KEY definition to the Starts_With FIELD, if there are any such predecessor FIELDS.

Also, regarding the use of Starts_With for HEKA FIELDS, Daytona considers HEKA to be classified into two groups: the "good" HEKA which are the digit strings perhaps with – and . interspersed and the "bad" HEKA which are not good and are stored verbatim after being prefixed with an escape character. Since the bad HEKA are allowed to be completely wild, i.e., anything, Daytona must enforce a protocol to keep the bad HEKA from disturbing queries seeking to "find good HEKA that Starts_With". This protocol is the following: queries that wish to "find good HEKA that Starts_With" will only find good HEKA; queries that wish to "find bad HEKA that Starts_With" will only find bad HEKA.

Furthermore, prefix searching on initial substrings of the Default_Value by using an index will
never find any records. This is true whether or not Ignore_DFR_Missing_Values is ‘yes’, i.e., whether
the B-tree contains an (empty) entry for each Default_Value record occurrence or whether there is
simply no entry at all. As a corollary, prefix queries will never catch any Default_Value associated
with a FIELD whose values are allowed to be missing in the data file. So, users who want sensitivity to
the Default_Value in prefix searching need to write their Cymbal in such a way that it does not choose
to use the B-tree associated with their use of Starts_With.

### 13.2.2 Indexed Range Queries

Range queries arise when a there_isa constrains FIELD values to be within INTERVALS as in:

```
Local_Timepoint Is_In [ ^1990-08-24@00:14^DC -> ^1990-08-24@00:14^DC ]
```

The issue is whether such constraints are to be tested by using indices or else by using simple
inequality tests on found records. In the latter case, some mechanism like sequential search produces
the records to apply the inequality tests to whereas in the former case, a B-tree index is used to identify
records that are known to satisfy the constraint simply by having the index produce them in order.
Thus in the index case, there is no need to apply those tests directly to the records once the index has
been used to retrieve them.

Which approach is faster is generally determined by what fraction of the table satisfies such a
constraint: at some point, the fraction becomes large enough that the disk access time associated with
the index retrieval will exceed the wall-clock time associated with the alternative sequential access
visiting every record to apply the INTERVAL tests. This implies that sequential access will then be
faster in the user’s perception, although it typically chews up far more in terms of CPU and disk
resources. On the other hand, if only a relatively few number of records satisfy the constraint, then the
indexed approach is clearly superior since not only is its wall-clock time less but so also is its
consumption of CPU and disk resources.

As it turns out, Daytona offers two different ways of using indexes to retrieve records satisfying
INTERVAL constraints. The first has already been discussed, i.e., box-of-key-field-values whereby
certain enumerable Is_In INTERVAL satisfaction claims on FIELD values are repositioned before their
original there_isa so that the satisfaction claim becomes a generator, not a test, and each generated
value is used by Daytona to look up any associated records. While effective and useful, this approach
nonetheless has a number of limitations. First, it can only be used on INTERVALS of certain types like
for example DATES and INTS where it is easy and efficient to generate a not unwieldy LIST of all
values in the INTERVAL. On the other hand, INTERVALS of FLOATS, CLOCKS, and STRINGS are
not so amenable. Secondly, it is inefficient to the extent that there is a large number of values in the
INTERVAL that result in no associated records although still incurring the costs of equality-based
indexed search.

So, Daytona offers what it calls *indexed range queries*. Here is an example of one using a
HEKDATE_CLOCK INTERVAL (isin_range.4.Q).
for_each_time [ .nbr, .hdc ]

is_such_that(
    there_is_a SMORGAS5_C where(
        Number = .nbr and
        Local_Hekdate_Clock_Yyyymmdd_24_Hhmm_
        Is_In_Range [ "1990-08-24@00:14"DC -> "1990-08-31@00:15"DC ] and
        Local_Hekdate_Clock_Yyyymmdd_24_Hhmm_ = .hdc
    )
)

do Write_Words( .nbr, .hdc );

As illustrated here, Daytona relies on special syntax whereby the user informs Daytona to try out opportunities for doing indexed range queries. The syntactic clue is the use of the Is_In_Range PREDICATE. Of course, no indexed range query is possible unless there is an rcd-specified B-tree INDEX whose KEY includes the corresponding FIELD; also, no indexed range query implementation will be employed if Daytona can determine a better way to use any other B-tree INDICES. In those cases, the use of Is_In_Range is treated as if it were simply an Is_In-based test. However, in the case of this SMORGAS5_C, there is an INDEX "hdc2" for KEY "hdc2", the latter of which is precisely [ Local_Hekdate_Clock_Yyyymmdd_24_Hhmm_ ]. Since there is a ground Is_In_Range satisfaction claim using this FIELD and no other ground FIELD satisfaction claims that could be candidates for indexed retrieval, Daytona will in fact implement this query so that it uses indexed range retrieval.

When indexed range retrieval is used, the query proceeds by using the B-tree to find the first, i.e., the least, key value that is greater than or equal to the first element in the INTERVAL. Assuming such a value exists and is less than or equal to the upper bound on the INTERVAL, if any, then that value corresponds to a record that satisfies the constraint, which is then retrieved from disk to be subject to further query processing. Then retrieval continues by asking the B-tree to produce the next key value in its ordering which prompts the same kind of query processing as before if the value satisfies the upper INTERVAL bound constraint, if any. This process repeats itself until either the B-tree runs out of key values in the ordering or until a key value is produced which does not satisfy the upper INTERVAL bound constraint, if any. In this way, all the records in the table that satisfy the INTERVAL constraint are produced and no others. Note how efficient this is compared to the box-of-key-values method in that there cannot be any non-productive searches.

(It should be clear now that a by argument to the INTERVAL for Is_In_Range doesn’t make sense (and so is not allowed): in the case of a generator, it is the B-tree that is supposed to be doing the generating, not the by argument, whereas by arguments are ignored for INTERVAL tests, which become just inequality tests on the endpoints.)

By the way, for convenience, Daytona range queries can be specified with INTERVALS that are half-open on the right as in (isin_range.2.Q):
set .lhc1 = "3:44am"CLOCK;
fet (.gh, .nbr, .lhc)
list( there_isa SMORGAS3 where(
    Local_Heclock Is_In_Range [ .lhc1 -> "5:00pm"CLOCK ]
    and Local_Hekclock = .lhc
    and Global_Heka = .gh
    and Number = .nbr )
)
   do Write_Words( .gh, .nbr, .lhc );
}

(Half-open INTERVALs correspond to a less-then constraint as opposed to a less-than-or-equal constraint.) Indeed, the user is also welcome to use INTERVALS in range queries that have no right end-point at all as in:

[ "23:00"CLOCK -> )

Not every type currently supports range queries: currently, the only types that do are: DATE, CLOCK, DATE_CLOCK, UINT(_long_), UINT(_huge_), IP(?), IPNET(?), IPORT2, IPORT6, STRING, and HEKA and any HEKA counterparts of the preceding types. For STRING and HEKA, the ordering is the C lexicographic ordering. The cases for UINT(_long_) / UINT(_huge_) are special because in distinction to the others, those two require a special indexing directive; this is discussed in the subsection following this one. All these possibilities can be seen by the import for Is_In_Range in sys.env.cy:

// The INT is for _right_, the absence of a right endpoint
overloaded PRED: Is_In_Range[
    with_arg_1 STR(=)|HEKA(=)|DATE|CLOCK|DATE_CLOCK|UINT(_long_)|UINT(_huge_)
    |IP(?)|IPNET(?)|IPORT2|IPORT6,
    with_arg_2 INTERVAL[
        STR(=)|HEKA(=)|DATE|CLOCK|DATE_CLOCK|UINT(_long_)|UINT(_huge_)
    |IP(?)|IPNET(?)|IPORT2|IPORT6 .lower,
    STR(=)|HEKA(=)|DATE|CLOCK|DATE_CLOCK|UINT(_long_)|UINT(_huge_)
    |IP(?)|IPNET(?)|IPORT2|IPORT6|INT .upper ]

Indexed range queries are supported for the partial use of multiple FIELD KEYS with Eq predicates leading up to the use of Is_In_Range. Here is an example (isin_range.5.Q):
for_each_time [ .seq_nbr, .dh, .dest_ip, .pkts ]
  is_such_that(
    there_is_a NETFLOW keyed_for_index ^sckt^ where(
      Src_Addr = "192.168.10.4"IP2 and
      Src_Port = 3000 and
      Dest_Addr Is_In_Range [ ^10.10.10.0"IP2 -> ^10.10.10.1"IP2 ] and
      Dest_Addr = .dest_ip and
      Date_Hour = .dh and
      Pkts_Sent = .pkts and
      Seq_Nbr = .seq_nbr )
  ){ do Write_Words( .seq_nbr, .dh, .dest_ip, .pkts ); }

The use of *keyed_for_index* obliges Daytona to use the KEY:

[ Src_Addr, Src_Port, Dest_Addr, Dest_Port ]

This query provides ground equality tests for the first two KEY FIELDS and then uses Is_In_Range on the third FIELD (Dest_Addr) and completely ignores the fourth KEY FIELD (Dest_Port).

It is important to keep in mind that the INTERVALs used for indexed range queries will never catch any Default_Value associated with a FIELD (whose values are allowed to be missing in the data file). Furthermore making the Default_Value the lower bound of the INTERVAL will, in general, lead to meaningless and untrustworthy results. So, don’t do that.

Currently, FIELDSs that use Filter_Funs do not support indexed range queries.

### 13.2.2.1 Indexed Range Queries For UINT

Indexed range queries for UINT require B-trees that are especially constructed for this purpose; in other words, they are different from those that are created by default for handling the UINT types. Any KEY that contains a UINT FIELD and that has a B-tree INDEX that is to be used with Is_In_Range in a Cymal query must have an <Is_In_Range, UINT> note in that INDEX node in the rcd. Here is what one looks like in situ from rcd.CORDER_MD:

```markdown
#{ KEY ( rul )
  <Fields, [ Number, Part_Nbr ]>
  #{ INDICES
    #{ INDEX ( rul )
      <Kind, btree>
      <Is_In_Range, UINT>
    }# }# }#
```

Due to the nature of the <Is_In_Range, UINT> INDEXes, the user must confine their FILE delimiters to the same set required by the use of HEKA-type FIELDS:

```markdown
% # : | [] {} 
```

One can use this <Is_In_Range, UINT> feature to support indexed range queries on FLOATs and
types that are like FLOATs, meaning TIME and MONEY currently, as long as there are no negative quantities. The idea is to store the (positive) FLOAT, TIME, or MONEY as a UINT and then to use an <Is_In_Range, UINT> INDEX to search on it. When there is no fractional part of the FLT values, then clearly the associated UINT is the integer part of the FLT value. If there is a fractional part that the user wishes to keep in play, then the user first needs to decide how many decimal places of the FLT values to keep. Next, the user must shift the decimal point of the indexable FLT values to the right for that number of places and store the associated integer parts as the values of the associated UINT FIELD. Then, to do an indexed range query in this setting, a query would use a construction like this, when shifting 3 places:

```c
and My_Uint_For_Flt_Field Is_In_Range
    [ (UINT)(1000 * .beg_flt) -> (UINT)(1000 * .end_flt) ]
```

### 13.2.3 Indexed Subnet Queries

By means of Is_In_Subnet, Cymbal offers special syntax for asserting that an IP address is in a given subnet:

```c
^192.168.1.1^IP Is_In_Subnet [ .ip, ^255.255.240.0^IP ]
^192.168.1.1^IP2 Is_In_Subnet [ .ip2, 23 ]
.ipv6 Is_In_Subnet [ ^3FFE:D0BA::^IP6, 24 ]
.ipv6 Is_In_Subnet ^3FFE:D3BA::/24^IPNET6
.ipv6 Is_In_Subnet .ipnet6
```

Clearly, an IP2 subnet can be specified here by a TUPLE whose first component is an IP address and whose second component is a netmask specification consisting of either an IP address or a number of masked bits. Alternatively, the subnet can be specified by an IPNET2 or IPNET6 object as in a VBL dereference.

As a special case of indexed range query, indexed retrieval is supported for Is_In_Subnet for both IP(_uint_) and IP6 addresses, but not for IP(_heko_):

```c
for_each_time ( 
    there_isa SMORGAS4 where( 
        Local_Ip_Uint Is_In_Subnet [ ^135.205.95.0^IP2, 24 ]
    )
  )
  with_format _packet_ where( this_isa SMORGAS4 ) do Describe;
```

Since [ Local_Ip_Uint ] is a KEY with a B-tree INDEX and no other possibilities present themselves, Daytona will choose to use that INDEX to run through all the data records whose Local_Ip_Uint values are members of the indicated subnet.

Here is how to cause Daytona to use an indexed subnet query on each of several subnets (isin_subnet.1.Q):

Copyright © 2013 AT&T
All Rights Reserved.
September 15, 2013
Just as with indexed range queries, indexed subnet queries are supported for the partial use of multiple FIELD KEYS with Eq predicates leading up to the use of Is_In_Subnet. Here is an example (isin_subnet.1.Q):

```plaintext
for_each_time [ .seq_nbr, .dh, .dest_ip, .pkts ]
  is_such_that(
    there_isa NETFLOW keyed_for_index ^sckt^ where(
      Src_Addr = ^192.168.10.4^IP2 and
      Src_Port = 3000 and
      Dest_Addr Is_In_Subnet [ ^10.10.10.0^IP2, 31 ] and
      Dest_Addr = .dest_ip and
      Date_Hour = .dh and
      Pkts_Sent = .pkts and
      Seq_Nbr = .seq_nbr )
  )
  do Write_Words( .seq_nbr, .dh, .dest_ip, .pkts )
  )
```

As before, the use of keyed_for_index obliges Daytona to use the KEY:

```
[ Src_Addr, Src_Port, Dest_Addr, Dest_Port ]
```

Here is an example showing the use of both IPNET6 but also a VBL dereference being used to convey the necessary IPNET6 subnet (ipnet_v6.3.Q).

```plaintext
local: IPNET6 .sn6
set .sn6 = [ ^3FFE:D0BA::^IP6, 24 ];
do Write_Words(
  aggregates( of [ count( ), avg( over .ps ) ]
    each_time(
      there_isa NETFLOWED2 where(
        Src_Addr Is_In_Subnet .sn6
        and Pkts_Sent = .ps )
    )));
```
It is important to keep in mind that the subnets used for Is_In_Subnet queries will never catch any Default_Value associated with a FIELD (whose values are allowed to be missing in the data file).

### 13.2.4 Hash Joins

The joins illustrated towards the end of Chapter 9 are called nested loop joins. There is a variety of joins, including what are called hash joins. Hash joins can also be expressed in Cymbal. To compare and contrast, consider these next examples showing equivalent nested loop and hash joins.

First, a nested loop join:

```cymbal
in_lexico_order
with_format _table_
do Display each[ .supplier, .city, .onbr, .qty ]
each_time(
 there_isa SUPPLIER where(
 Number = .sno and Name = .supp and
 City = .city which Is_In [ "Anaheim", "Fairfield" ]
 and
 there_isa ORDER where( Supp_Nbr = .sno
 and Number = .onbr and Quantity = .qty )
);```

What makes this a join is the occurrence of .sno in both Cymbal descriptions. The reason why it is called a nested loop join is because Daytona implements it by processing each SUPPLIER record so as to generate values for sno (and supp and city) and then for each of those .sno, it processes all the ORDER records that have Supp_Nbr = .sno. Note that since SUPPLIER has an index on City which can be used with box-of-key-field-values, that index is used to directly access only those SUPPLIER records that satisfy the City constraint. Furthermore, since ORDER has an index on Supp_Nbr and the occurrence of .sno in the ORDER description is ground, that index will be used to constrain the search to only those ORDER records with the necessary Supp_Nbrs. If there were no such index, then the ORDER table would be sequentially scanned. It is the use of ORDER’s Supp_Nbr index that makes this an indexed nested loop join.

Which processing strategy is faster, an indexed search of ORDER or a sequential scan? It depends on what fraction of the ORDER table satisfies the index constraint. If it is a sufficiently large fraction (and more than 20% is certainly large enough for that), then the time it takes disk arms to locate indexed records (which is measured in a few milliseconds per seek) will accumulate to much more than it would take the query to sequentially access the data (via disk read-ahead). The former puts the work on the disks and the latter on the CPU.

Note that an indexed nested loop is almost always faster than a nested loop join that does not use an index. In the latter case, in effect, the Cartesian product of the two tables is being computed and that has the quadratic complexity $\Theta(mn)$ where $m$ and $n$ are the numbers of records in the outer and inner tables respectively. For the purposes of this discussion, $\Theta(m)$ means that for $m$ large enough, the complexity is essentially $Cm$ for some constant $C$, meaning that it is bounded between $C_1m$ and $C_2m$ for some constants $C_i$. On the other hand, the complexity for an indexed nested loop join is $\Theta(m) + \Theta(n)$. 

Copyright © 2013 AT&T
All Rights Reserved.
September 15, 2013
\( \Theta(m \log n) \) which is at least asymptotically significantly smaller (and amounts to be \( \Theta(m \log n) \)).

Is it better/faster to put the smaller table first in a nested loop join? The answer is not necessarily. To see this, suppose \( m = 2^k \) and \( n = 2'm \) for any fixed \( i \geq 1 \) and then observe that as \( k \to \infty \),

\[
\frac{C_1 2^k + C_2 (k+i)2^k}{D_1 2^{k+i} + D_2 k2^{k+i}} = \frac{C_1 + C_2 (k+i)}{2^i(D_1 + D_2 k)} = \frac{C_2}{2^iD_2}
\]

So, while the smaller-table-first version of the query is faster if \( C_2 = D_2 \) and is more likely to be faster the bigger \( i \) is, the answer in general is that it depends and indeed, in the subsequent performance section, it will be seen by example that putting the smaller table first can well be the worst -- or the best.

Here is the equivalent query using a hash join (hashjoin.2.Q):

```sql
in_lexico_order
with_format _table_
do Display each[.supplier, .city, .onbr, .qty ]
each_time(
    // cache only what the query needs in the in-memory dynara
    .supplies = { .sno => [ .supp, .city ] :
        there_isa SUPPLIER where(
            // This hash join uses the index on City; could require a sequential scan though
            Number = .sno and Name = .supp and
            City = .city which Is_In [ "Anaheim", "Fairfield" ] )
    }

    // sequential scan of the inner table.
    and there_isa ORDER where( Supp_Nbr = .sno
        and Number = .onbr and Quantity = .qty )
    // the join constraint
    and [ .supplier, .city ] = .supplies[ .sno ]
);
```

The idea behind a hash join is to visit records from the first or outer table of the join (either sequentially or by B-tree index) and store TUPLEs of values to join the inner table with an in-memory hash table (dynara), along with any other FIELD values from those outer table records needed by the query. Then the second and last step is to visit records from the inner table (either sequentially or by B-tree index!) and to test them for inclusion in the join by seeing if their join values are a key to the hash table, in which case, all the information from the pair of joined tuples is made available to the query. The nature of the dynara .supplies is to map join key values to relevant FIELD values from the outer table. Note that the dynara .supplies is declaratively defined above; it could have been defined instead procedurally before the Display call. An important advantage of a declarative definition is that it enables hash joins to be utilized in views.

Note that the computation that is intended to be faster by using the hash join instead of indexed nested loops join is the matching up of the join record pairs. When not using indices, the complexity of hash joins is \( \Theta(m) + \Theta(n) \) which is asymptotically significantly less than \( \Theta(m \log n) \). To get an idea as to why this might be true, let \( m = 2^k \) and \( n = 2'm \) for fixed \( i \) and then the hash join complexity is
versus $\Theta((k + i)2^k)$ for nested loops. So, the hash join will at least eventually become better as $k$
 grows, while of course in the meantime, the actual complexity constants make all the difference. Most
textbook discussions on hash joins assume that both tables are being scanned sequentially, but as
remarked above, there is no necessity for that as indices can be used (or not) for both tables. Note than
when indices are used on the tables individually, then the cost of finding the records in the outer table
is $\Theta(m_k \log m) + \Theta(A_k m_k)$ where $m_k$ is the number of keys being looked up and $A_k$ is the average
reach for this index and, for indexed nested loop, the number of records to look up in the inner table is
$\Theta(A_k m_k)$ and so the complexity of working with the inner table is $\Theta(A_k m_k \log n)$.

An advantage of hash joins is that they do not require that the inner table have a B-tree built
(previously) on the join FIELDS. A serious disadvantage is that if the hash table (dynara) does not fit
into memory or fits so poorly that it is getting paged out all the time, then the query can die or run
very slowly. In contrast, indexed nested loop joins never, ever run out of memory per se, although, of
course, like all queries, they can run afoot of other problems. Also, since hash joins clearly rely on
Cymbal for expression, the only way to use them in DSQL is implicitly by means of a view.

Note that this particular order of SUPPLIER on the outside and ORDER on the inside reflects the
one-to-many relationship between SUPPLIER and ORDER. One might briefly entertain the notion of
reversing the order and putting ORDER on the outside with SUPPLIER on the inside. This is not a
good idea. The reason is that the outer table is visited just once and at that time, everything that the
query needs from that table is squired away in the dynara for lookup (on the join-key Supp_Nbr)
when visiting the inner table. This would entail creating a mapping from Supp_Nbr to a LIST of
TUPLES of FIELD values taken from ORDER. Not only is this a more complicated data structure than
the simple Supp_Nbr to single TUPLE of SUPPLIER FIELD values used in the one-to-many case, but it
is simply storing much more information in the hash table in the form of many more TUPLES; after all,
many ORDER records correspond to one SUPPLIER record, as a rule. So, the moral of the story is to
always do these joins one-to-many (or one-to-one).

But what about many-to-many relationships? Often a many-to-many relationship, say from ABC
records to XYZ records, is represented by having a middleman/link table ABC_XYZ that has attributes
about the relationship and which is such that both ABC and XYZ have a one-to-many relationship with
ABC_XYZ. This is the case for SUPPLIER, PART, and ORDER in the Daytona test suite. The following
query shows how to hash-join SUPPLIER to PART via their linking table ORDER (hashjoin.3.Q). Note
that for each SUPPLIER record, there are many PARTs and vice-versa.
in_lexico_order
with_format _table_
display each[ .supplier, .city, .onbr, .qty, .part ]
each_time(
  // cache only what is needed in the in-memory dynaras
  .supplies = { .sno => [ .supp, .city ] :
    there_isa SUPPLIER where(
      Number = .sno and Name = .supp and
      City = .city which Matches "Anaheim" | Matches "Fairfield" ) }
  and
  .parties = { .pno => [ .part ] :
    there_isa PART_ where( Number = .pno and Name = .part ) }
  // sequential scan of the other table.
  and there_isa ORDER where( Supp_Nbr = .sno and
    Number = .onbr and Quantity = .qty and Part_Nbr = .pno )
  and [ .supplier, .city ] = .supplies[ .sno ]
  and [ .part ] = .parties[ .pno ]
);

Note that relevant information from the SUPPLIER and PART tables is cached in two dynara for subsequent use when visiting the ORDER table.

As illustrated in this next query, also part of hashjoin.3.Q, a hash join and an indexed nested loops join can be used together to accomplish a many-to-many join in a group-by setting.
// group-by using both hash join and indexed nested loop
in_lexico_order
with_col_labels [ "Supplier", "Part", "Tot_Qty_Ordered" ]
with_format _table_
do Display
each_tuple_of { [ .supplier, .part, sum( over .qty ) ] :
// cache only what is needed in the in-memory dynara
  .supplies = { .sno => [ .supp, .city ] :
    there_isa SUPPLIER where(
      Number = .sno and Name = .supp and
      City = .city which Is_In [ "Anaheim", "Fairfield" ] ) } 

// INDEXED scan of the other table
and there_isa ORDER where(
  Supp_Nbr = .sno and Number = .onbr
  and Date_Placed Is_In_Range [ ^1983-09-07 ^ -> ]
  and Quantity = .qty and Part_Nbr = .pno )
and [ .supplier, .city ] = .supplies[ .sno ]
// indexed nested loops (saves memory over doing a hash)
and there_isa PART_ where( Number = .pno and Name = .part )
};

Here the query produces all SUPPLIER/PART pairs along with the total quantity ordered for each such combination.

A (not-as-bad-as-it-sounds) drawback with Cymbal hash-joins is that it is neither easy to express nor efficient to process a chain of them, i.e., where $k$ tables are being joined and the result of the first $i$ hash-joins is being hash-joined to the next table. It can be written though and done but it is not pretty. Here is such a version of a previous, much better written query (hashjoin.3.Q):
in_lexico_order
with_format _table_
do Display each[ .supplier, .city, .onbr, .qty, .part ]
each_time(
  .supp_ord = [ .pno => [ .onbr, .supplier, .city, .qty ] :
    .supplies = [ .sno => [ .supplier, .city ] :
      there_isa SUPPLIER where(
        Number = .sno and Name = .supplier
        and City = .city which Matches "Anaheim" | Matches "Fairfield" ) }
    and there_isa ORDER where(
      Supp_Nbr = .sno and Number = .onbr and Quantity = .qty and Part_Nbr = .pno )
    and .supplies[.sno] = [ .supplier, .city ]
  }
  and there_isa PART_ where( Number = .pno and Name = .part )
  and .supp_ord[.pno ] = [ .onbr, .supplier, .city, .qty ]
);
ORDERs for SUPPLIER 600.

This poses a problem though because, since there are no ORDERs for SUPPLIER 600, there is also no value for ORDER.Quantity -- and yet one is needed in order to construct an answer if ORDER.Quantity is to be part of any answer. Daytona solves this problem by first realizing that what is happening in these situations is that some of the ORDER records being sought are missing (i.e., absent) and that, when that happens, the query needs to provide some dummy value to report instead as the associated Quantity. So, by introducing and supporting but_if_absent for RECORDs, one gets:

\[
\text{there\_isa\ SUPPLIER\ where}\ (\ Name = .\ supplier\ and\ Number = .\ sn )
\text{and\ there\_isa\ ORDER\ where}\ (\ Supp\_Nbr = .\ sn\ and\ Quantity = .\ qty )
\text{but\_if\_absent}( .\ qty = -1 )
\]

Now, when there is no ORDER RECORD for a given SUPPLIER, the corresponding answer will consist of the SUPPLIER information plus −1 for ORDER.Quantity. This is consistent with Cymbal semantics in that the combination there\_isa\-but\_if\_absent is essentially an if-then-else which itself is essentially a disjunction. And the rule for disjunction is that the set of VBLs that are defined in one disjunct must also be defined in all of the disjuncts. In this case, that set contains only the VBL qty. The output of a Display query whose assertion is the one immediately above would contain these lines:

<table>
<thead>
<tr>
<th>Supplier</th>
<th>Qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barbary AG</td>
<td>4551</td>
</tr>
<tr>
<td>Barbary AG</td>
<td>4752</td>
</tr>
<tr>
<td>Standard AG</td>
<td>3383</td>
</tr>
<tr>
<td>Standard AG</td>
<td>3655</td>
</tr>
<tr>
<td>Apex AG</td>
<td>-1</td>
</tr>
</tbody>
</table>

The −1 is the user-selected "dummy value" for Quantity when there is no ORDER RECORD corresponding to a given SUPPLIER RECORD. In the literature, a join like this is called a left outer join. The word outer is used because the join result contains information about records that don’t actually join with the other table and the word left is used because those records belong to the table on the left, in the sense of being the first table mentioned -- which, in SQL, really is on the left. (After discussing how outer joins are expressed (with more generality) in Cymbal, the discussion here will turn to how they can appear in DSQL using largely standard SQL syntax even though DSQL/Cymbal has no null values.)

Here is a three-way join that illustrates the generality of left outer joins in Cymbal (leftouterjoin.2.Q #3):
set .cutoff = 3000;
    with_format _table_
do Display each [ .supplier, .qty, .part, .cap_color ]
each_time(
    
    there_is_a SUPPLIER where(
        // Apex/502 has no ORDER
        Name = .supplier and Number = .sn which_is >= 500 )
        // don’t need to reference a FIELD value
        or ( .supplier = "Nosupp" and .sn = 555 )
    )
    and there_isa ORDER where(
        Supp_Nbr = .sn and Part_Nbr = .pno
        and Quantity = .qty which_is > .cutoff )
    // these user-provided "dummy" values cannot appear
    // in indexed fields used for outer join
    but_if_absent( [ .pno, .qty ] = [ -1, -1 ] )
    // .sn = 555 leads to .pno = -1 leads to no-part
    and there_isa PART where(
        Number = .pno and Name = .part
        and Color Matches "e"
    // an output value needing a dummy can be a function of a FIELD value
    and Color = .color
        where( .cap_color = upper_of(.color) )
    )
    but_if_absent( .part = "no-part"
        and .color = "no-color" and .cap_color = "NO-COLOR" )
    );

Here is the associated output:

<table>
<thead>
<tr>
<th>Supplier</th>
<th>Qty</th>
<th>Part</th>
<th>Cap_Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barbary AG</td>
<td>4551</td>
<td>wheel</td>
<td>ORANGE</td>
</tr>
<tr>
<td>Barbary AG</td>
<td>4752</td>
<td>drum brakes</td>
<td>PEACH</td>
</tr>
<tr>
<td>Standard AG</td>
<td>3383</td>
<td>jack</td>
<td>BLUE</td>
</tr>
<tr>
<td>Standard AG</td>
<td>3655</td>
<td>magnetic tape</td>
<td>CLEAR</td>
</tr>
<tr>
<td>Apex AG</td>
<td>-1</td>
<td>no-part</td>
<td>NO-COLOR</td>
</tr>
<tr>
<td>Nosupp</td>
<td>-1</td>
<td>no-part</td>
<td>NO-COLOR</td>
</tr>
</tbody>
</table>

If this ORDER table violated the foreign key integrity constraint that requires all of its Part_Nbrs to have corresponding entries in the PART table, then it would be possible for there to be an output line where Quantity != -1 but yet part and cap_color have their user-specified dummy values.

Anyway, this query illustrates a number of things. First, a there_isa with a but_if_absent like the
one for ORDER above doesn’t need to be preceded by a there _isa_. It just needs to be preceded by an assertion that is generating values for the KEY being used to access that there _isa_ as would be the case for a conjunct the enumerating (KEY) elements of a BOX.

In fact, but _if_absent_ can only be used with there _isa_ WIDGET in a query if Daytona will be using a WIDGET KEY to implement access to that there _isa_. This implies that dummy values should not appear as FIELD values for FIELDS that are in KEYS that will be used to work with a there _isa_ with a but _if_absent_. For example, the dummy value, i.e., −1, chosen above for .pno in handling there _isa_ ORDER must not appear as an actual/valid PART.Number KEY value because then a spurious answer will be generated, spurious because there should not be a PART associated with a (completely) missing ORDER. So, it is incumbent upon the user to choose dummy values that satisfy this restriction.

Note also that dummies are not necessarily for FIELD values per se. In the example above, a dummy value is needed for the computed quantity .cap _color_ simply because it is defined in the there _isa_ ORDER but note that cap _color_ is not a FIELD value itself (nor actually is it required to be associated with any FIELD).

### 13.2.6 Outer Joins: System-supplied Dummy Values

Now so far, these left outer join queries are requiring the user to explicitly provide dummy values as needed. This can certainly become tedious if dozens of FIELDS will be generating output and need dummies. Fortunately, Cymbal provides dummy values that can be used instead of user-specified ones. These are referred to in queries just by using the constant _dummy_ in those places where a user-specified dummy value is required. Daytona interprets _dummy_ as a system-specified value determined according to the corresponding type. For example, here is the definition of INT(_long_) from $DS_DIR/sys.env.cy:

```plaintext
define CLASS INT(_long_)  
  with_c_type 'Int32'  
  with_cy_init_val '0'  
  with_min_val -2147483647  
  with_max_val 2147483647  
  with_dummy_val -2147483646
```

So, the user may choose to replace this:

```plaintext
but_if_absent( [ .x, .y ] = [ -1, "N/A" ] )
```

with either of these two

```plaintext
but_if_absent( [ .x, .y ] = [ _dummy_, _dummy_ ] )
but_if_absent( [ .x, .y ] = [ _dummies_ ] )
```

where it is understood that Daytona will expand _dummies_ into the appropriate number of _dummy_ constants. So, in this example, if x is an INT VBL, its _dummy_ will be taken to be −2147483646 and if y is a STR VBL, then its _dummy_ will be taken to be "dummy-STR" as specified in $DS_DIR/sys.env.cy.

So, how have _dummy_ values been chosen for Cymbal types? First of all, they have to be (legitimate) members of the type; second, they are chosen so that they are unlikely to appear in user
data (and hence be indexed); third, they are chosen so that they do not have additional semantics as they would, for example, if they were the min, max, or the initial value for the type. As illustrated by BOOL, whose _dummy_ value is _false_, achieving all three goals is not always possible. Furthermore, note that BITSEQ does not have a (system-provided) _dummy_ value because none seemed to be universally applicable; consequently, any particular user/application needs to pick their own BITSEQ value for this purpose.

Note that a _dummy_ is not a Cymbal missing value because the phrase "Cymbal missing value" does not refer to a value, it refers to a situation where a value is possible but simply does not exist. Furthermore, a _dummy_ could be a Default Value for FIELD but that Default Value should probably be chosen so that it is not equal to the _dummy_ for the FIELD’s type so as to prevent the spurious answers referred to above due to recklessly indexing on dummy values. At any rate, the point is that they have different semantics, with _dummy_ being intended only for supporting outer joins. Also, note that a _dummy_ is not a NULL value -- because Cymbal does not have NULL values.

Since any _dummy_ is a legitimate member of its type and may appear in query answers, without further ado, it will be printed in output in the way that it should be for a member of the type: so, an INT(_long_) _dummy_ will be printed as −2147483646. However, one may wish to print something else instead that is more suggestive of its dummy nature like "N/A" or perhaps just blank space. This is most easily accomplished by using the if dummy then function as illustrated by (leftouterjoin.2.Q):

```plaintext
with_format _table_
  with_col_labels [ "Part", "Color", "Supplier", "Qty" ]
do Display
  each [.part, .color,
    if_dummy_then(.supplier,"-none-"), if_dummy_then(.qty,"") ]
each_time{
  ( there_isa PARTED where(
    Number = .pno and Name = .part
    and Color = .color which Matches "e"
  )
or there_isa PARTED where(
    Number = .pno and Name = .part
    and Color Is _absent_ where( .color = "sometime_soon" )
  )
  )
and there_isa ORDER where(
    Supp_Nbr = .sn and Part_Nbr = .pno
    and Quantity = .qty which_is > 3000
  )
  but_if_absent( [ _dummies_ ] = [ .sn, .qty ]
and there_is_a SUPPLIER where(
    Name = .supplier and Number = .sn which_is >= 500
  )
  but_if_absent( [ _dummies_ ] = [ .supplier ]
  )
);```

The resulting output is:
as opposed to if just .supplier and .qty were used instead of calling if_dummy_then:

```
<table>
<thead>
<tr>
<th>Part</th>
<th>Color</th>
<th>Supplier</th>
<th>Qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>wheel</td>
<td>orange</td>
<td>Barbary AG</td>
<td>4551</td>
</tr>
<tr>
<td>jack</td>
<td>blue</td>
<td>Standard AG</td>
<td>3383</td>
</tr>
<tr>
<td>magnetic tape</td>
<td>clear</td>
<td>Standard AG</td>
<td>3655</td>
</tr>
<tr>
<td>drum brakes</td>
<td>peach</td>
<td>Barbary AG</td>
<td>4752</td>
</tr>
<tr>
<td>waterclock</td>
<td>yellow</td>
<td>dummy-STR</td>
<td>-32766</td>
</tr>
<tr>
<td>timepiece</td>
<td>sometime_soon</td>
<td>dummy-STR</td>
<td>-32766</td>
</tr>
<tr>
<td>montre</td>
<td>sometime_soon</td>
<td>dummy-STR</td>
<td>-32766</td>
</tr>
</tbody>
</table>
```

The semantics of if_dummy_then should be clear: if its first argument is the system-defined _dummy_ value for that first argument, then the STR second argument is the result of the call to if_dummy_then; else the result is the (STR) cast of the first argument. If it weren’t implemented as a ds_m4 macro, it would have a Cymbal declaration of:

```
STR FUN( OBJ, STR ) FUN if_dummy_then
```

Two other fpps offer help in working with _dummy_. type_dummy returns the _dummy_ value associated with the type offered as its sole argument. Isa_Dummy is a PRED which is _true_ if and only if its sole argument is a _dummy_.

```
when( type_dummy( `IPNET(_uint_)` ) Isa_Dummy
    and type_dummy( `IPNET` ) Isa_Dummy
    and type_dummy( `IPNET6` ) Isa_Dummy
    and type_dummy( `DC` ) Isa_Dummy )
{
    do Write_Line( "Success!" );
}
```

There is a right outer join which is just the left outer join of the two tables in the reverse order. And there is the (full) outer join which is the (set-)union of the left and right outer joins. Here is a full outer join (leftouterjoin.3.Q):
It seems in practice that the most commonly desired outer join is the left outer join; the right and full outer joins can be derived from the left outer join.

13.2.7 Outer Joins: DSQL and hash join

Fortunately, the presence of system-defined \_dummy\_ values enables the syntax for SQL left outer joins to go over as is from the standard (leftouterjoin.2.Q):
select S.Name, O.Number, O.Quantity, P.Color
from
  ( SUPPLIER as S left outer join "ORDER" as O on S.Number = O.Supp_Nbr )
  left join PART as P on O.Part_Nbr = P.Number
where S.Number >= 500
  and (O.Quantity > .cutoff or O.Quantity = 1203 or O.Quantity = 706)
  and P.Color Matches "e[w".

Note that the outer keyword is optional.

Recalling that DSQL has no NULL values to print out, any type-specific values for _dummy_ will be printed in any output unless the user explicitly arranges for their own choice to be printed out. This can be done analogous to the way it is done in Cymbal by using either Isa_Dummy or sql_if_dummy_then (noting the sql_ prefix here for DSQL):

```sql
set .cutoff = 3000;
select S.Name,
  case when Isa_Dummy[O.Number] then "" else (:STR:)O.Number end
  as Number,
  case when Isa_Dummy[O.Quantity] then "" else (:STR:)O.Quantity end
  as Quantity
from SUPPLIER as S left join "ORDER" as O on S.Number = O.Supp_Nbr
where S.Number >= 500 and O.Quantity > .cutoff
;
select S.Name,
  sql_if_dummy_then(O.Number,""),
  sql_if_dummy_then(O.Quantity,"")
from SUPPLIER as S left join "ORDER" as O on S.Number = O.Supp_Nbr
where S.Number >= 500 and O.Quantity > .cutoff
```

The latter is clearly preferrable (and Daytona actually expands it into the former). leftouterjoin.3.Q also has an example of a full outer join in DSQL. If it weren’t implemented as a ds_m4 macro, sql_if_dummy_then would have a Cymbal declaration of:

```c
STR FUN( OBJ, STR ) FUN sql_if_dummy_then
```

One may wonder if one can do a left outer hash join (in Cymbal). The answer is yes (leftouterjoin.3.Q):
do Display each[ .supplier, .city, .onbr, .qty ]
each_time(
    .supplies = { .sno => [ .supp, .city ] :
        there_isa SUPPLIER where(
            Number = .sno and Name = .supp and
            City = .city which Is_In [ "Anaheim", "Fairfield" ] ) } and there_isa ORDER where( Supp_Nbr = .sno and Number = .onbr and Quantity = .qty )
    // the left outer hash join constraint
    and if( .supplies[ .sno ] = ? )
        then( [ .supplier, .city ] = .supplies[ .sno ] )
    else( [ .supplier, .city ] = [ "N/A", "N/A" ] and .onbr < 40 )
);

Note how the dummy value specification goes further and restricts the output of the dummies.

13.2.8 Instructive Performance Tests

Performance tests comparing hash and indexed nested loops joins were performed on some movie review data where there were 17,770 movies (557K) and 100,480,507 reviews (2.5GB). There were two basic scenarios where the first utilized a Review table that had been sorted on Movie_Id and the second utilized the same data but sorted on the Reviewer_Id. In these tests, the second Review table will be called ill-sorted because while sorted by Reviewer_Id, it is not sorted by Movie_Id. All the joining is done on Movie_Id. Any B-tree index that is used is a vanilla unclustered one EXCEPT for the Movie_Id B-tree for the Review table when sorted by Movie_Id which is a cluster B-tree. With indexed nested loops, one can choose whether to go from the small Movie table to the large Review table (sorted or ill-sorted) using a (non-unique) cluster or vanilla B-tree on Movie_Id (resp.) or go from the large table (sorted or ill-sorted) to the small using a (unique) B-tree on Movie_Id.

Here are two of the queries, with the others being straightforward variants:
// indexed nested loop join
select MOVIE.Name, count(*), rtn( avg(Rating), .001 )
from MOVIE, REVIEW
where MOVIE.Id = REVIEW.Movie_Id
group by MOVIE.Name
order by MOVIE.Name;

// hash join
sorted_by_spec [ 1 ]
do Display each_tuple_of
( [ .movie, count(), rtn( avg( over .rating ), .001 ) ]:
  .movie_info = { .movie_id => .nm :
      there_isa MOVIE where( Id = .movie_id and Name = .nm ) }  
  and
  there_isa REVIEW where( Movie_Id = .m_id and Rating = .rating )
  and .movie_info[ .m_id ] = .movie
);

Here are the "wall-clock" execution times where each is the median of three tests and where no
group of three overlaps another:

<table>
<thead>
<tr>
<th>Method</th>
<th>Sorted</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>idx nested loop</td>
<td>small</td>
<td>1m51s</td>
</tr>
<tr>
<td>hash</td>
<td>sorted</td>
<td>2m29s</td>
</tr>
<tr>
<td>hash</td>
<td>ill-sorted</td>
<td>3m41s</td>
</tr>
<tr>
<td>idx nested loop</td>
<td>large</td>
<td>8m1s</td>
</tr>
<tr>
<td>idx nested loop</td>
<td>large</td>
<td>13m50s</td>
</tr>
<tr>
<td>idx nested loop</td>
<td>small</td>
<td>29m54s</td>
</tr>
</tbody>
</table>

Caveat lector: at most, it is the relative sizes of these timings that are informative. The actual values in
minutes and seconds are not at all useful in predicting what users will get for their (different) queries on their
(different) machines. For example, these tests were done on a SPARC machine (in 2012); anyone using x86_64
for these tests would see much better absolute times and perhaps different relative ratios.

Before discussing these results, consider another set of queries. The point of these queries is to
pull information on reviews for the movies Titanic and Ghost. Since it turns out that there are 284116
of them and the output is 9268683 bytes long, this is not a trivial retrieval of a few records, although it
is .002 of the total number of reviews. To prevent indexed nested loops from using its cluster B-tree
advantage, the data used is the ill-sorted data for which regular B-trees are used.

The indexed nested loops query uses the MOVIE B-tree on Name and the REVIEW_BY_R B-tree
on Movie_Id:
in_lexico_order
do Display each[ .title, .year, .customer, .rating, .review_date ]
each_time(
  there_isa MOVIE where( Id = .movie_id and // B-tree used on Name
    Name = .title which Is_In [ "Titanic"STR, "Ghost" ] and Year = .year )
  and // B-tree used on Movie_Id
  there_isa REVIEW_BY_R where( Movie_Id = .movie_id and Rating = .rating
    and Review_Date = .review_date and Customer_Id = .customer )
);

Here is the equivalent hash-join query, which of course uses the MOVIE B-tree on Name as well --
as well it should:

in_lexico_order
do Display each[ .title, .year, .customer, .rating, .review_date ]
each_time(
  .movie_info = { .movie_id => [ .title, .year ] :
    there_isa MOVIE where( Id = .movie_id and Year = .year
      Name = .title which Is_In [ "Titanic"STR, "Ghost" ] ) )
  and
  there_isa REVIEW_BY_R where( Movie_Id = .m_id and Rating = .rating
    and Review_Date = .review_date and Customer_Id = .customer )
  and .movie_info[ .m_id ] = [ .title, .year ]
);

Here are the timings:

  idx nested loop  small_1st  hot_cache   8s
  idx nested loop  small_1st  cold_cache 35s
  hash              hot_cache  1m7s
  hash              cold_cache 1m21s
  idx nested loop  large_1st  cold_cache 13m6s

The speed of B-tree index retrieval is sensitive to the influence of various computing platform caches;
obviously, cached data is retrieved faster than data that has to be read off of disk. Anyway, in these
examples, it is apparent that indexed nested loops can be much better than hash join -- and much
worse: join order can matter a lot for indexed nested loops.

These examples suggest the following observations (but don’t necessarily prove anything in
general):

— The indexed nested loop join which uses the cluster B-tree on Movie_Id for the sorted Review
  table beats both hash join values. This really highlights the utility of sorting towards reducing
  disk seeks and towards minimizing any consequent reading/buffering of irrelevant records. Of
  course, data can only be sorted one way and so accesses by other fields will not in general be
  able to rely on one sort order to speed them up.
— Sometimes hash joins are faster and sometimes they are not, as is the case when the data is
sorted well or not for the group-by queries. It’s hard to say.

— Curiously, sorting makes a difference for hash joins -- as seen in the group-by queries which are
also using another dynara behind the scenes in their group-by calculations.

— In these examples, when indexed nested loops have the wrong join order, their performance is
much worse and much worse than the hash joins.

— These examples actually prove that for indexed nested loop joins sometimes putting the small
table first is faster but sometimes not. The complexity analysis indicates that this is more likely
to be the case the larger the large table is in comparison. However, keep in mind for the
group-by queries that there is so much going on there, that a simple complexity analysis is
likely to be inadequate.

So, the hash join is worth considering because sometimes it will perform much better than indexed
nested loops, especially when the indexed nested loops use the wrong join order.

Chapter 20 discusses parallelizing a hash join as illustrated by para_hashjoin.1xM.1.Q and
para_hashjoin.MxM.1.Q. Indeed, the latter shows another way to handle a different many-to-many
situation. See also the discussion of a parallelized hash join using shared memory in Chapter 21
(para_hashjoin.1xM.3.d.Q).

13.3 Miscellaneous Cymbal Description Capabilities

13.3.1 Total Missing Value Control

Chapter 3 contains a discussion of how missing values are represented in DC data files. This
section discusses the Cymbal description syntax that can be used to gain total query language control
over missing values in the data.

First, consider Daytona’s default behavior regarding missing values as illustrated by the following
query which asks for the Number and Date_Recd for each ORDER with Number <= 10 and Quantity >
1000.:

```
with_format _table_
do Display each [ .number, .date_recd ]
each_time(
   there_is_an ORDER where(
      Number = .number which_is <= 10
      and Date_Recd = .date_recd
      and Quantity > 1000. )
);
```

The output is:
By looking at the ORDER data which comes with Daytona’s sample application, the user will see that even though the ORDER with Number 1 has Quantity 2311 it does not appear in the output: this is because its Date_Recd value is missing. This behavior can be deduced from knowing that Cymbal variables must always have values in order to be used at all. So, for any record with a Date_Recd field that has no value, there can be no value for the variable date_recd that equals nothing (i.e., the absence of a value) and therefore, as a direct consequence of the conjunctive quality of descriptions, this record is skipped over and contributes in no way to the answers of the query, regardless of whether other fields referenced in the query description have values for that record or not. Note that this is consistent with Cymbal’s general missing value philosophy, colloquially expressed as: you can’t work with something that’s not there.

While this missing value philosophy has the advantage of simplicity and predictability, it is at variance with the SQL philosophy which is implemented by having the special out-of-band null value NULL to indicate a missing value, i.e., there is a special something to indicate nothing. (’out-of-band’ means that NULL is not a member of any of the standard types such as INT.) Consequently, SQL implementations are more than willing to print out answer tuples, the components of which may well be NULL. That can never happen with Cymbal, since Cymbal has no ‘null value’.

In order to be able to achieve the ability SQL has for working with records some of whose referenced FIELDS have missing values, Daytona provides the but_if_absent construct. This construct enables users to perform tests and more importantly, to generate new variable values in the event that there is no value for a given FIELD in a given record. In toto then, as illustrated next (whoabs.1.Q), the Cymbal user may work with a FIELD value if it is present, but if it is absent, then other work may be performed. This power is frequently used to generate in-band missing value indicators:
do Display
  with_title_line "first few orders with Date_Recd values"
  each[ .order_number, .prt_date ]
  each_time( 
    there_is_an ORDER where( 
      Number = .order_number which_is < 20 
      and Date_Recd = .date_recd 
      for_which( .prt_date = (STR) .date_recd ) 
      but_if_absent( .prt_date = "-unknown-" ) )
  );

In this query, if a Date_Recd value is _present_ for a given record, then its STR form is printed along with the ORDER Number. But if it is _absent_, then "-unknown-" is printed out in the Date_Recd position along with the ORDER Number. The use of .prt_date is necessary because if .date_recd is said to be "-unknown-" then Daytona will issue a type error saying that somebody is trying to assign a non-DATE to a DATE variable. Anyway, the general principle here is that by using but_if_absent, variables can get the values of fields if those values are there and if not, then they get whatever value the user thinks is appropriate. Or, as the next query will illustrate, the user can set an indicator variable to indicate whether the field value is there or not (whoabs.3.Q). Or one could even do both.

do Display
  with_title_line "orders with missing Date_Recd values"
  each[ .order_number ]
  each_time( 
    there_is_an ORDER where( 
      Number = .order_number 
      and Date_Recd Is _present_ for_which ( .indic = 0 ) 
      but_if_absent ( .indic = 1 ) 
    and .indic = 1
  )

Note the use of Date_Recd Is _present_ to test for the presence of the FIELD value. Absence can be tested similarly.

There are only two restrictions on the use of but_if_absent. First, but_if_absent is treated syntactically like a keyword for an assertion argument and as is always the case for such keyword arguments, the assertion must be in parentheses. Secondly, any variables that appear lexically for the first time in a note with a but_if_absent are being finitely defined on first use there and so, as is the case with any if-then-else, the same variables must be bound in all branches: consequently, if they appear in the but_if_absent part, then they must appear in the other part and vice-versa.
13.3.2 this is a

Advanced users can use the this is a construct to improve execution efficiency by reusing an existing data buffer. The following query is contrived and more complicated than it needs to be but it does illustrate the this is a construct and shows in addition that there is more than one way to skin a cat (whoabs.2.Q).

```c
  do Display
    with_title_line "orders with missing Date_Recd values"
    each[ .order_number ]
    each_time(
      there_is_an ORDER where( Number = .order_number )
      and if( this_is_an ORDER where( Date_Recd Is _present_ )
          then( .indic = 0 )
          else( .indic = 1 )
      and .indic = 1
    )
  
  What this is a does is to reuse the data buffer for the corresponding there is a instead of employing its own data buffer at the cost of an extra independent access. After all, if the data is already in memory, why read it in again? Daytona finds the corresponding there is a by taking the one which has the closest ancestor in the parse tree to the this is a. This is the one that the user will expect on the basis of parenthesis nesting. An error message will result if there is more than one qualifying there is a. Note that it is only necessary to refer to the fields of interest in a this is a; there are many instances where if a there is a is used instead, then it is necessary to include values for key fields in the description in order to ensure that the same record will be accessed again by this other independent access.

this is a, this is an, and this are synonyms.
```

13.3.3 Describe And Dump Output PROCEDURES

The Describe and Dump PROCEDURES use the this is a construct to provide convenient and efficient mechanisms for printing entire data records to some designated I/O CHANNEL. For example, consider the following loop over PERSONI records satisfying an Age constraint (describe.1.Q):

```c
  for_each_time there_is_a PERSONI where( Age > 11 )
  do {
    with_format _packet_ to _stderr_
    where( this_is_a PERSONI ) do Describe;
  }
```

The loop body invokes Describe on each qualifying PERSONI record to produce in the packet format such listings as:
Observe how LIST/SET-valued fields are printed out and that if a field value is missing (as is the case with Roger's Autos) then nothing is printed out.

The most important argument to a `Describe` call is the argument for the keyword where: this argument is a `this isa` assertion for some RECORD_CLASS. As with all `this isa` assertions, there must be a corresponding `there isa` which defines the corresponding set of records of interest. The `to` argument specifies the I/O CHANNEL to send the output to; if absent, it defaults to `_stdout_`. The `with format` argument is the same as it is for `Display`, although at the present time, it can only support the `_packet_`, `_desc_`, `_xml_`, `_safe_`, and `_data_` formats, with `_packet_` being the default.

An obvious convenience provided by `Describe` is that Daytona takes care of identifying all of the FIELDS by name and takes care of printing them out in a suitable format. Daytona also handles `Default Values` appropriately as well as the values for horizontal partitioning FIELDS. Furthermore, the values that `Describe` prints are at the user or tty level meaning that, for example, even if the data is stored using HEKA types, it will be printed out in its human-readable form.

In this way, Daytona is seen to be providing the same service that SQL’s `select *` does for single tables. This is highlighted by the fact that `Describe` will not skip over a record just because some of its field values are missing; instead it just won’t do anything in response to encountering missing field values, i.e., nothing will be output, the I/O cursor will not advance. This is another way to be consistent with Daytona’s general missing value philosophy which is that "you can’t do anything with nothing". Since `Describe's` only function is to print characters (and since it makes no use of any Cymbal variables that could receive FIELD values), it’s quite natural for it to simply not emit any characters in response to encountering a missing field value. So, this is the closest that Daytona comes to providing a `select *`: it still does not have a null value nor will it print "null"; rather it simply does not print anything at all in response to encountering a missing field value but otherwise it processes the containing record as usual.

The analog of what SQL’s `select *` provides for join expressions is illustrated by:
for_each_time
  (there_is_a SUPPLIER where(
    Name = "Barbary AG" | = "Acme Shipping"
    and Number = .snbr )
  and there_is_an ORDER where( Supp_Nbr = .snbr ))
do {
  skipping 1
  do Write_Words( 10*"=", "SUPPLIER-ORDER join record", 10*"=" );
  where( this_isa SUPPLIER) do Describe;
  where( this_is_an ORDER) do Describe;
}

Here is some sample output:

========== SUPPLIER-ORDER join record ==========

SUPPLIER RECORD

Number: 500
Name: ‘Barbary AG’
City: ‘Fairfield’
Telephone: ‘201-923-1288’

ORDER RECORD

Number: 480
Supp_Nbr: 500
Part_Nbr: 103
Date_Recd: ‘6/17/85’
Date_Placed: ‘5/4/83’
Quantity: 4551
Last_Flag: ‘0’

If the user would like to massage this output in some way, they can direct Describe to write its output into a _string_ CHANNEL whereupon it can be further manipulated.

13.3.3.1 Dump PROC

The Dump PROCEDURE provides the fastest way to get the entire contents of a Daytona record printed out in toto and as-is to some I/O CHAN:
for_each_time there_isa PERSONI
   do {
      where this_isa PERSONI do Dump;
   }

What happens here is that Daytona’s record buffer is written out character-by-character, as-is to _stdout_. In particular, no horizontal partitioning FIELD information is added, no Default_Value logic is applied, etc. What the user gets is exactly what they had in the data file.

13.3.4 Safe Dirty Reads

(Read-only) queries that are not contained within transaction boundaries will not be getting any Share locks and so will not be assured of transactional consistency. However, they will be able to read information and complete while other updating processes have exclusive locks on the files that would otherwise lock them out. Such reading is called ‘dirty reading’. To do dirty reading as safely as possible, use the with_dirty_reads_ok keyword with any and all appropriate there_isa’s. To also avoid unwanted interactions with contemporaneous running Sizups that are creating and removing files, also use one of the ignoring_failed_opens or with_ignoring_failed_opens_handler keywords as well.

13.3.5 Ignoring Failed Opens

There are occasions when queries accessing multiple bins in a horizontally partitioned table would like to ignore any bins that are not there or which have missing index files of any sort or in which any associated file cannot be opened for some justifiable reason. This could be because the table is being constantly updated as Sizup is being constantly run on arbitrary bins. (To be specific, the ‘not unexpected reasons’ for failures to open are precisely ENOENT, EACCESS, and ENOLINK from <errno.h> and certain btree errors: this means that either the file doesn’t exist, there are inadequate permissions to access it, it should be remotely mounted and cannot be found, or the btree is under construction. Thus ignoring_failed_opens will not hide from view other such errors as running out of file descriptors or memory.) Using the ignoring_failed_opens keyword in the associated there_is_a will achieve the desired end (ignr.nopens.1.Q). ignoring_failed_opens also works for non-horizontally partitioned RECORD_CLASSES.

For those who want finer control over those situations when data or index files cannot be opened, Daytona offers the user the opportunity to use the with_ignoring_failed_opens_handler keyword to provide to Daytona a Cymbal callback function for Daytona to call when a data or index file cannot be opened:

( 0->1 ) with_ignoring_failed_opens_handler BOOL FUN( STR .info ) = _null_fun_

Here is such a user-defined FUNCTION:
define BOOL FUN( STR .ifo ) ifo_handler 
{ 
do Write_Line( "Failed Open" ) 
do Write( .ifo ); 
return( .exit_on_fail ); 
}

In the event of a failed open for a there isa which has:

with_ignoring_failed_opens_handler ifo_handler

Daytona will call ifo_handler and pass it a STR describing that failed open which looks like this:

app=orders
recls=PARTED
bin_nbr=1
data_file_path=/home/john/d/PART.1
kind_of_file=siz
bad_file=/home/john/d/PART.1.siz

The nature of this information should be self-explanatory. If not printed as is, it can be easily parsed by using stokens(). If the return value for ifo_handler is _true_, then Daytona will exit the Cymbal program when ifo_handler returns; else it will continue with the execution as it ignores that failed open.

13.3.6 Descriptions Using using_source

As described in Chapter 3 and Chapter 23, the table corresponding to every there is a has a Source. In the simplest case, the Source can be specified in the rcd as (a shell expression for) the directory in which the data file can be found. (If no Source note is given for a data file and no Default Data_File_Source is available, then the data file is assumed to be in the directory containing the aar.) In Chapter 23, it is shown how to use an rcd Source note to specify a UNIX pipe which will generate the contents of the table.

The above specifications of a Source for a table take effect at compile time for a query. The using_source keyword provides the ability to dynamically change the source of data files at query run-time by overriding the compile-time specification. Here’s an example:

```local: STR .data_file_source
set [ .data_file_source ] = read( from _cmd_line_ but_if_absent[ "." ] );
for_each_time there_is_a SUPPLIER using_source .data_file_source
do {
    where( this_isa SUPPLIER ) do Describe;
}
```

13.4 Accessing Records By Position

Daytona offers a variety of ways to access records by their position in data files and indices. While these methods are useful, nonetheless, they are all characterized by not adhering to the relational...
model of data which considers a RECORD_CLASS or table to be a set of records in no particular order. Consequently, the notion of position really has to do with data files instead, not with RECORD_CLASSES. This becomes an impossible-to-ignore distinction when working with any horizontally partitioned RECORD_CLASS implemented with more than one non-empty data file.

### 13.4.1 there_isa_bin_first, there_isa_bin_last

One of the simplest file position-oriented requests that one might have is to return the first or last in a designated bin, either in terms of ordinal position in the file or perhaps in terms of ordinal position according to the sort order provided by a B-tree index. To this end, Cymbal provides the there_isa_bin_first and there_isa_bin_last variants of there_isa, or equivalently, there_is_a_bin_first and there_is_a_bin_last, resp. Here is an example of asking for information from the first and last ORDER records in the ORDER data file (bin_first_last.1.Q):

```plaintext
with_format _table_
  do Display each[ .onbr, .dp ]
  each_time(
    there_isa_bin_first ORDER using_siz where(
      Number = .onbr and
      Date_Placed = .dp )
  or
    there_isa_bin_last ORDER using_siz where(
      Number = .onbr and
      Date_Placed = .dp )
  );
```

Note the use of or to get information on both, which yields, not surprisingly:

```
<table>
<thead>
<tr>
<th>Onbr</th>
<th>Dp</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1984-04-24</td>
</tr>
<tr>
<td>1000</td>
<td>1984-09-13</td>
</tr>
</tbody>
</table>
```

To understand the terminology, what is being said here is that there is a bin-last ORDER; in other words, there are various kinds of last and this is a bin-last kind of last as opposed to a record-class-last kind of last. And certainly, there_isa_bin_last has nothing to do per se with the last bin.

Instead of specifying the use of the .siz ordinal index (as just done), the only other way to use these there_isa variants is to use keyed_for_index to explicitly specify a B-tree index to use in determining what is first and last (bin_first_last.1.Q):
with_format _table_
do Display each[ .onbr, .dp ]
each_time(
    there_isa_bin_first ORDER keyed_for_index `dp` where(
       Number = .onbr and
       Date_Placed = .dp )
or
    there_isa_bin_last ORDER keyed_for_index `dp` where(
       Number = .onbr and
       Date_Placed = .dp )
)
);  

The initially surprising fact about this query is that only one answer is printed out!  

\begin{verbatim}
Onbr Dp
--------
32 1984-12-28
--------
\end{verbatim}

The explanation for the missing answer is a missing field value. The facts are that Date_Placed can have missing values, that missing values are stored in the Date_Placed B-tree by default as empty strings, and that therefore entries for the records with missing Date_Placed values appear at the start of the B-tree. Consequently, even as Daytona locates the first such record, it discovers that it has to skip over that record because a Date_Placed value is requested and none is forthcoming. Here is a way to handle that situation and get some output for the first record in the B-tree sort order (bin_first_last.1.Q):

\begin{verbatim}
with_format _table_
do Display each[ .onbr, .dp ]
each_time(
    there_isa_bin_first ORDER keyed_for_index `dp` where(
       Number = .onbr and
       Date_Placed = .dp but_if_absent( .dp = `1900-01-01` ))
or
    there_isa_bin_last ORDER keyed_for_index `dp` where(
       Number = .onbr and
       Date_Placed = .dp )
    );
\end{verbatim}

with corresponding output:

\begin{verbatim}
Copyright © 2013 AT&T
All Rights Reserved.
September 15, 2013
\end{verbatim}
Another way to handle missing values in this situation is to use 
<Ignore_DFR_Missing_Values, yes> to require Daytona to omit entries for all missing key values 
from the B-tree index. This is the situation for Date_Recd in the horizontally partitioned ORDERA. 
This next query uses the hparti attributes to specify a specific bin to work with.

```sql
with_format _table_
do Display each[ .onbr, .dr ]
each_time( 
    there_isa_bin_first ORDERA keyed_for_index "dr" where(
        Region = 1 and
        Category = "B" and
        Number = .onbr and
        Date_Recd = .dr )
or
    there_isa_bin_last ORDERA keyed_for_index "dr" where(
        Region = 1 and
        Category = "B" and
        Number = .onbr and
        Date_Recd = .dr )
);
```

So with no missing value logic needed because no missing value keys are in the index, the answer is 
(bin_first_last.1.Q):

```
Onbr Dr
---
15   1900-01-01
32   1984-12-28
---
```

What would happen if the hparti attributes were not mentioned at all in this query?
with_format _table_
do Display each [ .onbr, .dr ]
each_time(
    there_is_a_bin_first ORDERA keyed_for_index `dr` where(
        Number = .onbr and
        Date_Recd = .dr )
or
    there_is_a_bin_last ORDERA keyed_for_index `dr` where(
        Number = .onbr and
        Date_Recd = .dr )
);

The answer is derivable from first principles: Daytona would process the first there_is_a_variant for each of the ORDERA bins and then do the same for the second variant. Thus all of the "first" entries would appear before the all of the "last" entries. This is probably not what one wants. Instead, the following query would be more informative because, for each bin, it identifies the bin and then the first and last elements in the bin (as sorted by Date_Recd) (bin_first_last.1.Q):

    fet [ .reg, .cat, .onbr_1, .dr_1, .onbr_n, .dr_n ]
    ist(
        there_is_a_bin_for ORDERA where(
            Region = .reg and
            Category = .cat
        ) and
        there_is_a_bin_first ORDERA keyed_for_index `dr` where(
            Region = .reg and
            Category = .cat and
            Number = .onbr_1 and
            Date_Recd = .dr_1 )
    and
        there_is_a_bin_last ORDERA keyed_for_index `dr` where(
            Region = .reg and
            Category = .cat and
            Number = .onbr_n and
            Date_Recd = .dr_n )
    )
    do Write_Words( .reg, .cat, .onbr_1, .dr_1, .onbr_n, .dr_n );
}

Note the use of there_is_a_bin_for to get a listing of all pairs of hparti attributes. The output looks like this:
All this being said, one of the most useful ways to use this functionality is just to be able to quickly and efficiently identify the largest value of an indexed field, whether the RECORD_CLASS is partitioned or not (bin_last.1.Q):

```c
do Write_Words( "Max_ORDERA_Date_Placed =", 
    max( over .dp each_time( 
        there_is_a_bin_last ORDERA keyed_for_index "dp" 
        where( Date_Placed = .dp ))));
```

This is much more efficient (read faster) than the equivalent:

```c
do Write_Words( "Max_ORDERA_Date_Placed =", 
    max( over .dp each_time( 
        there_is_a ORDERA 
        where( Date_Placed = .dp ))));
```

because the former only involves Daytona doing a $\log n$ search of the Date_Placed B-tree for each bin (and actually, sometimes a quick linear scan of the last B-tree node) whereas the latter will do a sequential scan of all the records in each bin, using both the data file and its .siz file.

### 13.4.2 Using Descriptions To Access Records By Ordinal Position

Once again, although relational database theory considers tables to be sets of records, i.e., no duplicates and no particular sort order for the records, Daytona’s implementation of a bin of records does generate an order of appearance for each record in the file. There are occasions when it is convenient to be able to access a record by its ordinal position in the data file. The at_bin_pos keyword can be used in a Cymbal description to cause Daytona to retrieve a record by ordinal position (atbinpos.1.Q):
for_each_time [ .order_nbr ]
  is_such_that(
    .pos Is_In [ -300, 0, 1, 10, 15, 254, 750, 999, 1000, 1005 ] and
    there_is_a ORDER at_bin_pos .pos where(
      Number = .order_nbr
    )
  ) do {
    do_Write_Line(.order_nbr);
  }

This query prints the ORDER Number taken from records appearing at the indicated positions in the file. The positions begin at 1. Attempts to access records that are not there, e.g., beyond the end of the file, yield a Cymbal description whose truth value is false. If the slot in the file corresponding to an ordinal position contains a deleted record, then Daytona will scan forward to the next (undeleted) record, if any, and take that instead. Just as with sampling, the ordinal positions are taken to be within the current bin as opposed to being within the table as a whole, which of course could be the union of many bins in the horizontally partitioned case. Daytona implements at_bin_pos efficiently by seeking directly to the desired location, thus ignoring any other records.

This at_bin_pos capability enables a query to visit every $n^{th}$ record in a file:

set .idx = 1;
while( there_isa ORDER at_bin_pos .idx )
do {
  for_each_time .onbr is_such_that( this_isa ORDER where( Number = .onbr ) )
do { do_Write_Line(.onbr) }
  set .idx +=100;
}

The corresponding with_bin_pos vbl keyword takes a VBL of the user’s choice and populates it appropriately with the ordinal position of the current record in the bin.
set .color = "beige";
do Display each[ .bpos, .nbr, .name, .wt ]
each_time(
    there_isa PARTC with_bin_pos_vbl bpos
    where(
        Number = .nbr and
        Name = .name and
        Weight = .wt and
        Color = .color )
    // test .bpos out for fun
    and there_isa PARTC at_bin_pos .bpos
    where(
        Number = .nbr and
        Name = .name and
        Weight = .wt and
        Color = .color )
);

with_bin_pos_vbl cannot be used unless the associated access uses the .siz file.

13.4.3 Using Descriptions To Access Records In A Cluster By Ordinal Position

Just as Daytona supports seeking to a particular record in a file by ordinal position, it also supports seeking to a particular record in a cluster (in a file) by ordinal position in the cluster. This is accomplished by using the keyword at_bin_cluster_elt_pos (btclustelt.1.Q):

do Display each[ .nbr, .name, .wt ]
each_time(
    there_isa PARTCU at_bin_cluster_elt_pos 2
    where(
        Number = .nbr and
        Name = .name and
        Weight = .wt and
        Color = "khaki" )
);

In this query, PARTCU is a RECORD_CLASS which has a Unique cluster B-tree INDEX on Color. (Indeed, in order to prevent confusion, the cluster B-tree that is associated with the use of at_bin_cluster_elt_pos does have to be Unique, meaning that there is at most one cluster per cluster KEY value.) Anyway, what happens with this query is that the cluster index is used to locate the (unique) cluster in the data FILE using the condition Color = "khaki" and then the record at at_bin_cluster_elt_pos 2 within that cluster becomes the record associated with the there_isa. This happens as quickly as it would take to get access to the first record of the cluster because by design,
that first record is not actually seeked to but rather only the ordinally indicated cluster member is
seeked to. The ordinal positions begin at 1; furthermore, the specification of an ordinal position that
exceeds the size of the cluster will result in no record being found. Obviously, the utility of this
construct is dependent upon the user knowing in advance that the elements of the clusters have been
sorted by some criterion and that the user can identify the ones of interest by means of their ordinal
position in the cluster. This is a file (BIN), not RECORD_CLASS, -based construct meaning that the
determination of the records to visit is done independently for each qualifying BIN in the horizontal
partitioning case. Here is an example of finding the start of the cluster and then using at_bin_pos to
visit the even elements of the cluster (btclustelt.1.Q):

```plaintext
set .color = "khaki";
do Display each[ .bpos, .idx, .nbr2, .name2, .wt2 ]
each_time(   
    there_is_a PARTCU at_bin_cluster_elt_pos 1 with_bin_pos_vbl bpos
      keyed_for_index 'c'
    where(
      Color = .color )
    and .idx Is_In [ 0-> 6 by 2 ]
    and there_is_a PARTCU at_bin_pos .bpos+.idx
    where(
      Number = .nbr2 and
      Name = .name2 and
      Weight = .wt2 and
      Color = .color )  // makes sure no wandering outside of cluster
);
```

Suitable updates and deletes (no adds) are allowed as in btclustelt.1.IQU:
define PROC( STR(25) .color, INT .cepos ) txn task:
Update_Wt
{
  fet [ .wt ]
  ist(
    // TEST: hparti bin PARTEC.2
    there_isa PARTEC at_bin_cluster_elt_pos .cepos
    where(
      Date_Added = "12-3-87" and
      Info_Source = "March Hare" and
      Group_Nbr = 11 and
      Rating = .85 and
      Number = .nbr and
      Name = .name and
      Weight = .wt and
      Color = .color )
  )
  do Change so_that( this_isa PARTEC where( Weight = 1000*.wt ) );
}
define PROC( STR(25) .color, INT .cepos ) txn task:
Delete_Clustelt
{
  do Change so_that(
    there_is_no PARTEC at_bin_cluster_elt_pos .cepos
    where(
      Date_Added = "12-3-87" and
      Info_Source = "March Hare" and
      Group_Nbr = 11 and
      Rating = .85 and
      Color = .color ));
}
However, no update or delete is allowed if it could potentially lead to splitting the cluster into two separate clusters, which would violate the uniqueness constraint. This implies that any updates to records have to be done in place.

13.4.4 Sampling Records Using Descriptions

Sampling provides an excellent way to compute the characteristics of a population of database records when it would simply take too long to visit every database record and perform the required calculations to compute the quantities exactly. For example, if it took on the average, say, 1 millisecond to visit a record and extract the required information for subsequent aggregation, then it would take about 14 hours to visit each of 50 million records in order to compute the desired aggregate statistics.
On the other hand, the truly amazing thing about sampling is that a sample of a few thousand records or less can yield estimates of aggregate quantities that are, say, 99% sure to be within a few percent of the true values for those quantities. And if the collection of database records itself is a sample (as would be the case, for example, with a day’s worth of bank transaction data), then there is probably really little point in computing the exact value of the aggregate statistics for any given day when the goal is to predict, say, the average weekday volume.

Fortunately, Cymbal descriptions offer built-in sampling support. They will produce true without-replacement random samples that are as efficient as possible since they seek and read just those records that happen to be in the sample. In other words, whatever records are not in the sample will be ignored and not processed in any way over and above possibly being read into memory in the same block as a sampled record. Consequently, a sample of size 1000 will take (essentially) 1000 data file seeks and record unpacks regardless of the size of the file it is taken from.

Cymbal descriptions using the from _a_bin_sample_of_size keyword or the from _a_bin_sample_of_frac keyword will cause Daytona to efficiently sequentially scan the data and draw the specified sample. The former keyword is used when an exact INT sample size is specified and the latter keyword when a sampling fraction between 0.0 and 1.0 is specified. Here’s how to estimate the average ORDER Quantity based on a sample of size 50 (samp.1.Q):

```c
do Write_Line( "estimated avg Quantity from a sample of size 50 =",
    avg( over .qty
    each_time(
        there_is_a ORDER from_a_bin_sample_of_size 50 where(
            Quantity = .qty
        )
    )
) );
```

The random sampling algorithm used is Algorithm S from Volume 2 of Knuth: this creates what is called a “without replacement” random sample, meaning that there are no ties, no items selected more than once for the sample. The random number generation uses rand _uni(); consequently, in order to get a different a sample, just run call rand _uni() a number of times and the random number generation will advance in its sequence and a new sample will be available. Sample sizes that exceed the population sizes will elicit a warning and cause the entire population to be included in the sample. Deleted records in the file are ignored but if there are too many of them, then the target sample size may not be attained and a warning message will be issued.

The word _bin is used in these keywords to emphasize that the sampling is done on a bin-by-bin basis in the horizontally partitioned case (and thinking of the non-horizontally-partitioned case employing one bin to contain the entire table.) In more detail, the sampling parameter, whether it be the size or the fraction, is applied to each bin as it appears for processing. Consider for example a 25% sample from the horizontally partitioned ORDERA table:
for_each_time [ .order_nbr ]
is_such_that(
    there_is_a ORDERA from_a_bin_sample_of_frac .25
    where(
        Region = 3 and
        Number = .order_nbr
    )
) do {
    _Show_Exp_To(.order_nbr);
}

This table is partitioned on both Region and Category, the lack of mention of the latter implying that all seven of the Region 3 ORDERA bins will be visited (corresponding to Categories A through G). The semantics is that regardless how large each bin is, a sample of 25% will be drawn from each.

13.4.5 there_is_a_next

One of the variants of there_is_a that Daytona provides is called there_is_a_next. Each time program control passes through a there_is_a_next, Daytona resumes reading from the next record after the last one visited during the previous invocation. If no file access method is explicitly given, there_is_a_next assumes using_siz, meaning that the records will be accessed using the siz file in order of increasing offset in the data file and in the case of horizontally partitioned data, from the first bin to be mentioned in the rcd to the last. Conversely, there_is_a_next with an explicit using_reverse_siz begins with the record at the greatest offset in the data file and proceeds towards the start of the file; in the case of horizontally partitioned data, the process begins with records in the last bin to be mentioned in the rcd and proceeds towards the first. Of course, this is the behavior one would expect when employing using_siz and using_reverse_siz with a plain vanilla there_is_a. The distinguishing characteristic of there_is_a_next is that it remembers its position from one visit to the next. In other words, if the flow of control revisits the assertion containing the there_is_a_next, then the there_is_a_next will resume reading with the next record, if any, after the one it finished with the last time. Here is a simple example taken from tailf.f.Q:
do Get3();
do Get3();

define PROC Get3()
{
    set .i = 0;
    for_each_time [.num, .name, .col] is_such_that(
        there_is_a_next PART where(
            Name = .name and Number = .num and Color = .col )
    ) do {
        do Write_Words(.num, .name, .col);  
        set .i++;
        when (.i = 3) break;
    }
    do Write_Line("Get3 done");
}

The output is:

100 Crt screen blue
101 transistor red
102 keyboard magenta
    Get3 done
103 wheel orange
104 nut purple
105 bolt azure
    Get3 done

When all the records have been visited, subsequent visits to \texttt{there\_is\_a\_next} will fail to produce any records, \textit{unless updating transaction has added some in the meantime.}

As a result, \texttt{there\_is\_a\_next} can provide a kind of \texttt{tail -f} capability for periodically examining data files to see what is new. This is a feature that is useful for certain special-purpose applications. Typically, \texttt{there\_is\_a\_next} is placed in a transaction task which is called periodically to process whatever data records had been added since the last time it was called. This is what occurs in the following query:
set .out_chan = _stdout_;  
for_each_time .ii Is_In [ 1 -> 10 ]  
do{  
    do Report_Latest_ORDERS;  
    do Sleep( 1 );  
}
global_defs:  
declare PROC transaction task: Report_Latest_ORDERS ()  
{  
    import: CHAN .out_chan
    for_each_time [ .ord_nbr, .qty ] is_such_that(  
        there_is_a_next ORDER_ where(  
            Number = .ord_nbr and Quantity = .qty )  
    ){  
        to .out_chan  
        do Write_Words( .ord_nbr, .qty )  
    }  
}  

there_is_a_next can be used with a skipping keyword argument so as to obtain a special ungetc-like feature whereby it skips any number of record slots forward or backward (as indicated by the sign of skipping argument) before it resumes reading. Note that skipping works in terms of record slots, not records, where a record slot is either the space left behind by a deleted record or the space occupied by an active record. If skipping some number of slots lands on the slot of a deleted record, skipping will continue on to the first slot containing an active record and stop there. In the non-horizontally-partitioned case, in the using_siz context, any attempt to skip before the start of the file will result in reading from the start and an attempt to skip beyond the end of the file will position the cursor just past the last record and not result in reading anything. The latter can also be achieved by using the special _to_eof_ skipping argument. In the case of a horizontally partitioned record class, in the using_siz context, any attempt to skip before the start of the current bin will result in reading from the start of that bin (i.e., not necessarily the first bin) and an attempt to skip beyond the end of the entire record class or a deliberate use of _to_eof_ will result in the cursor being positioned just after the last record in the last bin and not result in not reading anything. The reverse of these scenarios holds when using_reverse_siz is used. Incidentally, because UNIX handles reading backwards in a file so differently from reading forwards, using_reverse_siz uses a buffer size of 8192, regardless of what the Seq_Acc_Bufsize has been set to. Here is an example of the special argument _to_eof_ being used to cause skipping to the end of a record class:

there_is_a_next ORDERA skipping _to_eof_ where(  
    Number = .ord_nbr  
    and Quantity = .qty )  

Note that if there_is_a_next is being used to monitor many bins of a horizontally partitioned
record class, then regardless of how various Max_Open_Bins parameters have been set, there_is_a_next will NOT lose its memory of where it last was for any bins that it has to close.

There are several caveats to the use of there_is_a_next. First, it does require that a .siz file exist and provide the usual ordinal indexing of the data file records. Second, any task it is in must be flush_on_return because if it is close_on_return, then on return, all accesses are closed and therefore all associated state position such as file cursor position is lost. Also, in order for a there_is_a_next to be sensitive to updates to its table made within the same process by other tasks, the there_is_a_next should be within a transaction task.

Further, note that if there_is_a_next is being used with FILES which have a <Reuse_Freed_Space, yes> annotation in their rcd, then since the free tree is available, other transactions running in the interim may delete old records and add new records back into the slots left by the old records. Since the methodology Daytona uses to keep track of which records are next to visit is based solely on keeping a cursor in the siz file, Daytona would fail to detect these new records coming into old slots already passed by.

13.4.6 skipping With there_isa

There are also occasions when the user wishes to skip in a certain number of record slots in a BIN before starting to read the remainder. This can be achieved by using the skipping keyword with a there_isa (there_isa_skipping.1.Q). The difference between this use of skipping and the one associated with there_is_next is that there_is_next is designed to remember where it was when last used, hence living up to its name, whereas everything is de novo as usual for there_isa when used with skipping. All motion is relative to each BIN in turn that is visited (if there is more than one). If the skipping argument is negative, it is ignored, which will result in all records in the BIN being visited; any attempt to skip to a place beyond the end of a BIN will result in no records being visited for that BIN.
14. Aggregation In Cymbal

Aggregation occurs in the database setting when the user defines one or more groups of objects and then computes one or more characteristics for each of these groups. Such group characteristics commonly include count, sum, avg, prod, min, and max. Cymbal offers more of these aggregate measures than usual by offering as well the second order statistical measures stdev, var, covar, and corr. This chapter shows how to use Cymbal both in defining sets of groups in very general ways and in specifying which aggregate measures to compute. Also included is a discussion of how to compute top-k queries: this is a special kind of aggregate computation which in effect maps a sequence of TUPLEs into a LIST of just the top (or bottom) k TUPLEs according to some sorting criterion -- and most importantly, does so in such a way that never are more than k+1 TUPLEs actually stored during computation.

14.1 Scalar Aggregate Functions With No Grouping

Basically, any Cymbal aggregate function should be thought of as being just like any other Cymbal function taking keyword arguments. The only difference is that one of an aggregate function’s keyword arguments is an assertion: assertions are not usually encountered as fpp arguments but there is no reason why they shouldn’t be and in this case, it is essential to have them be so. Here is a query that asks for a count of the number of orders that have been placed but not received (cf., countabs.1.Q):

do Display
   with_title_line "count the number of missing Date_Recd values"
   each[ .nbr_absent ]
   each_time(
      .nbr_absent = count(
         each_time( there_is_an ORDER where( Date_Recd Is _absent_ )))
   );

As can be seen, count, like all Cymbal aggregate functions, takes an assertion argument for the keyword each_time. This query asks for all values of nbr_absent such that somehow its value is equal to the count of each time there is an ORDER record where the Date_Recd FIELD value is missing.

Here is a query that produces the average weight of parts described in the PARTS table:
do Display
    with_title_line "find the average weight of the parts"
each[ .avg_wt ]
each_time(  
    .avg_wt = avg( over .wt  
        each_time( there_is_a PART where( Weight = .wt )))  
)

Notice that avg, like most aggregate functions, must have an argument for the keyword over. Arguments for the keyword over may be any scalar term with the exception of calls to the second order statistical functions covar and corr (for covariance and correlation) which take instead a 2-element TUPLE of scalars as their over argument (see ny.blu.stats.Q below).

The argument for each_time in an aggregate function call must be an assertion whose free variables include the variables referenced in the over arguments; any other free variables must have been finitely defined on first use previously. It is worth emphasizing that the assertion argument for each_time can otherwise be any valid assertion; as such, it can contain additional aggregate function calls as well as quantifiers and, in fact, anything that would be acceptable to give to Display for its each_time argument.

The meaning of these aggregate function calls should now be clear; in this example, the call to avg means

"each time a value for the wt variable is found such that there is a PART record where the Weight field value is that value, collect that value in a list of such values and then return the average value for that list."

Even though most of the examples in this chapter are database-oriented, any Cymbal aggregate function can also work with assertions that make no reference whatsoever to there_is_a and the like. Here is an example (simpagg.IQ):

    set [ .bound ] = read( from _cmd_line_ but_if_absent[10]);
    do Write_Line( "Stdev = ",
        rtn( stdev( over .n*.n each_time( .n Is_In [ 1 -> .bound ] )), .001) );

Note the use of an expression as the argument to over. Also, use the function rtn (or round_to_nearest) to reduce the number of places after the decimal point that are printed when wanting to print out a floating point number.

The same homonym VBL scoping conventions used for boxes and Displays and dynara (see Section 12.1) are used for these aggregate function calls as illustrated by agg.scoping.Q:
set .x = 2;

// new VBL x here
set .y = min( over .x each_time(.x Is_In [33, 55]));
_Show_Exp_To(.y)
_Show_Exp_To(.x)

// same VBL x as in first assignment
set .y = min( over .z*.x each_time(.z Is_In [20+.x, 77]));
_Show_Exp_To(.y)
_Show_Exp_To(.x)

Here is the output:

.y = 33
.x = 2
.y = 44
.x = 2

Here are the imports for all of the built-in scalar aggregate functions:

INT FUN:
  count( (0->1) distinct,
          (0->1) over VALCALL | TUPLE[(1->) VALCALL],
          each_time ASN )

overloaded STR|FLT|INT|UINT|IP|TIME|DATE|CLOCK|DATE_CLOCK|BOOL|MONEY FUN:
  min( over STR|FLT|INT|UINT|IP|TIME|DATE|CLOCK|DATE_CLOCK|BOOL|MONEY, each_time ASN )

overloaded STR|FLT|INT|UINT|IP|TIME|DATE|CLOCK|DATE_CLOCK|BOOL|MONEY FUN:
  max( over STR|FLT|INT|UINT|IP|TIME|DATE|CLOCK|DATE_CLOCK|BOOL|MONEY, each_time ASN )

overloaded STR|INT|UINT|FLT|TIME|MONEY FUN:
  sum( (0->1) distinct,
       over STR|INT|UINT|FLT|TIME|MONEY, each_time ASN, (0->1) with_sep STR )

overloaded FLT|TIME|MONEY FUN:
  avg( (0->1) distinct, over FLT|TIME|MONEY, each_time ASN )

STR|INT(?)|UINT(?)|FLT(?)|TIME|MONEY FUN
  median( (0->1) distinct,
over STR|INT(?)|UINT(?)|FLT(?)|TIME|MONEY,
each_time ASN )
STR|INT(?)|UINT(?)|FLT(?)|TIME|MONEY FUN
quantile(
    ( 0->1 ) distinct,
    namely FLT,
    over STR|INT(?)|UINT(?)|FLT(?)|TIME|MONEY,
    each_time ASN )
overloaded FLT|INT|UINT|TIME FUN
prod( over FLT|INT|UINT|TIME, each_time ASN )
overloaded FLT FUN
stdev( ( 0->1 ) distinct, over FLT, each_time ASN )
overloaded FLT FUN
var( ( 0->1 ) distinct, over FLT, each_time ASN )
overloaded FLT FUN
covar( over TUPLE[ FLT, FLT ], each_time ASN )
overloaded FLT FUN
corr( over TUPLE[ FLT, FLT ], each_time ASN )

(Actually, there are two more, the little-used sq_mean_dev_sum and cross_mean_dev_sum.)

Of particular note here is the overloaded sum function which may take an STR over argument. In
the STR case, the function used to ‘add’ quantities together is the string concatenation function. Any
with sep argument is interspersed among the over quantities as they are concatenated together. The
STR sum aggregate function is frequently useful when working with LIST/SET-valued fields
(multival.4.Q):

for_each_time [ .parent, .kid_cat ]
is_such_that(
    there_is_a PERSON where( Name = .parent )
    and .kid_cat
    = sum( over .kid
        each_time( this_isa PERSON where( one_of_the Children = .kid ) )
        with_sep "*" )
    )
    do Write_Line( .parent, ":[", .kid_cat, "]" );
}

This invocation of the quantile function returns the .25-quantile of the indicated LIST, which of
course is quietly sorted behind the scenes in order to find the necessary value:
do Write_Line(
    quantile( namely .25 over .x each_time(
        .x Is_The_Next_Where( .x Is_In [ 3, 1, 22, 41, 12, 47, 15, 4, 12 ] )
    )));

The median of course is the .5-quantile.

Also, when the keyword \textit{distinct} is used in conjunction with those aggregate functions that support it, the effect is to eliminate duplicates before applying the aggregate function.

\section*{14.2 There Is Something About $count$}

The previous use of $count$ is actually quite unusual relative to the way that Cymbal queries typically work. Recall that the query is essentially this:

\begin{verbatim}
do Write_Line( count( each_time( there_is_an ORDER where( Date_Recd Is _absent_ )))
)
\end{verbatim}

Notice that there are no variables whatsoever appearing in this query. How can it possibly be valid? After all, Cymbal queries typically request the production of all TUPLEs of values for an assertion’s free VBLs that will satisfy the assertion after being substituted in the assertion. Yet in this case, there aren’t even any VBLs! Furthermore, consider this query:

\begin{verbatim}
local: INT .one = 1
do Write_Line( count( each_time( .i Is_In [ .one -> 5 ] )));
// which is an abbreviation for:
do Write_Line( count( each_time( there_exists .i such_that( .i Is_In [ .one -> 5 ] )))
\end{verbatim}

While the \textit{each_time} assertions here each reference the values of two VBLs, the query is not asking to produce any values for either one: after all, \textit{i} is not free since it is being quantified over and the VBL \textit{one} is an outside procedural VBL that functions as a constant in the assertions. Or to put it differently, the \textit{each_time} assertions in both of these queries are ground. (They are also both closed.) That’s what these two queries have in common.

Yet, when read as (mathematical) English, the meaning of these two queries is clear: in the first case, the total number of ORDER records with a missing Date_Recd value should be produced and in the second case, surely 5 is the number of times that \textit{i} can assume values in $[1\rightarrow 5]$.

To handle this type of query, Daytona Cymbal processing has been extended to support a different paradigm called \textit{GroundEnumeration} processing. Ordinarily, all that Daytona can do with a ground assertion is to determine its truth value. However, in \textit{GroundEnumeration} cases, Daytona will force itself to enumerate all possible ways that prove that the assertion is true, assuming it is true; otherwise, when false, there is nothing that can be done. So, for example, the case where \textit{i}=2 is one way to show that

there_exists .i such_that( .i Is_In [ .one -> 5 ])

is true.

\textit{count} then, as used above, simply returns the total number of ways to show that the assertion is true. There is another way to use \textit{count} and that is with an \textit{over} argument:
do Write_Line( count( over .n each_time(
    there_is_an ORDER where( Number = .n and Date_Recd Is _absent_ ))));

Since the each_time assertion is no longer ground due to the free occurrence of the VBL n, this is not the Ground_Enumeration situation but rather it is the typical one where Daytona is being asked to produce all values for the free VBLs that satisfy the assertion. The fact that the system is not doing anything more with the values other than reporting the total number of them is not material to the nature of the query. In this case, this use of count produces the same answer as before because, in the sample database, every ORDER record with a missing Date_Recd has a value for the Number field, which needn’t necessarily be the case.

However, the next query illustrates that these two ways of using count can produce different results (count.9.Q):

    local: INT .one = 1
    do Write_Line( "20 =",
        count( each_time(
            .i Is_In [ .one -> 5 ]
            and there_isa SUPPLIER where( City = "Omaha" ))));
    do Write_Line( "5 =",
        count( over .i each_time(
            .i Is_In [ .one -> 5 ]
            and there_isa SUPPLIER where( City = "Omaha" ))));

For the sample database, since there are 4 SUPPLIER records whose City is Omaha, the first count returns 20 whereas the second count returns 5. The explanation is simple: the GroundEnumeration processing for the first count causes each of those SUPPLIER records to be visited for each value that i takes in the INTERVAL of size 5; on the other hand, since the evaluation of the second count sees the SUPPLIER assertion as being a ground test, for each of the 5 values for i, that test is just seen to succeed (once).

To test the reader’s understanding, here is another example of how they may differ (count.9.Q):
So, there is a choice of two counts: one counts all ways to make a ground assertion true and the other counts all ways to satisfy an assertion with free VBLs.

Ground Enumeration is not just restricted to count; it shows up in other settings as well. Here is one:

```plaintext
for_each_time there_isa SUPPLIER where( City = "Omaha" )
{
    with_format _packet_ where this_isa SUPPLIER do Describe;
}
```

See also count.A.Q and count.B.Q.

### 14.3 Nested Scalar Aggregate Functions With No Grouping

Since ordinary function calls can be nested, so also can aggregate function calls. The next query nests a call to `sum` within a call to `corr` in order to ask for the correlation between the weights of parts and the total amounts ordered for each part (cf., qtywt.corr.Q).
do Display
            with_title_line
            "find correlation between weight of part and total ordered"
          each [ .qty_wt_corr ]
          each_time(
            .qty_wt_corr =
              corr(
                over [ .part_wt, .total_ordered ]
                each_time( somehow(
                  there_is_a PART where(
                    Number = .part_nbr
                    and Weight = .part_wt )
                  and .total_ordered =
                    sum( over .qty
                      each_time(
                        there_is_an ORDER where(
                          Part_Nbr = .part_nbr
                          and Quantity = .qty )))
                  )
                )))
          )

The use of part_nbr in this query serves to illustrate how information from outside an aggregate function call can be communicated to inside the call. Here, the scope of part_nbr is that of the somehow which, in fact, is an abbreviation for there_exists .part_nbr. Since the entire sum function call is within this somehow’s scope, the value of part_nbr can be communicated to within the sum call merely by referencing it. Notice that, as required, part_nbr is finitely defined on first use before the sum call. It is in this way that totals for each part are generated. (By the way, it is not necessary for the user to explicitly write the somehow because if they do not, the system will in effect put it in anyway: see the rules in Chapter 5.)

Observe also how much more general this use of correlation is than just being able to compute the correlation between two columns of the same table. In fact, in this case, one of the columns doesn’t even exist explicitly as a column in the database but rather is computed by summing up database Quantities from a different table.

14.4 TUPLeS Of Aggregate Functions With No Grouping

There are occasions when the values of several aggregate functions are desired for the same data. Instead of capturing the desired values one by one by constructing an aggregate function call for each, the Cymbal built-in function aggregates enables the user to specify and compute a TUPLE of aggregate function values in one step by processing a (common) each_time assertion exactly once, instead of once per aggregate function. The next query accomplishes this for a count of all parts along with the total, average, minimum, and maximum of the part weights (cf., partwtstats.Q).
do Display
    with_title_line "number, sum, avg, min and max weights over all parts"

each[ .number, .total_wt, .avg_wt, .min_wt, .max_wt ]

each_time(

    [ .number, .total_wt, .avg_wt, .min_wt, .max_wt ] =
    aggregates(
        of [ count(), sum( over .wt ), avg( over .wt ),
        min( over .wt ), max( over .wt )]
        each_time(
            there_is_a PART where( Weight = .wt ) )
    )
)

The aggregates function is a function that returns a TUPLE of values. It takes two keyword arguments: the argument for of is a list of scalar aggregate function calls without their each_time arguments and the argument for each_time is an assertion as usual. As can be seen, the each_time argument has been effectively factored out of the several aggregate function calls in the of argument list. Even though only one over term is used in this example, in general, any number of different over terms may be employed as long as any variables they contain appear free in the each_time argument. Of course, the implication is that all of the variables over all of the over arguments must be free in the shared each_time assertion. This in turn implies that a count() with no over is not a Ground Enumeration count after all if some other aggregate function in the aggregates() call is using an over argument of its own.

Just as with read and tokens, aggregates can also be used to assign values to TUPLE-valued VBLS or conventional ARRAY-valued VBLS.

The keyword distinct is not allowed in the of argument to any aggregates() call. Of course, an Is_Something_Where can be deployed to bracket an entire each_time assertion so as to eliminate duplicates in the TUPLEs of values that it generates. Furthermore, neither median nor quantile can be used as arguments to any aggregates() call. However, the effect of any hoped-for aggregates() call using distinct, median, or quantile can be achieved by using a "constant" special-case Cymbal group-by BOX call (to be described shortly), where the single group is indicated by a constant expression (like 47): see medquant.3.Q for examples.

An aggregates function call is computed faster than the corresponding series of single aggregate function calls would be since all ways to satisfy the each_time assertion are computed only once instead of once for each call in the case of the series.

The next example indicates that the each_time assertion for an aggregates call may make reference to any number of variables and that consequently the aggregates computed may be "over" different variables (and terms) (cf., ny.blu.stats.Q):
with_title_lines [ "for New York suppliers, count, avg wt(in kg), avg qty,",
    " and corr of wt & qty for orders of bluish parts" ]
do Display
each[ .supplier, .nbr, .avg_qty, .avg_wt, .corr_qty_wt ]
each_time(
    there_is_a SUPPLIER named .supplier where(
        Number = .supp_nbr
        and City = "New York"
    )
    and [ .nbr, .avg_qty, .avg_wt, .corr_qty_wt ]
    = aggregates(
        of[ count(), avg( over .qty ), avg( over 2.2*.wt_in_lbs ),
            corr( over [ .qty, 2.2*.wt_in_lbs ] ) ]
    each_time(
        there_is_an ORDER where(
            Supp_Nbr = .supp_nbr
            and Part_Nbr = .part_nbr
            and Quantity = .qty
        )
        and there_is_a PART where(
            Number = .part_nbr
            and Color Is_In
            { "blue", "powder blue", "turquoise", "azure" }
            and Weight = .wt_in_lbs
        )
    )
)

Notice the 2-element list used as the argument for the keyword over in the corr call above.

14.5 Scalar Aggregate Functions With Grouping: Initial Formulations

Sometimes the user wishes to divide up the data into groups and to compute one or more
aggregate measures for each group. This is what SQL accomplishes with GROUP BY and HAVING
whereby tuples are GROUPED BY their having common values for one or more columns and then
aggregates are computed on each of the groups and kept according to whether or not their values meet
the optional HAVING condition. Cymbal accomplishes this with more generality and surprisingly
does this with the primitives that are already in the language.

Cymbal provides several ways of expressing group-by-having queries. For pedagogical reasons,
these are presented in order of increasing power, efficiency, and conciseness. The next query asks the
system to determine the number of parts ordered of each color (cf., partbreak.Q ) as long as that total is
greater than 100000 and to sort the results by decreasing total.
sorted_by_spec[-2]
do Display
    with_title_line "Number of parts ordered of each color"
each[ .color, .nbr_parts_ordered ]
each_time(
    .color Is_Something_Where ( there_is_a PART where( Color = .color ))
    and .nbr_parts_ordered =
    sum( over .qty
        each_time(
            there_is_a PART where(
                Number = .part_nbr
                and Color = .color )
            and there_is_an ORDER where(
                Part_Nbr = .part_nbr
                and Quantity = .qty ) ))
        and .nbr_parts_ordered > 100000
    )

The first conjunct above serves to partition the parts up into groups having the same color whereas the second conjunct defines the value of the \textit{nbr\_parts\_ordered} variable to be the sum over the quantities of parts ordered of a given color.

\textit{Important Caveat}: There are two drawbacks to this approach! First, it accesses the data twice, first sequentially to compute the groups, and then second by the presumed index on \textit{Color}. Therein lies the second drawback, which is that not only is there the presumption of an index being available but also that it is fast to access all the data by means of an index, which in general, is most certainly not true. Both of these drawbacks can be avoided by adopting the one-pass associative array approach also explained in Chapter 11 and illustrated here (dynara.grpby.1.Q):
local: INT ARRAY[ STR .color : with_default @ => 0 ] .part_total

do WriteLine( "This is dynara.grpby.1.Q" );

for_each_time [ .color, .qty ]
is_such_that(
    there_is_a PART where( Number = .part_nbr and Color = .color )
    and there_is_an ORDER where( Part_Nbr = .part_nbr and Quantity = .qty )
){
    set .part_total[.color] += .qty;
}

for_each_time [ .color, .tot_qty ]
Is_The_Next_Where( .part_total[.color] = .tot_qty
    and .tot_qty > 100000
) sorted_by_spec[-2]
{
    do Write_Words( .color, .tot_qty );
}

This query goes sequentially through the PART table once which is the least amount of work that can be done since every record is in some group and will need to be visited at least once. This will necessarily faster than the two-pass, indexed approach.

14.6 Scalar Aggregate Functions With Grouping Via BOX-formers

By means of group-by BOXes, Cymbal provides much more compact, efficient, and declarative syntax for expressing group-by-having’s of all sorts. Here is how this syntax would express the preceding query (grouper.1.Q):

\[
\text{with_format \_table\_} \\
\text{with_col_labels [ "Color", "Tot_Qty" ]} \\
\text{sorted_by_spec[-2]} \\
\text{do Display each\_tuple\_of} \\
\text{[ [ .color, \text{sum( over .qty )} ] :} \\
\text{there\_is\_a PART where( Number = .part\_nbr and Color = .color )} \\
\text{and there\_is\_an ORDER where( Part\_Nbr = .part\_nbr and Quantity = .qty )} \\
\text{: selecting\_when( sum( over .qty ) > 100000 )} \\
\text{]};
\]

In general, the BOX-former syntax for group-by is the usual LIST/SET-former syntax extended so that the TUPLE specification contains one or more aggregate function call prototypes in addition to one or more simple atomic terms. Just as with the \textit{aggregates} function call, these aggregate function call prototypes are not thought of as being evaluated in place in and of themselves (how can they be -- they don’t have an \textit{each\_time} assertion argument) but rather they function as patterns indicating what aggregates are to be computed for each TUPLE in the LIST/SET. (However, in contrast to \textit{aggregates} calls, these aggregate function call prototypes are allowed to use the \textit{distinct} keyword in the same way
that is supported for the scalar non-grouped case.)

The atomic terms in the TUPLE are the group-by quantities; they can be any term (see Chapter 5 for the definition of term). The system considers there to be exactly one group per distinct TUPLE of values for the group-by terms; the prototype aggregate function calls are evaluated over the values generated for their over argument terms each time the BOX-former assertion generates values for its free variables that yield the specific values for the group-by terms that specify a given group. (For typical purposes, the user will not notice any difference in output depending on whether SET or LIST syntax is used for the group-by BOX. Differences (that of course are confined to whether there are duplicates or not) only appear in advanced uses of group-by BOX where certain aggregate functions themselves compute a BOX which is then spliced into the results.)

From the example above, the group-by term is .color. Suppose red is a value for the group-by term. Then the sum is computed over all values of the VBL qty each time the assertion is satisfied in such a way that .color = "red".

Any HAVING condition is expressed using a selecting_when keyword argument in the BOX-former where the selecting_when assertion can make reference to grouping quantities and aggregate function call specifications (as well as to box ancillary variables).

For emphasis, a group-by SET is just an extension of the previously defined BOX construct and so it can avail itself of any general BOX functionality that makes sense. Consider for pedagogical purposes (grouper.1.Q):

```plaintext
with_format _table_
with_col_labels [ "Color", "Tot_Qty" ]
sorted_by_spec[-2]
do Display each_tuple_of
   { [ upper_of(.color), sum( over .qty ) ] :
      there_is_a PART where( Number = .part_nbr and Color = .color )
      and there_is_an ORDER where( Part_Nbr = .part_nbr and Quantity = .qty )
      : with_candidate_index_vbl ci
         selecting_when( strlen(.color) > 3 and sum( over .qty ) > 100000 and .ci % 2 = 0 )
   };
```

Note the use of the ancillary box VBL ci. Along these same lines, a group-by SET can be used just like any other BOX (for example, by defining it to be the value of some VBL) and so does not have to be used as it is here as the argument of a keyword to Display.

### 14.6.1 Comparison With SQL Group-By Syntax

Incidentally, in SQL, penultimate group-by-having query could be written as (group.4.S):
select Color, sum( Quantity ) as Tot_Qty
from PART, \`ORDER\`
where PART.Number = \`ORDER\`\.Part_Nbr
group by Color
having Tot_Qty > 100000
order by Tot_Qty desc

This is certainly more concise than the Cymbal expression. However, the Cymbal syntax does have its advantages. SQL insists that the grouping quantities be FIELDS from some RECORD_CLASS . This means that a temporary table must be specified in order to group by computed quantities, as illustrated by (grouper.1.Q):

```sql
select Wt_Cat, sum( Quantity ) as Tot_Qty
from ( select
    Number,
    case
    when Weight < 2.0 then "light"
    when Weight < 5.0 then "medium"
    else "heavy"
end
from PART ) as PARTMP( Number, Wt_Cat ),
\`ORDER\`
where PARTMP.Number = \`ORDER\`\.Part_Nbr
group by Wt_Cat
having Tot_Qty > 100000
order by Tot_Qty desc
```

As written, to compute this, first the PART table must be scanned to compute a new (temporary) table which provides the Wt_Cat computed weight categories and then that new table must be joined with the ORDER table. Perhaps the optimizer can rewrite the query so that just one pass over the PARTS is done but the Daytona optimizer does not. On the other hand, the corresponding Cymbal code still computes the query in one pass (grouper.1.Q):
with_format _table_
with_col_labels [ "Wt_Cat", "Tot_Qty" ]
sorted_by_spec[-2]
do Display each_tuple_of
  { [ .wt_cat, sum( over .qty ) ]:
    there_is_a PART where( Number = .part_nbr and Weight = .wt )
    and there_is_an ORDER where( Part_Nbr = .part_nbr and Quantity = .qty )
    and if( .wt < 2.0 )
      then( .wt_cat = "light" )
    else(
      if( .wt < 5.0 )
        then( .wt_cat = "medium" )
      else( .wt_cat = "heavy" ))
    : selecting_when( sum( over .qty ) > 100000 )
  };

14.7 TUPLEs Of Aggregate Functions With Grouping Via BOX-formers

It is a trivial matter to use this same BOX-former syntax to compute multiple aggregate function values. The next query computes the count, average, and standard deviation of the quantities ordered for each part-supplier pair subject to certain selection criteria (cf., grouper.1.Q).

with_format _table_
in_lexico_order
with_col_labels
  [ "Part", "Supplier", "Tot_Orders", "Avg Qty", "Stdev Qty" ]
do Display
each_tuple_of
  { [ .part, .supplier,
    count(), avg( over .qty ), rtn(stdev( over .qty ),.001) ]:
    there_is_an ORDER where( Part_Nbr = .pno and Supp_Nbr = .sno and Quantity = .qty )
    and there_is_a PART where( Number = .pno and Name = .part )
    and there_is_a SUPPLIER where( Number = .sno and Name = .supplier )
    : selecting_when( count() >= 2 and avg( over .qty ) > 1000. )
  };

Since Cymbal variables must have values in order to be worked with, whenever the Quantity field value above is _absent_ then that ORDER tuple is not worked with. Hence, the aggregates computed above are based only on the ORDER tuples for which order quantities are present.

14.8 top-k Queries

A common optimization problem consists of searching through a space of TUPLEs and saving only the best top \( k \) TUPLEs for some given integer \( k \). Typically, _best_ means the top \( k \) (i.e., the first \( k \)
according to some sorting criterion (which would be the bottom $k$ according to the opposite sorting criterion). In other words, when the TUPLEs are sorted from least to greatest using the given sorting criterion, the top-$k$ TUPLEs are the ones whose sort indices are 1 through $k$. As an example, consider the top 100 films of all time sorted by decreasing total gross revenue. The first such film is the one that made the most money and it has sort index 1; the 100th film in the list made so much money that only 99 other films made as much or more. Daytona calls queries that achieve this objective top-$k$ queries. A top-$k$ query is an aggregation query because it takes a collection of TUPLEs and computes a characteristic of them, i.e., the best TUPLEs of the lot, just like the max aggregation function computes a (scalar) maximum of a set of numbers.

### 14.8.1 High-level top-$k$ LISTS/SETs -- with no grouping

The most high-level and consequently the easiest way to express a top-$k$ query is to capture the requisite TUPLEs in a specially configured BOX. It takes using just one new keyword to configure a BOX to be a top-$k$ BOX and that is `keeping_at_most_top` as illustrated by:

```plaintext
set .my_topk_box = [ [.date_placed, .nbr, .qty ]:
    there_is_an ORDERA where( Number = .nbr
        and Date_Placed = .date_placed and Quantity = .qty )
    : with_sort_spec [ 3, 1 ] keeping_at_most_top 40 ];
```

Daytona understands the directive `keeping_at_most_top` $k$ to mean that the box should contain at most the top $k$ elements according to the first sort specification given. In this case, there is only one sort_spec but Daytona will also accept a `with_sort_specs` argument as well. Selection order is not a sort_spec. Also, sort specifications like `with_lexico_order` are unacceptable because they are too broad.

One advantage of implementing top-$k$ for Cymbal by adding just one new keyword is that all the rest of Cymbal is available for reuse. Consequently, the following discussion can afford to be terse because all the other Cymbal constructs employed, especially the BOX-related ones, are discussed elsewhere.

Of course, having built a box, one wants to use it. In this case, that would be as simple as executing:

```plaintext
fet .tu Is_In .my_topk_box sorted_by_spec [ 3, 1 ]
{
    do Write_Words( .tu );
}
```

Often though, one would like to see the results accompanied by row numbers:
fet [.sn, TUPLE[ DATE, INT, INT ] .tu ] ist{
  .tu Is_In .my_topk_box sorted_by_spec [ 3, 1 ]
  with_sort_index_vbl sn
})
do Write_Words( .sn, .tu );

This can all be captured more compactly with nicer formatting by a Display call (top-k.box.1.Q):

set .max_cnt = 12;
with_format _table_
  with_col_labels [ "Sort_Index", "Date_Plcd", "Onbr", "Qty" ]
do Display each_tuple_of
{ [.sn, .date_placed, .nbr, .qty ]:
  .my_topk_box = [ [.date_placed, .nbr, .qty ]:
    there_is_an ORDERA where( Number = .nbr
      and Date_Placed = .date_placed and Quantity = .qty )
    : with_sort_spec [ 3, 1 ] keeping_at_most_top .max_cnt
  ]
  and
  [.date_placed, .nbr, .qty ] Is_In .my_topk_box
  sorted_by_spec [ 3, 1 ] with_sort_index_vbl sn
};

Or even more compactly by (top-k.box.1.Q #1.7),

set .max_cnt = 12;
with_format _table_
  with_col_labels [ "Sort_Index", "Date_Plcd", "Onbr", "Qty" ]
do Display each [.sn, .date_placed, .nbr, .qty ]
each_time( [.date_placed, .nbr, .qty ] Is_The_Next_Where(
  there_is_an ORDERA where( Number = .nbr
    and Date_Placed = .date_placed and Quantity = .qty )
  ) keeping_at_most_top .max_cnt
  sorted_by_spec [ 3, 1 ] with_sort_index_vbl sn
);

Getting the bottom-k in the original ordering can be achieved by asking for the top-k in the reverse order as in:

  sorted_by_spec [ -3, -1 ]

Happily, parallelization comes quite naturally to top-k queries. The key realization is that if each clone returns its top-k individuals and the parent returns the top-k of the individuals in all those top-k lists, then the parent has to have returned the top-k for the entire search space. This is manifestly the best parallelization possible because it could happen that one clone contained the top-k for the entire search space. This strategy is illustrated in top-k.box.1.Q:
set .max_cnt = 10;
set .tot_sects = 12;
with_format _table_
with_col_labels [ "Sindex", "Timestamp", "Src_Addr", "Src_Port", 
                    "Dest_Addr", "Dest_Port", "Pkts_Sent" ]
do Display each_tuple_of
    [ [ .sn, .ts, .sa, .sp, .da, .dp, .pkts ] :
    [ .ts, .sa, .sp, .da, .dp, .pkts ] Is_The_Next_Where(
        .sect_nbr Is_In [ 1 -> .tot_sects ]
        and parallelizing(
            [ .ts, .sa, .sp, .da, .dp, .pkts ] Is_The_Next_Where(
                there_isa NETFLOW from_section[ .sect_nbr, .tot_sects ]
                where(
                    Date_Hour = .ts and 
                    Src_Addr = .sa and 
                    Src_Port = .sp and 
                    Dest_Addr = .da and 
                    Dest_Port = .dp and 
                    Pkts_Sent = .pkts )
            ) sorted_by_spec[ 6 ] keeping_at_most_top .max_cnt
        )
    ) sorted_by_spec[ 6 ] keeping_at_most_top .max_cnt
    with_sort_index_vbl sn parallel_for 4
];

top-k.box.1.Q contains another top-k parallelization query (#4.5) which illustrates taking this approach
on a horizontally partitioned RECORD_CLASS ORDERA, where each job is a BIN. Also, in top-

k.box.1.Q (as query #5), it is shown how the view RECORD_CLASS PARA_TOPK_ORDERA2 is
implemented using parallelization on an hparti RECORD_CLASS where each job is a group of BINs.

14.8.2 High-level top-k LISTs/SETs – with grouping

Top-k shows up in two different ways in group-by queries. The easiest to explain is where the

notion of top-k is applied after the group-by aggregation has been performed, i.e., after the tuples have
been segregated into groups and the aggregate characteristics of those groups have been computed.
Then the goal is to compute the top-k groups, as illustrated by the following which uses a Cymbal
group-by box to Display the top 10 groups by decreasing Quantity (top-k.box.1.Q #3.6):

Copyright © 2013 AT&T
All Rights Reserved.
September 15, 2013
flushing
with_col_labels [ "Sort_Index", "Supplier", "Tot_Qty" ]
with_format _table_
do Display each [ .sn, .supplier, .tot_qty ]
each_time(
    [ .supplier, .tot_qty ] Is_In
    { [ .supplier, sum( over (INT(_huge_)) .qty ) ] :
        there_isa ORDER where( Supp_Nbr = .sno and Quantity = .qty )
        and there_isa SUPPLIER where( Number = .sno and Name = .supplier )
        : keeping_at_most_top 5
    }
    sorted_by_spec [ -2 ] with_sort_index_vbl sn
);

Here is the output:

----------------------------------------
<table>
<thead>
<tr>
<th>Sort_Index</th>
<th>Supplier</th>
<th>Tot_Qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Hannibal Leasing</td>
<td>62271</td>
</tr>
<tr>
<td>2</td>
<td>Thebes Distributing</td>
<td>50638</td>
</tr>
<tr>
<td>3</td>
<td>Nikolaos Rentals</td>
<td>44584</td>
</tr>
<tr>
<td>4</td>
<td>Bouzouki Receiving</td>
<td>43489</td>
</tr>
<tr>
<td>5</td>
<td>Hera Import-Export</td>
<td>42024</td>
</tr>
</tbody>
</table>
----------------------------------------

Note that this is not the same thing as producing the top-k individuals in each group but rather it is producing the top-k groups.

Indeed, that is the essence of the other way for top-k to appear in a group-by setting: after the groups are formed but before aggregate characteristics are computed, Display information on the top-k individuals in each group. So, instead of computing the top-k groups (out of all the groups), this second paradigm computes and displays the desired attributes of the top-k individuals in each group. In other words, this second paradigm is computing a SET- or LIST-valued aggregate function for each group, that being the SET or LIST of the attributes of the top-k individuals. This is in contrast to scalar- or atomic-valued aggregate functions like sum and min. The following query uses a special form of a group-by box to Display attributes of the top-5 members of every group of supplier records (top-k.groupby.1.Q #6):
Here are the first 10 output rows of 99. (Given that there are 20 suppliers, why 99 and not 100? It is because one of the suppliers is only associated with 4 records: remember what is asked for: keeping at most top 5.)

<table>
<thead>
<tr>
<th>Serial</th>
<th>Supplier</th>
<th>Sort_Index</th>
<th>Order_Nbr</th>
<th>Qty_In_10s</th>
<th>Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alexander Import-Export</td>
<td>1</td>
<td>976</td>
<td>488</td>
<td>rotor</td>
</tr>
<tr>
<td>2</td>
<td>Alexander Import-Export</td>
<td>2</td>
<td>421</td>
<td>387</td>
<td>saw</td>
</tr>
<tr>
<td>3</td>
<td>Alexander Import-Export</td>
<td>3</td>
<td>708</td>
<td>284</td>
<td>glue</td>
</tr>
<tr>
<td>4</td>
<td>Alexander Import-Export</td>
<td>4</td>
<td>576</td>
<td>177</td>
<td>wood screw</td>
</tr>
<tr>
<td>5</td>
<td>Alexander Import-Export</td>
<td>5</td>
<td>314</td>
<td>145</td>
<td>AA battery</td>
</tr>
<tr>
<td>6</td>
<td>Ashurbanipal Shipping</td>
<td>1</td>
<td>859</td>
<td>411</td>
<td>bow</td>
</tr>
<tr>
<td>7</td>
<td>Ashurbanipal Shipping</td>
<td>2</td>
<td>675</td>
<td>403</td>
<td>sealant</td>
</tr>
<tr>
<td>8</td>
<td>Ashurbanipal Shipping</td>
<td>3</td>
<td>958</td>
<td>374</td>
<td>cabinet</td>
</tr>
<tr>
<td>9</td>
<td>Ashurbanipal Shipping</td>
<td>4</td>
<td>796</td>
<td>338</td>
<td>eyelet</td>
</tr>
<tr>
<td>10</td>
<td>Ashurbanipal Shipping</td>
<td>5</td>
<td>14</td>
<td>292</td>
<td>rotor</td>
</tr>
</tbody>
</table>

There is a lot to say about this feature-packed query. First, the heart of this query is a Cymbal group-by BOX, where instead of the typical scalar aggregate function call prototypes like `sum( over .x )`, one finds an analog:

```%
%?[ [ .nbr, .qty/10, .part ] :
  : keeping_at_most_top 5 with_sort_spec [ -2 ] preceded_by_sort_index ]
```
have a SET of TUPLES consisting of a supplier and a spliced arbitrary member of a LIST of TUPLES of Order_Nbr, Qty_In_10s, Part. In other words, think of it as a pattern characterizing the desired result. And what is the nature of this LIST of TUPLES? It is the top-5 orders for a given supplier based on the associated assertion, because like all of these Cymbal group-by boxes, the aggregate function specifications are all relative to the (same) assertion, in this case, the one talking about ORDER/SUPPLIER/PART. The optional keyword preceded_by_sort_index merely requests that the sort index for each top-k LIST member appear before the associated attribute values in the output.

The use of the nested TUPLE in the with_col_labels argument is explained in the section on Display options in Chapter 9. The Serial number provided for the entire output is accomplished in the usual fashion for BOXes.

14.8.3 top-k Querying From Basic Concepts

The previous section showed the easy way to compute a top-k result. This section shows how to do so using more primitive concepts which in turn allow additional flexibility. Once again, if the entire search space of TUPLES was stored in a sorted box, it would be easy to produce the top k. However, it can be remarkably inefficient to store all such TUPLES before selecting the best. Fortunately, as the next query illustrates, it is a simple matter to arrange the search so that no more than \( k+1 \) TUPLES are being stored at any given time (top-k.1.Q). This query produces information identifying all those ORDERA whose Date_Placed puts them at the most recent.

```plaintext
local:
  LIST[ TUPLE[ DATE .date_placed, INT .ord_nbr, FLT .qty ]
set [ .max_cnt ] = read( from _cmd_line_ bia[ 10 ] );
for_each_time [ .date_placed, .nbr, .qty ]
  ist(
    there_is_an ORDERA where( Number = .nbr and Date_Placed = .date_placed
      and Quantity = .qty )
  )
  do Change so_that( [ .date_placed, .nbr, .qty ] Is_In_Again .ord_box );
  when( .ord_box.Elt_Count > .max_cnt )
  { do Change so_that_previous( ? Is_Not_In .ord_box with_sort_index 1 );
  }
  fet .tu Is_In .ord_box sorted_by_spec[ 1 ]
  { do Write_Words( .tu );
  }
```

The idea behind the first for_each_time is to visit each candidate record, put it in the sorted box, and then remove the smallest from the box. Recall from Chapter 12 that the do Change so that_previous construction here is to be read as: do Change so that the previous something (i.e., TUPLE) at sort
position 1 in .ord_box is no longer in the box.

Assuming there are no ties on sort keys, a simple induction argument on the number of candidate tuples for the box will prove that at any given point in its construction, the box has at most the $k$ top tuples encountered so far -- and so it does at the end. However, keep in mind that this algorithm does not distinguish among any sort key ties. The guarantee is that the algorithm will provide $k$ answer tuples whose sort key values are extremal in that there are no other candidate answer tuples who have sort key values that are strictly better than any sort key value among the $k$. Obviously, tuples whose sort key values are equal to the smallest among the (final) top $k$ may or may not be in the final box: after all, only $k$ will fit and the others will be excluded depending on when they were encountered as candidates.

For fun, note that essentially the same query can be written using the opposite sort on the box, i.e., a sort_spec sorting from largest to smallest (top-k.1.Q):

```plaintext
local:
  LIST[ TUPLE[ DATE .date_placed, INT .nbr, FLT .qty ]
    : with_sort_spec [ -1 ] ] .ord_box2

for_each_time [ .date_placed, .nbr, .qty ]
  ist(
    there_is_an ORDERA where( Number = .nbr and Date_Placed = .date_placed
      and Quantity = .qty )
  )
  do Change so_that( [ .date_placed, .nbr, .qty ] Is_In_Again .ord_box2 );
  when( .ord_box2.Elt_Count > .max_cnt )
  {
    do Change so_that_previous(
      ? Is_Not_In .ord_box2 sorted_by_spec[ -1 ] with_sort_index .max_cnt+1 );
  }
  flushing with_format _table_ sorted_by_spec[ -1 ]
  with_col_labels [ "Date_Placed", "Ord_Nbr", "Qty" ]
  do Display each_tuple_of .ord_box2 ;
```

Other than output formatting, this version of the query differs from its predecessor by treating ties in a different way and by sorting its output in the opposite order, i.e., from largest to smallest (see next subsection). Note in particular, the second do Change statement uses with_sort_index .max_cnt+1 instead of with_sort_index 1.

Fortunately, there is a somewhat higher level way to express the idiom consisting of the two "do Changes" above and that is:

```plaintext
do Change so_that(
  [ .date_placed, .nbr, .qty ] Is_In_Again .ord_box2 keeping_at_most_top .max_cnt );
```

so that one could write instead:
for_each_time [ .date_placed, .nbr, .qty ]

ist(
    there_is_an ORDERA where( Number = .nbr and Date_Placed = .date_placed
        and Quantity = .qty )
) {
    do Change so_that(
        [ .date_placed, .nbr, .qty ] Is_In_Again .ord_box2
            keeping_at_most_top .max_cnt );
}

Once again, this higher-level keeping at most top do Change will delete as needed the box element with the highest allowed sort index, not the first one in the ordering, as was done in the first example in this section. Actually, this higher-level idiom uses while instead of when for the conditional do Change. The effect of this is to bring the size of the box down to the desired size in one step when (for some reason) the bound is being changed adaptively (and considerably) in the course of a search.

14.8.3.1 top-k by deleting biggest or smallest

This raises the question of how does the strategy of deleting the first compare to that of deleting the highest. The answer is that deleting the highest produces the same answers (but in the opposite order) that deleting the first produces with the reversed order. The #7/7.5 example of top-k.1.Q makes this clear:

for_each_time [ .part, .wt ] ist(
    there_is_a PART_ where( Name = .part and Weight = .wt )
){
    do Change so_that( [ .part, .wt ] Is_In_Again .part_box );
    when( .part_box.Elt_Count > 15 )
    {
        do Change so_that_previous( ? Is_Not_In .part_box with_sort_index 1 );
    }
}

_Say_Nbr(7)

fet [.sn, .tu ] ist(
    .tu Is_In .part_box sorted_by_spec[ 2 ] with_sort_index_vbl sn
){
    do Write_Words( .sn, .tu );
}

for_each_time [ .part, .wt ] ist(
    there_is_a PART_ where( Name = .part and Weight = .wt )
){
    do Change so_that( [ .part, .wt ] Is_In_Again .part2_box );
    when( .part2_box.Elt_Count > 15 )
    {
        do Change so_that_previous( ? Is_Not_In .part2_box with_sort_index .part2_box.Elt_Count );
    }
}

_Say_Nbr(7.5)

fet [.sn, .tu ] ist(
    .tu Is_In .part2_box sorted_by_spec[ -2 ] with_sort_index_vbl sn
){
    do Write_Words( .sn, .tu );
}

This produces the following output:
Note that the two procedures differ according to the 7.9 ties at the point where cut-offs occur.

### 14.8.3.2 top-k sorting by computed quantities

Note also that the user can in effect choose their own metric of goodness since the sorting can be done on the basis of computed quantities, not just plain field values. The following query produces the top longest-taking fulfilled orders.
local:
LIST[ TUPLE[ INT .days_to_fill, INT .nbr, FLT .qty ]
for_each_time [ .days_to_fill, .nbr, .qty ]
   ist(
      there_is_an ORDERA where( Number = .nbr
         and Date_Placed = .date_placed
         and Date_Recd = .date_recd
         and Quantity = .qty )
      and .days_to_fill = .date_recd - .date_placed
   ){
      do Change so_that( [ .days_to_fill, .nbr, .qty ] Is_In_Again .ord_box );
      when( .ord_box.Elt_Count > .max_cnt )
      {
         do Change so_that_previous( ? Is_Not_In .ord_box with_sort_index 1 );
      } } }

Obviously, the computation of days_to_fill here could in general be any computation on any combination of field values taken from any set of tables.

Note the following variant which (approximately) ensures that the top 5% of ORDERA are returned.

set .cand_idx=0;
for_each_time [ .date_placed, .nbr, .qty ]
   ist(
      there_is_an ORDERA where( Number = .nbr and Date_Placed = .date_placed and Quantity = .qty )
   ){
      set .cand_idx++;
      do Change so_that( [ .date_placed, .nbr, .qty ] Is_In_Again .ord_box );
      when( .ord_box.Elt_Count > (INT)(.05*.cand_idx)+1 )
      {
         do Change so_that_previous( ? Is_Not_In .ord_box with_sort_index 1 );
      } } }

Actually, this is not quite right. What this query actually produces is a list of best tuples which, at each point in the computation, is the top 5%. This is not the same as comprising the top 5% of all of the ORDERA tuples because some good tuples may appear early and may not make it into the box due to the low occupancy rate at that time but which would have been eligible for storage if considered in comparison with the totality all at one time. Of course, if the total number of tuples to visit is known in advance then that can be used to determine the true 5% bound.

Obviously, top-k queries enable not only the extremal values (in this case, DATES) to be identified
but also as much information as desired about the data that cause them to be extremal.

14.9 top-epsilon Queries

Related to top-k queries are top-epsilon queries where the goal is to collect all tuples that are within epsilon of the max according to some sort order. This sorting can even be done on computed quantities (top-epsilon.1.Q):
local:
  LIST[ TUPLE[ INT .qty_cat, INT .ord_nbr, DATE .date_placed, FLT .qty ]
    : with_sort_spec [ 1, 4 ] ] .ord_box
  INT .saved_top_qty_cat = -1;
  fet .epsilon Is_In [ 0, 1, 2 ]
  {
    set .ord_box = [];
    for_each_time [ .qty_cat, .nbr, .date_placed, .qty ]
    ist(
      there_is_an ORDERA where( Number = .nbr
       and Date_Placed = .date_placed
       and Quantity = .qty )
      and .qty_cat = ((INT).qty)/100
    )
    {
      // minimizes box work
      when( .qty_cat < .saved_top_qty_cat - .epsilon ) continue;
      do Change so_that( [ .qty_cat, .nbr, .date_placed, .qty ] Is_In_Again .ord_box );
      for_the_first_time [ .low_qty_cat, .top_qty_cat ]
      ist( .ord_box.Elt_Count > 1
        and [.top_qty_cat, 3? ] Is_In .ord_box with_sort_index .ord_box.Elt_Count
        and [.low_qty_cat, 3? ] Is_In .ord_box with_sort_index 1
        and .low_qty_cat < .top_qty_cat - .epsilon
      )
      {
        set .saved_top_qty_cat = .top_qty_cat;
        fet.tu Is_In [ [ .qty_cat, .nbr, .date_placed, .qty ] :
          [ .qty_cat, .nbr, .date_placed, .qty ] Is_In .ord_box sorted_by_spec[1,4]
          and .qty_cat < .top_qty_cat - .epsilon
        ]
      }
      { do Change so_that( .tu Is_Not_In .ord_box ); } 
    }
    fet .tu Is_In .ord_box sorted_by_spec[ 1,4 ] { do Write_Words( .tu ); } 
    do Write_Line( 10**=" );
  }

An important special case of the top-epsilon querie happens when epsilon is zero, i.e., when the objective is to collect information identifying each time that the extremal value is achieved, since in general that may occur more than once. In this special case the logic can be simplified as follows (top-epsilon.2.Q):
local:
   LIST[ TUPLE[ INT .qty_cat, INT .ord_nbr, DATE .date_placed, FLT .qty ]
       : with_sort_spec [ 1, 4 ] ] .ord_box
   INT .max_qty = -1;

for_each_time [ .qty_cat, .nbr, .date_placed, .qty ]
   ist(
       there_is_an ORDERA where( Number = .nbr
           and Date_Placed = .date_placed
           and Quantity = .qty )
       and .qty_cat = ((INT).qty)/100
   ){
       switch_on( .qty_cat )
       {
           case( < .max_qty ){
               continue;
           }
           case( = .max_qty ){
           }
           case( > .max_qty ){
               set .ord_box = [];
               set .max_qty = .qty_cat;
           }
       }
       do Change so_that( [ .qty_cat, .nbr, .date_placed, .qty ] Is_In_Again .ord_box );
   }
for_each_time .tu Is_In .ord_box sorted_by_spec[ 1,4 ]
   {
       do Write_Words( .tu );
   }
15. Macro-Like Devices: Macro Predicates, Views, lambda Opcofuns, Apply

With macro predicates, views, lambda opcofuns, and apply, Cymbal contains a variety of useful constructs, which upon reflection, all have in common a reliance on rewriting Cymbal into Cymbal. As it turns out, the notion of rewriting is widely used behind the scenes in the implementation of Daytona as well but here it is actually visible and definable at the user level.

Views are virtual tables, which means that they are RECORD_CLASSes that are defined declaratively by Cymbal logic assertions. A view table can be used with the same basic syntax that can be used with any RECORD_CLASS. Views turn out to be remarkably useful but before discussing them, it is appropriate first to discuss their simpler compatriot, macro PREDICATES.

This chapter also discusses two other related declarative constructs: arbitrary choice and definite description.

15.1 Macro Predicates

A macro predicate is a declaratively-defined PREDICATE whose definition is given by an assertion that must be true for exactly the TUPLES that satisfy the PREDICATE. Consider (decl.pred.1.Q):

// in the test suite, the Supplies defn is in orders.env.cy
define PRED[.x, .y ] Supplies
eff(
    there_is_a SUPPLIER named .x where( Number = .x_nbr )
    and there_is_an ORDER where( Supp_Nbr = .x_nbr and Part_Nbr = .y_nbr )
    and there_is_a PART named .y where( Number = .y_nbr )
)
when( "Bookbinder Inc." Supplies "muffler" )
{
    do Write_Line( "Bookbinder sells mufflers!" )
}
when( ! Supplies[ "Mercury Distributing", "muffler" ] )
{
    do Write_Line( "Sorry, Mercury Distributing does not sell mufflers" );
}
do Write_Line( "Here are the parts for Bookbinder Inc." );
for_each_time [ .part ] is_such_that( "Bookbinder Inc." Supplies .part ){
    do Write_Line(.part);
}

These examples show how easy it is to define and use a macro predicate. The definition begins by
specifying the type and name of the PRED, including its parameter variables, and then finishes by providing the assertion that it is equivalent to. When the PRED is used, what the system does is to replace the invocation with the defining assertion after the terms in the call have effectively replaced their corresponding parameter variable appearances in the assertion. It’s a little more complicated than that since the parameter variables get new names and since steps are taken to ensure that typing of the parameter variables is honored, including sometimes type-casting the arguments to equal their corresponding parameter. One useful implication is that by means of these new intermediary system-defined variables, each argument function call appears only once in the expanded invocation and so will only be evaluated once. This process is basically an elaborated, type-sensitive macro expansion.

The striking thing about macro predicates is that they really do behave declaratively in that the arguments given to them can, depending on the context of the invocation, result in generating values or testing them. In the example above, the when statements use Supplies for a pure test, whereas the for_each_time generates values for the second argument and tests with the value of the first. This contrasts greatly with procedurally-defined predicates which can only be used to test -- they cannot generate values for variables (except when defined with alias parameters which then implies that those parameters are always aliases).

As the next example illustrates, argument type constraints, keyword arguments and argument defaults can be provided and macro predicate definitions can invoke other macro predicates (decl.pred.1.Q).

```
define PRED: Is_A_Part[ STR .part, with_wt FLT .wt, with_color STR .color ]
   iff(
       there_is_a PART named .part where( Weight = .wt and Color = .color )
   )
define PRED[ STR .part, with_wt .wt = 5.5 ] Is_A_Red_Part
   iff(
       .part Is_A_Part with_wt .wt with_color "red"
   )
for_each_time [.part, .wt ]
ist(
   .part Is_A_Red_Part with_wt .wt
}{
   do Write_Words( .part, .wt );
}
do Write_Line( 25 * "=" );
for_each_time [.part ]
ist(
   .part Is_A_Red_Part
){
   do Write_Words( .part );
}
```

This is a truly interesting query where if the with_wt argument is not provided then a test on Weight results, whereas if it is provided, then it can lead to a generator or a test for Weights. Furthermore, notice that the definition of Is_A_Red_Part makes use of the declaratively defined Is_A_Part.
For convenience, macro predicate definitions can be put into Cymbal *.env.cy environmental files.

15.1.1 Handling Outside Variables In Macro Predicates

The macro flavor of macro predicates enables them to use outside variables and assertion components that can’t stand alone by themselves. Consider (decl.pred.1.Q):

\[
\begin{align*}
\text{define PRED[ ] This_Is_A_Big_Order} \\
\text{iff(} \quad \text{this_isa ORDER where( Quantity > .cutoff )} \\
\text{)} \quad \text{using( outside[ INT .cutoff ] )}
\end{align*}
\]

\[
\begin{align*}
\text{local: INT .cutoff = 3000} \\
\text{fet .order_nbr ist(} \\
\text{\quad there_is_an ORDER where(} \\
\text{\quad \quad Number = .order_nbr where( .order_nbr \% 200 = 1) } \\
\text{\quad \quad and This_Is_A_Big_Order) } \\
\text{\quad \{} \\
\text{\quad \quad _Show_Exp_To(.order_nbr) } \\
\text{\quad \})
\end{align*}
\]

Note the mandatory use of an outside TUPLE of VblSpecs (see Chapter 6) to declare the use of outside VBLs: these helpful specifications enable the reader of the definition to understand the nature of that definition just in terms of itself, i.e., not needing to refer to the context of any possible invocations. In particular, the use of outside makes it clear which variables are outside VBLs and which are implicitly scoped local VBLs. These outside VBLs may be locals, exports, imports or declarative. A follow-on benefit is that there can be no surprises where what is intended to be an implicitly scoped local VBL is inadvertently bound by an outside VBL of the same name when a call to the PRED is issued in the program: this cannot happen by virtue of this requirement.

And this usage is indeed mandatory because if an outside VBL is used in the definition but is not specified in an outside argument, then the user will receive cryptic errors like this:

\[
\begin{align*}
\text{error: unable to handle a certain satisfaction claim with free variables} \\
\text{...} \\
\text{The questionable satisfaction claim is:} \\
\text{Quantity > .This_Is_A_Big_Order_PRED_Local_Standin_Vbl_1_For_cutoff}
\end{align*}
\]

The tip-off is that the intended outside VBL cutoff has become a PRED_Local_Standin_Vbl.

15.1.2 Perspectives On Macro Predicates

An important caveat: macro predicates cannot be defined recursively, e.g., with one or more uses of a predicate inside its own defining assertion. However, some declarative linear recursive predicate definitions are supported by means of the path predicates discussed in Chapter 16.
For readers who know about logic programming, a Cymbal macro PRED is a generalization of a Prolog definition of a predicate. Consider:

\[
\begin{align*}
P(x,y) & \leftarrow S(x,y) . \\
P(x,y) & \leftarrow A(x,u), B(y,u) . \\
P(x,y) & \leftarrow C(u,x), D(y,u) . \\
P(x,y) & \leftarrow E(x,u), F(u,y) .
\end{align*}
\]

In Cymbal, this definition would be represented using this definition:

\[
\text{define PRED[ .x, .y ] Pp } \iff \begin{align*}
& Ss(.x, .y) \\
& \text{or } (Aa(.x, .u) \text{ and } Bb(.y, .u)) \\
& \text{or } (Cc(.u, .x) \text{ and } Dd(.y, .u)) \\
& \text{or } (Ee(.x, .u) \text{ and } Ff(.u, .y)) \\
\end{align*}
\]

Cymbal’s construct is more general because the defining assertion can be any Cymbal assertion (that Daytona can process). For example, this would include both kinds of quantifiers, negation, aggregate functions, associative arrays, set-formers, uses of path PREDs, etc. Also, the Cymbal definition is an indivisible syntactic unit, whereas the Prolog version is just a collection of statements that may or may not be collected together syntactically.

Incidentally, Cymbal is justified in using \textit{iff} in contradistinction to the implication used by Prolog. The reason is because the following series of statements are equivalent:

\[
A(x) \iff B(x) \\
(\text{if } A(x) \text{ then } B(x)) \text{ and } (\text{if } B(x) \text{ then } A(x)) \\
(\text{if } A(x) \text{ then } B(x)) \text{ and } (\text{if } ! A(x) \text{ then } ! B(x))
\]

As seen in the first example of this section, Daytona is perfectly happy to work with the negation of a ground \textit{Supplies} satclaim. It processes that by seeking to prove the assertion that is the expansion of the \textit{Supplies} definition and it can’t do it; so it assumes that expansion assertion must be false, i.e., its negation is true and so by the \textit{iff} definition, the negation of the ground \textit{Supplies} satclaim is true. The underlying assumption here is that when Daytona has some specific facts about what is true, then it has all the true facts of that kind; consequently, any other purported fact of that kind must be false. For example, the assumption behind the SUPPLIER table in the test suite is that it contains information about all of the SUPPLIERS; consequently, since there is no record for \textit{General Aviation}, \textit{General Aviation} must not be a SUPPLIER. In this regard, recall that the meta-function \textit{truth} will map any ground assertion Daytona accepts to either \texttt{true} or \texttt{false}.

(This is all related to what the literature calls the \textit{closed-world assumption} or \textit{negation as failure}, where, roughly, if a ground assertion cannot be proved to be true in some deductive way, then it must be false, i.e., this failure to construct a proof \textit{amounts to} a proof that its negation is true. The major convenience of all these assumptions is that they preclude the necessity of storing a huge number of explicit facts about what is \textit{not} true -- because what is not true can be inferred by what cannot be proved.)
15.1.3 Perspectives On The English Monarchy Via Macro Predicates

The Daytona test suite contains truly fascinating data about the MARRIAGEs of English ROYALs from William The Conqueror to Elizabeth I. This is fertile ground for creating interesting macro predicates. Arguably, Daytona’s MARRIAGE table is a better and more natural way to model this information than the typical parent(x,y) relations found in the literature. Here is (part of) the schemas for these two tables as produced by Synop:

RECORD_CLASS: MARRIAGE
Field 1: STR Husband
2: STR Wife
3: INT Year_Married
4: SET( (0->) STR ) Children

RECORD_CLASS: ROYAL
Field 1: STR Name
2: STR Title
3: STR Father
4: STR Mother
5: STR Sex
6: INT Year_Born
7: INT Accession_Year
8: INT Year_Died

Indeed, here is the analog of a parent(x,y) relation (daytona.env.cy):

```define PRED[ STR .x, STR .y ] Is_The_Royal_Father_Of iff
   ( there_isa MARRIAGE where( Husband = .x and one_of_the Children = .y ) )
define PRED[ STR .x, STR .y ] Is_The_Royal_Mother_Of iff
   ( there_isa MARRIAGE where( Wife = .x and one_of_the Children = .y ) )
define PRED[ STR .x, STR .y ] Is_A_Royal_Parent_Of iff
   ( .x Is_The_Royal_Mother_Of .y )
   or
   ( .x Is_The_Royal_Father_Of .y )
)
```

The history of the English monarchy is sufficiently complex that it is necessary to account for half-siblings (decl.genea.1.Q).
define PRED[ TUPLE[ STR .x, INT .yob ] .sib_info, STR .y ]
    Is_A_Royal_Half_Sibling_Of
iff {
    there_isa MARRIAGE where( 
        Husband = .h and Wife = .w and one_of_the Children = .y )
    and there_isa MARRIAGE where( Husband = .h and Wife != .w 
        and one_of_the Children = .u )
    and there_isa ROYAL where( Name = .u and Year_Born = .yob )
    and .sib_info = [ .u, .yob ]
}
fet .tu ist( .tu Is_A_Royal_Half_Sibling_Of "Elizabeth I" )
{
    _Show_Exp_To(.tu)
}

Recall that Elizabeth I was a daughter of Henry VIII who had many wives. Here is the answer to this query (as supported by the data stored, if not history, which tends to pass over the many children who died early in those times):

    .tu = Mary I 1516
    .tu = Edward VI 1537

Daytona does make use of the types of the parameter VBLs in a macro PRED definition. In this case, since .tu has not been explicitly given a type, Daytona infers a type for it by assigning it the TUPLE type of the .sib_info parameter.

15.1.4 The Types Of Macro Predicates Do Matter

In general, the types of the parameter VBLs in a macro PRED definition are handled as follows. (Behind all the technicality here is the simple premise/promise that if the user specifies that a parameter VBL has a specific type (i.e., not involving OBJ), then Daytona will ensure that the defining assertion is processed with that being the case.)

First, consider the case of the argument being a VALCALL. If the parameter VBL type is OBJ VBL, then without further ado when expanding a macro invocation, the parameter VBL itself is everywhere replaced in the defining assertion with the argument VBL. Otherwise, assume the parameter VBL type does not contain OBJ at all and is not a UNION type: in other words, it is a completely specific type. Then if the argument VBL is an OBJ VBL, it gets the (specific) type of the parameter VBL instead and the same VBL-to-VBL name substitution happens. Otherwise, if the argument VBL type does not contain OBJ at all, then if it matches the parameter VBL type exactly (or exactly matches one of the UNION member CLASSESes), then the same VBL-to-VBL name substitution happens. Any other case is an error.

Second, suppose the argument is a non-VALCALL term like a FUNCALL. Then if the parameter VBL has a specific type involving neither OBJ nor UNION, the parameter VBL value is asserted in the expansion to be equal to the argument term casted to the type of the parameter VBL values -- so that cast better work. Otherwise, the parameter VBL value is asserted to be equal to the argument term.
Consider this example of a macro PRED intended to determine whether or not two BOXes are the same:

```plaintext
define PRED[ .s1, .s2 ] My_Set_Eq
  iff( .s1.Elt_Count = .s2.Elt_Count
       and for_each .x such_that( .x Is_In .s1 )
         conclude( .x Is_In .s2 ) )
set .b1 = { "a", "b" };
when( .b1 My_Set_Eq { "b", "a" } ) do Write_Line( "Success!" );
```

This program writes out Success!. Note that, according to the rules above when the parameter VBL type is OBJ VBL, My_Set_Eq can be used for any SET regardless of the types of its elements or any BOX keywords that may be used in its definition. My_Set_Eq can only be used however when its arguments are not generating occurrences for VBLs. Daytona itself uses the more general Eq_Set, defined in sys.env.cy, that can be used in all situations by virtue of its reliance on Ground_In_Use. Likewise, Daytona uses Eq_List: see how they differ.

Here is another example:

```plaintext
define PRED[ INT(_short_) .x, DATE(_DDDMMMddyyyy_) .y, DATE(_ddMMMyyyy_) .z ]
  Scheduled_Date
  iff( .z = .y + .x )
set .bump = 31; // bump is an INT(_short_) VBL
for [ .dd ] list( // dd is an OBJ VBL
// dd is a generator in each disjunct
  Scheduled_Date[ 23+.bump, "2007-08-08", .dd ]
  or Scheduled_Date[ .bump, "2007-08-08"+10, .dd ]
}{ _Show_Exp_To(.dd) }
```

decl.pred.6.Q and decl.pred.8*.Q contain more examples. They are well worth reading.

### 15.1.5 Algorithms As Macro Predicates

Lastly, macro PREDs provide a straightforward way to implement the computation of greatest common divisors. This highlights the power of declarative programming because an examination of this program will reveal that it is just a definition/description of what it means to be a greatest common divisor -- and yet it can be used to compute them directly (decl.gcd.2.Q)!
define PRED[ INT .d, TUPLE[ INT, INT ] .pair ] Is_A_Common_Divisor_Of
iff(
   if( .pair#1 = 0 or .pair#2 = 0 )
   then( .d = max(abs(.pair#1), abs(.pair#2)) )
   else( 
      .d Is_In [ 1 -> min(abs(.pair#1), abs(.pair#2)) ]
      and .pair#1 % .d = 0
      and .pair#2 % .d = 0
   )
)
define PRED[ INT .d, TUPLE[ INT .i, INT .j ] .pair ] Is_The_Gcd_Of
iff( 
   .d Is_A_Common_Divisor_Of[ .pair#1, .pair#2 ]
   and there_does_not_exist INT .e such_that( 
      .e Is_A_Common_Divisor_Of[ .pair#1, .pair#2 ] and .e > .d
   )
)
for_each_time [ TUPLE[ INT, INT ] .tu, .gcd ]
is_such_that( 
   .tu Is_In [ [ 1234, 5678 ], [ 987654321, 123456789 ] ]
   and .gcd Is_The_Gcd_Of .tu
)
}{
   do Write_Words( .gcd, .tu )
}

And here are the answers:

2 1234 5678
9 987654321 123456789

This is not a very efficient algorithm; Euclid’s recursive algorithm is much faster and will be presented in the next chapter on path PREDs. So, some logic programs are slower than others at solving the same problem, just like some C programs are. Likewise, a slow logic program can sometimes be made faster by a little tweak. When running this example, the user will notice that the program prints out the second answer -- and then continues for some seconds after that. What happened is that after it found the gcd, it continued searching for even larger common divisors even though it had just proven that there weren’t any. Obviously, it didn’t know when to quit. However, it is easy enough to tell it when to quit (decl.gcd.3.Q):
define PRED[ INT .d, TUPLE[ INT, INT ] .pair ] Is_The_Gcd_Of
  iff(
    .d Is_The_First_Where(
      .d Is_A_Common_Divisor_Of[ .pair#1, .pair#2 ]
      and there_does_not_exist INT .e such_that(
        .e Is_A_Common_Divisor_Of[ .pair#1, .pair#2 ] and .e > .d
      )
    )
  )

It’s the use of Is_The_First_Where that tells it quit after it provably finds the solution. Notice that this is a small syntactic change that still does not obscure the basic logic here. This use of Is_The_First_Where to terminate search (early) on success is similar to the way cut can be used in Prolog. It also has the same flavor as the (arbitrary) choice construct used by some other logic programming systems, although Is_The_First_Where makes a clear statement as to which of several possibilities will be chosen.

15.2 Ground In Use

Macro PREDs and views are declarative constructs whose arguments when the construct is invoked can be either ground or free. This support of arguments either ground or free at invocation often works without further ado but sometimes the definition of a macro PRED or view needs to be able to break the analysis of each argument down into those two cases. The Ground_In_Use construct makes that possible.

PREDICATEs express relationships among objects. Consider this relationship between two FLTs where one is the square root of the other:

define PRED[ FLT .x, FLT .sqx ] Is_Sqrt_Of
  iff(
    .x = sqrt(.sqx)
  )
do Display each[ .x ] each_time( .x Is_Sqrt_Of 36 );

While the intent is to relate two numbers if and only if if the first is the square root of the second, this PRED does not behave symmetrically in use in that, while Daytona will process the query above and print the square root of the second number, 36, Daytona will not allow this PRED to produce the square of the first number, if that is the one provided instead. In other words, trying to process this query:

do Display each[ .y ] each_time( 6.0 Is_Sqrt_Of .y );

will result in this compilation error:

error: one of the subjects in an equality assertion contains variables \with indeterminable values. ...
The questionable subject is:
sqrt( .y )
Here, Daytona needs to generate a value for $y$ and unfortunately, the given satclaim $6.0 = \sqrt{.y}$ is not of a form that Daytona will accept for generating VBL values. If only the satclaim were the equivalent $6.0 \times 6.0 = .y$; then that would tell Daytona exactly how to compute the value for $y$. Of course, if the satclaim in the definition were $.x \times .x = .y$ instead, then the first query would not work for the same reason. So, is it possible to have one’s cake and eat it too (in this context)?

The answer is yes; the answer is to use the special 0-argument $\text{Ground\_In\_Use}$ $\text{PRED}$ as in (decl.pred.7.Q):

\begin{verbatim}
define PRED[ FLT .x, FLT .sqx ] Is_Sqrt_Of
  iff(
    if( Ground_In_Use )
    then( .sqx = .x *.x )
    else( .x = sqrt(.sqx) )
  )
\end{verbatim}

The implied subject of the $\text{Ground\_In\_Use}$ satclaim is the dereference of one of the parameter VBLs for the associated macro $\text{PRED}$ and that dereference, $.x$ in this case, is the one all by itself on one side of an equality in the $\text{else}$ assertion that either is the whole $\text{else}$ assertion or else is the last conjunct in the $\text{else}$ assertion. ($\text{Ground\_In\_Use}$ is also supported when the assertion identifying the “implied subject” is an $\text{Is\_In}$ assertion with the subject of that assertion being the $\text{Ground\_In\_Use}$ "implied subject" (vu.box.2.Q).) In this case, the macro $\text{PRED}$ definition is essentially:

\begin{verbatim}
define PRED[ FLT .x, FLT .sqx ] Is_Sqrt_Of
  iff(
    if( /* x is */ Ground_In_Use )
    then( .sqx = .x *.x )
    else( .x = sqrt(.sqx) )
  )
\end{verbatim}

The way to read this is, if $x$ is ground in the use of $\text{Is\_Sqrt\_Of}$, then when expanding the macro $\text{PRED}$, replace the $\text{if-then-else}$ with the $\text{then}$’s assertion; else replace it with the $\text{else}$’s assertion. Note that the $.x$ is ground when $\text{Is\_Sqrt\_Of}$ is used like $6.0 \text{ Is\_Sqrt\_Of } .y$ and it may or may not be ground when $\text{Is\_Sqrt\_Of}$ is used like $.x \text{ Is\_Sqrt\_Of } 36$. The case when both arguments are ground works fine because then the satclaim is a test, implying that there is no need to generate a value for a VBL, hence either the $\text{then}$ or $\text{else}$ case would offer satisfactory code. The only case that won’t work now is the one that shouldn’t work, i.e., when both arguments are not-ground VBL dereferences because that amounts to asking Daytona to use this assertion to produce all possible pairs of numbers and their square roots.

Note that, strictly speaking, this use of $\text{if-then-else}$ and $\text{Ground\_In\_Use}$ is not typical Cymbal. If it were typical Cymbal, then both the $\text{then}$ and $\text{else}$ assertions would have to be able to generate values for the same VBLs (assuming they had not-ground VBL occurrences). That is clearly not the case here. So, the true nature of this construct is that it is a metalanguage directive that specifies how to expand the corresponding macro $\text{PRED}$ (or view) into typical Cymbal by replacing the $\text{if-then-else}$ with either the $\text{then}$ and $\text{else}$ assertions.

One of the more common uses of $\text{Ground\_In\_Use}$ is to enable the same bit of Cymbal to use...
keyed retrieval from RECORD_CLASSES when needed and if not so needed, then to compute values from the KEY FIELDS (as opposed to insisting that they have certain values as is done with keyed retrieval) (decl.pred.7.Q):

```plaintext
define PRED[ STR .fixed_val, UINT .uint_val ] Is_Smorgas_Related_To
  iff(
    there_is_a SMORGAS where(
      F_Heka_Fixed = .heka_val
      and F_Unt = .uint_val
    ) and
    if( Ground_In_Use )
      then( .heka_val = (HEKA(5)) .fixed_val )
      else( .fixed_val = (STR).heka_val )
  )
```

Here if .x is ground when using .x Is_Smorgas_Related_To .y, then the ground value of x will be converted to HEKA(5) in order to support the use of the INDEX on the presumed KEY FIELD F_Heka_Fixed in looking up a record to use in generating or testing a value for y; otherwise, access to SMORGAS in order to handle the there_isa will not use that index but rather will pull the required value out of the F_Heka_Fixed FIELD and convert it to a STR value for x.

Here is a nested use of Ground_In_Use in defining a PRED relating the x-y coordinates of a point on a circle with its radius r (decl.pred.7.Q). The effect is that this PRED will compute the value(s) of any one of the three quantities given values for two of them. Note the reliance on the convention that the Ground_In_Use subject VBL has to be identifiable from an equality (or Is_In assertion) that is the last conjunct in the else assertion (or else that equality (or Is_In assertion) is the entire else assertion itself). The advantage of this convention is that it enables an arbitrary preceding sequence of conjuncts to define what the value(s) for the VBL are in that else case.
```c
#define PRED[ FLT .x, FLT .y, FLT .r ] On_A_Circle2_With_Radius
iff(
    if( Ground_In_Use )
    then ( 
        if( Ground_In_Use )
        then( .r = sqrt( .x*.x + .y *.y ) )
        else( .y0 = sqrt( .r*.r - .x*.x )
                and (.y1 = .y0 or .y1 = -.y0)
                and .y = .y1
        )
    ) else ( 
        .x0 = sqrt( .r*.r -.y*.y )
        and (.x1 = .x0 or .x1 = -.x0)
        and .x = .x1
    )
)
```

```
fet .y ist( On_A_Circle2_With_Radius[ -3, .y, 5 ] )
{ _Show_EXP_To(.y) }

// here is the output:
.y = 4.0
.y = -4.0
```

In general, Ground_In_Use supports defining all $2^n$ binding patterns of ground and free for an $n$-place predicate although as the example above shows, that is not always what makes sense: in this case, knowing the value for one of the three does not make it feasible to compute the values for the other two. On the other hand, if Ground_In_Use was being used to support an interface to a table with $n$ FIELDS, then it would not be prima facie unreasonable to support all $2^n$ binding patterns, although even then it may be possible to avoid the tedium of enumerating code for all $2^n$ possibilities if default behavior is acceptable for some patterns.

### 15.3 Views

A **view** is a RECORD_CLASS whose definition is given declaratively by a Cymbal assertion. This means that characterizing whether or not a RECORD is in that view is determined by the truth or falsity of an assertion about it. In this sense, a view is a virtual table, i.e., a table defined by what amounts to be a query. Consequently, it is appropriate to think of a view as a virtual RECORD_CLASS and correspondingly, to think of a "traditional" RECORD_CLASS (that is implemented by explicit lists of records) as a physical RECORD_CLASS. Note that Daytona physical RECORD_CLASSES can not only be implemented by files of records on disk but also through programs that create records to be piped to Daytona. The difference is precisely that of intension (i.e., view) versus extension (i.e., traditional RECORD_CLASSES): likewise, a BOX can be defined by a membership assertion whose truth characterizes the elements of the BOX or else by an explicit listing of BOX’s elements. Note that Synop works with both kinds of RECORD_CLASSES.

Anyway, in effect, the implementation processes a view by replacing a `there_is` for the view
RECORD_CLASS with its defining assertion. Notice that a there_isa for a RECORD_CLASS behaves similarly to a satclaim for a PRED in assertions and that both views and macro PREDs have a macro kind of character. However, views differ from macro PREDs in several ways:

1. the user works with views using as much of the Cymbal there_isa syntax (as well as SQL table/column syntax) as could reasonably be expected instead of using the usual Cymbal syntax for PREDs;
2. the system optimizes view use by removing any useless/pointless computation;
3. due to their use of there_isa syntax, views can also be used to some extent and with careful thought to update tables;
4. there is the currently unrealized potential of supporting materialized views;
5. the fully specified typing of view parameter VBLs is required.

Views are remarkably and surprisingly useful. Here is the current list of the functionality they offer to users.

- Generalized horizontal partitioning, whereby any criteria can be used to partition groups of rows into files. Such criteria include defining partitions by enumerated sets of values, by ranges, by hashed assignment, by round-robin assignment, and in general, by having a specified value for a given partitioning function mapping one or more field values per physical record to partitions.

- Vertical partitioning, which implements a table by storing sequences of fields/columns each in their own separate files. This supports easily adding new fields to a table as well as segregating large and/or infrequently accessed fields from their counterparts.

- Both horizontal and vertical partitioning simultaneously.

- Nested tables, where in effect, a FIELD of the view functions as a nested table.

- Computed fields, which are virtual fields that are computed at run-time from the values of other fields.

  This includes LIST/SET-valued FIELDS for SQL: whereas SQL only supports atomic FIELD values but Cymbal supports both LIST and SET FIELDS, views offer the ability to package up Cymbal LIST/SET FIELD values into virtual SQL (atomic) STR FIELDS.

- Variant fields, where the contents of the same (field) position in a record are interpreted differently in terms of name and type across different records according to the computed category of the record.

- Chain variant records, where the view discriminates among different underlying base record classes that form a chain, i.e., a totally ordered arrangement where the fields of any record class in the chain are an initial subsequence of the fields of any successor. In this case, there is a single view RECORD_CLASS that makes use of multiple base RECORD_CLASSes defined over the same data files. In this situation, views provide support for the completely transparent and
simple addition of new FIELDS to a RECORD_CLASS schema without making any physical changes to the existing data, indices or queries and while supporting the indexing of any field and even the intermingling of records with differing numbers of fields in the same file. In other words, the only changes are to metadata; it's like magic.

• Branching variant records, where there are multiple distinct view RECORD_CLASSes each of which is defined in terms of its own base RECORD_CLASS but for which all base RECORD_CLASSes share an initial sequence of the FIELDS whose values, among other things, serve to distinguish one variant from another.

• Parallelized SQL (and Cymbal) table access, while hiding the Cymbal parallelization strategy used. This provides an effective way to parallelize SQL queries by simply using a behind-the-scenes parallelized view table as the first table in the FROM clause.

• In-memory tables, where the same flexible "there_isa" query syntax is used for the in-memory boxes underlying the view as it is for disk-based tables.

• Data abstraction, where end-user query syntax is isolated from changes in the (implementation of) underlying physically-stored tables. This is known as logical-physical data independence.

• Query abstraction, where complicated query logic involving group-by’s, joins, parallelization, etc. is hidden from the user of the view table.

• Authorization restrictions, whereby only a subset of a base table’s fields is offered for access by the view table.

15.3.1 A Pedagogical View

Here is a completely artificial view definition that paradoxically captures the essence of views (vu.box.2.Q):
// Defining the view
define RECORD_CLASS SQ_SQRT as_a_view_where(
  for_each [ INT .nbr, INT .sq, FLT .rt ]
  conclude(
    there_isa SQ_SQRT where(
      Number = .nbr and
      Square = .sq and
      Sqrt = .rt )
    iff(
      .nbr Is_In [ 1 -> 100 ] and
      .sq = .nbr * .nbr and .rt = sqrt( .nbr ) )
  ))
// Using the view
select Number, rtn(Sqrt, .001) // round_to_nearest
from SQ_SQRT where Number in ( 2, 16, 96, 339 );
with_format _table_
do Display each[ .nbr, .sq, .rt ]
each_time( there_isa SQ_SQRT where( Number = .nbr which_is > 40 & < 55
  and Square = .sq and Sqrt = .rt ) );

The intent of this definition is to define a virtual table SQ_SQRT that can be queried using SQL and Cymbal. Ironically, this SQ_SQRT view table has nothing to do with (persistent or even temporary) data on disk and yet SQL can be used to "query" it. (In one sense, it can't be any more virtual than this -- there is simply no physical (i.e., disk) manifestation of any data here.) Note that it is the for_each equivalence assertion that characterizes what it means for a TUPLE to be a member of this RECORD_CLASS. As with all Cymbal, this definition should be read as mathematical English at which point its meaning should be clear.

Note that the for_each VBL LIST provides the fully specified types for the FIELDS of SQ_SQRT. Furthermore, the expected (and non-trivial) use of there_isa syntax is supported as well as the use of SQ_SQRT and its FIELDS as a regular SQL-type table and columns. Lastly, the implementation is smart enough not to calculate the squares of the numbers requested by the SQL SELECT above since the user's query is not asking to produce them.

A limitation of the SQ_SQRT view is that when .nbr is known, the view can only be used to compute squares and square roots when By using Ground_In_Use for Is_In, that limitation can be eliminated (vu.box.2.Q):
define RECORD_CLASS SQ_SQRT4 as_a_view_where(
  for_each [ INT .nbr, INT .sq, FLT .rt ]
  conclude(
    there_isa SQ_SQRT4 where(
      Number = .nbr and
      Square = .sq and
      Sqrt = .rt )
  iff(
    if( Ground_In_Use )
    then( .nbr = .nbr
    ) else ( .nbr Is_In [ 1 -> 100 ]
    )
    and .sq = .nbr * .nbr and .rt = sqrt( .nbr )
  ))
)

15.3.2 A Simple Database View: Vertical Partitioning

The next view shows how easy it is to do vertical partitioning in Cymbal, which is where the view table is logically cut vertically into base tables corresponding to groups of columns.

define RECORD_CLASS VPARTV as_a_view_where(
  for_each [ INT .nbr, STR .name, STR .color, FLT .weight ]
  conclude(
    there_isa VPARTV where(
      Number = .nbr and
      Name = .name and
      Color = .color and
      Weight = .weight )
  iff(
    there_isa PART_1 where(
      Number = .nbr and
      Name = .name )
    and
    there_isa PART_2 where(
      Number = .nbr and
      Color = .color and
      Weight = .weight )
    ))
  using( all_keyed_record_existence_tests_pass );
)

select Name from VPARTV where Name Matches "^[a-c]" order by 1;

Here PART_1 and PART_2 are regular RECORD_CLASSes for which Number is assumed to be a Unique KEY (vu.vparti.4.Q). Obviously, some of VPARTV’s fields are being stored in one base table.
PART_1 and the others in the other (PART_2). It is the Unique KEY Number that links them together. Furthermore, there is no reason why an indefinite number of vertical partition base tables cannot be included in the definition.

Note that the query in the example only makes reference to the Name field. In this case, one would only want the first base table PART_1 to be visited even though PART_2 is part of the view. As instructed by the all_keyed_record_existence_tests_pass keyword in the optional using specification, Daytona makes sure that that is the case. Specifically, as a matter of course, references to FIELDS that are not used are automatically removed in Tracy’s processing of the view, which in this case would leave the keyed record existence test for a given .nbr:

\[
\text{there\_isa PART\_2 where( Number = .nbr )}
\]

As instructed by the keyword, this assertion is then removed. Note that this keyword is saying that such keyed record existence tests are pointless if in fact there is a referential integrity constraint that requires that any foreign key be defined in another table. How Daytona processes views, including these simplification processes, will be elaborated right after this section on vertical partitioning.

By the way, it is perfectly permissible to use SQL in the definition of a view, because Daytona always expansively considers SQL to be a subset or dialect of Cymbal.

\[
\text{define RECORD\_CLASS VPARTV as_a\_view\_where(}
\text{for\_each [ INT .nbr, STR .name, STR .color, FLT .weight ]}
\text{conclude(}
\text{there\_isa VPARTV where(}
\text{Number = .nbr and}
\text{Name = .name and}
\text{Color = .color and}
\text{Weight = .weight )}
\text{iff(}
\text{[ .nbr, .name, .color, .weight ] Is\_Selected\_By}
\text{ )[ select P1.Number, P1.Name, P2.Color, P2.Weight}
\text{from PART\_1 as P1, PART\_2 as P2}
\text{where P1.Number = P2.Number ]}
\text{ )})
\text{select * from VPARTV where Number <= 105;}
\]

As always, Is\_Selected\_By is used to avoid the overhead of making a box, as would occur if Is\_The\_Next\_Where or Is\_Something\_Where were used.

### 15.3.2.1 Vertical Partitioning: Nested Tables

For the moment though, consider this example of what would happen in this vertical partitioning situation if the KEY Number were NonUnique (usr.env.cy).
define RECORD_CLASS DETAILED_PART
as_a_view_where(
    for_each [ INT .number, STR .name, STR(25) .color, FLT .wt,
               INT .onbr, STR(30) .supplier, INT .qty ]
    conclude(
        there_isa DETAILED_PART where(
            Number = .number and
            Name = .name and
            Color = .color and Weight = .wt and
            Order_Nbr = .onbr and
            Order_Supplier = .supplier and
            Order_Qty = .qty )
        iff (
            there_isa PART where(
                Number = .number and
                Name = .name and
                Color = .color and Weight = .wt
            ) and
            there_isa ORDER where(
                Part_Nbr = .number and
                Number = .onbr and
                Quantity = .qty and
                Supp_Nbr = .snbr
            ) and
            there_isa SUPPLIER where(
                Number = .snbr and
                Name = .supplier )
        ) )
)

In this DETAILED_PART view, a PART Number links to zero or more ORDER records, each of which itself is linked to exactly one SUPPLIER record by Supp_Nbr (so as to look up the SUPPLIER Name). Here is what a portion of the view table would look like when selected:
select * from DETAILED_PART where Number in ( 123, 147, 177 );

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Color</th>
<th>Weight</th>
<th>Order_Nbr</th>
<th>Order_Supplier</th>
<th>Order_Qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>button</td>
<td>orange</td>
<td>5.9</td>
<td>53</td>
<td>Mountain Export</td>
<td>1961</td>
</tr>
<tr>
<td>123</td>
<td>button</td>
<td>orange</td>
<td>5.9</td>
<td>85</td>
<td>Nikolaos Rentals</td>
<td>3313</td>
</tr>
<tr>
<td>123</td>
<td>button</td>
<td>orange</td>
<td>5.9</td>
<td>309</td>
<td>Tutankhamen Rentals</td>
<td>2989</td>
</tr>
<tr>
<td>123</td>
<td>button</td>
<td>orange</td>
<td>5.9</td>
<td>413</td>
<td>Tiryns Import</td>
<td>3857</td>
</tr>
<tr>
<td>123</td>
<td>button</td>
<td>orange</td>
<td>5.9</td>
<td>434</td>
<td>Electra Receiving</td>
<td>3068</td>
</tr>
<tr>
<td>147</td>
<td>shelf</td>
<td>green</td>
<td>9.5</td>
<td>64</td>
<td>Apollo Sales</td>
<td>1792</td>
</tr>
<tr>
<td>147</td>
<td>shelf</td>
<td>green</td>
<td>9.5</td>
<td>95</td>
<td>Sunshine Warehouse</td>
<td>924</td>
</tr>
<tr>
<td>147</td>
<td>shelf</td>
<td>green</td>
<td>9.5</td>
<td>461</td>
<td>Knossos Export</td>
<td>3713</td>
</tr>
<tr>
<td>177</td>
<td>belt</td>
<td>khaki</td>
<td>9.9</td>
<td>166</td>
<td>Ra Import-Export</td>
<td>2940</td>
</tr>
<tr>
<td>177</td>
<td>belt</td>
<td>khaki</td>
<td>9.9</td>
<td>300</td>
<td>Ephesus Sales</td>
<td>115</td>
</tr>
</tbody>
</table>

Notice the field values that are specific to the PART (like Color values) are repeated for each tuple of ORDER/SUPPLIER information. What if in addition there were multiple INVOICEs per PART as well? If one just added an additional `there isa` conjunct for the INVOICEs, then each ORDER/SUPPLIER pseudo-tuple would be paired with each INVOICE pseudo-tuple in a Cartesian product manner, and thus the information on each PART would be further replicated in displays of DETAILED_PART records such as the above. This kind of redundancy is undesirable -- and avoidable.

The view mechanism can be used to create a LIST of TUPLEs of ORDER/SUPPLIER information for each PART, i.e., each PART can have a LIST[TUPLE] FIELD for representing all its ORDER/SUPPLIER information (and likewise for INVOICEs in addition). Such LIST[TUPLE] are essentially as nested tables and here is how to make them.
define RECORD_CLASS LIST_DETAILED_PART
as_a_view_where(
    for_each [ INT .number, STR .name, STR(25) .color, FLT .wt, 
        LIST[ TUPLE[ INT .onbr, STR(30) .supplier, INT .qty ] ] .order_details ]
    conclude(
        there_is_a LIST_DETAILED_PART where(
            Number = .number and 
            Name = .name and 
            Color = .color and 
            Weight = .wt and 
            Order_Details = .order_details 
        )
    )
    iff ( 
        there_is_a PART where(
            Number = .number and 
            Name = .name and 
            Color = .color and Weight = .wt 
        ) and 
        .order_details = 
        [ [ .onbr, .supplier, .qty ]:
            there_is_a ORDER where(
                Part_Nbr = .number and 
                Number = .onbr and 
                Supp_Nbr = .snbr and Quantity = .qty 
            ) and 
            there_is_a SUPPLIER where(
                Number = .snbr and 
                Name = .supplier ) ]
    ))
)

And here is an example of how to query such a view (vu.vparti.3.Q):
do Write.Line( "Part\n OrdNbr Supplier Qty" );

let[ .part, .detail_list ]

ist( there_isa LIST_DETAILED_PART where( Number Is_In [ 110, 120, 133, 190 ] and Name = .part and Order_Details = .detail_list )
)

do Write.Words( .part );

let .tu Is_In .detail_list
{
  do Write.Words( " ", .tu );
}

And here is some sample output from this query:

<table>
<thead>
<tr>
<th>Part</th>
<th>OrdNbr</th>
<th>Supplier</th>
<th>Qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>button</td>
<td></td>
<td>Mountain Export</td>
<td>1961</td>
</tr>
<tr>
<td></td>
<td>53</td>
<td>Nikolaos Rentals</td>
<td>3313</td>
</tr>
<tr>
<td></td>
<td>85</td>
<td>Tutankhamen Rentals</td>
<td>2989</td>
</tr>
<tr>
<td></td>
<td>109</td>
<td>Tiryns Import</td>
<td>3857</td>
</tr>
<tr>
<td></td>
<td>309</td>
<td>Electra Receiving</td>
<td>3068</td>
</tr>
<tr>
<td>shelf</td>
<td></td>
<td>Apollo Sales</td>
<td>1792</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>Sunshine Warehouse</td>
<td>924</td>
</tr>
<tr>
<td></td>
<td>95</td>
<td>Knossos Export</td>
<td>3713</td>
</tr>
<tr>
<td>belt</td>
<td></td>
<td>Ra Import-Export</td>
<td>2940</td>
</tr>
<tr>
<td></td>
<td>166</td>
<td>Ephesus Sales</td>
<td>115</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>Leslie Receiving</td>
<td>2510</td>
</tr>
<tr>
<td></td>
<td>328</td>
<td>Baldur Distributing</td>
<td>1744</td>
</tr>
</tbody>
</table>

So, this is vertical partitioning where one of the columns is stored separately as a column of nested LIST[TUPLE] FIELD values.

### 15.3.3 An Exemplary Database View: NEW_ORDER

Here is a view table NEW_ORDER that is constructed via joins from three base tables (vu.new_order.1.Q):
define RECORD_CLASS NEW_ORDER
as_a_view_where(
   for_each [ INT .id, STR .supp, STR .part, FLT .qtyd, FLT .tot_wt ]
   conclude(
      there_isa NEW_ORDER where(
         Id = .id and
         Supplier = .supp and
         Part = .part and
         Quantity = .qtyd and
         Total_Weight = .tot_wt )
      iff
      ( there_isa ORDER where(
         Number = .id and
         Part_Nbr = .pno and Supp_Nbr = .sno and
         Quantity = .q where( .qtyd = ((FLT).q)/.denom )
      ) // Date_Recd, Date_Placed not used
      and
      there_isa SUPPLIER where(
         Number = .sno and Name = .supp )
      and
      there_isa PART where(
         Number = .pno and Name = .part and
         Weight = .wt )
      and .tot_wt = .qtyd * .wt )
   )
   using(
      all_keyed_record_existence_tests_pass
      outside [ INT .denom ]
   )
)
local: INT .denom = 1000
select Supplier, sum( Total_Weight )
from NEW_ORDER
order by Supplier
;
do Display each[ .part ] each_time(
   there_isa NEW_ORDER where(  
      Quantity > 4.32 and
      Part = .part which Matches "^[a-e]"
   ));

Note that the NEW_ORDER Quantity FIELD is computed from its base table correspondent by using
the VBL denom which the using specification imports from the outside (in the same manner as done for
macro PREDs). The Total_Weight FIELD is also a computed field. One of the purposes of the joins is
to translate from Supplier and Part numbers to English names for use in NEW ORDER.

15.3.4 How Daytona Processes A View

It can be helpful for the user to be aware (generally) of how Daytona processes the use of Cymbal views. Consider the NEW ORDER query above, now revisited:

```plaintext
do Display each[ .part ] each_time( there_isa NEW_ORDER where( Quantity > 4.32 and Part = .part which Matches "^[a-e]" ));
```

The first thing that Daytona is to rewrite the query into a logically equivalent form so that the use of the view in the query is isomorphic to the use of the view in the view definition:

```plaintext
do Display each[ .part ] each_time( there_isa NEW_ORDER where( Id = .id and Supplier = .supp and Part = .part and Quantity = .qtyd and Total_Weight = .tot_wt ) and .qtyd > 4.32 and .part Matches "^[a-e]" );
```

Note the introduction of innocuous references to FIELDs not mentioned in the query. Then, modulo the VBL renaming and type enforcement steps that Daytona does in practice, the definition of the view is substituted for the use resulting in the logically equivalent (using .denom = 1000):

```plaintext
do Display each[ .part ] each_time( there_isa ORDER where( Number = .id and Part_Nbr = .pno and Supp_Nbr = .sno and Quantity = .q where( .qtyd = ((FLT).q)/1000 )) and there_isa SUPPLIER where( Number = .sno and Name = .supplier ) and there_isa PART where( Number = .pno and Name = .part and Weight = .wt ) and .tw = .q * .wt and .qtyd > 4.32 and .part Matches "^[a-e]" );
```

So far, the processing amounts to type-sensitive macro expansion. The next phase is more interesting because it has to do with the simplification of the view. For example, Daytona observes that it is only the PART name that is being requested. Consequently, it seems useless to compute the total weight and for that matter, to look up the SUPPLIER name. And indeed it is. The policy that Daytona uses is to iteratively eliminate useless VBLs (and keyed record existence tests, if so designated, as pointless).
VBL is *useless* iff all of its occurrences are defining occurrences in the expansion of the view invocation. (Recall that if a VBL is defined in a disjunction, then it can have multiple defining occurrences.) In other words, if a VBL has no uses (i.e., is use-less) -- and note that it is use-ful to print it out, then there is no point in computing it: it is a "WORN" VBL, which is to say, write-once, read-never. Further note that if the view uses a VBL to sort a box on, then it is use-ful. So, some care may need to be exercised to avoid reflexively sorting a view box with _lexico_order instead of using the more selective as_unique_key keyword; see vu.sql.2.Q where the two sort specifications in DYNORD_P and DYNORD_P_DR result in different treatments of missing values. In this regard, know that unless as_unique_key is being used, every SET and every sorted BOX is implemented using at least one key that involves all the fields, even if only some-of-the-fields keys are specified. In this next example, literally removing useless VBLs and (pointless) keyed record existence tests in an iterative fashion results in the suggestively commented:

```sql
do Display each[ .part ] each_time{
    there_isa ORDER where( // Number = .id and
    Part_Nbr = .pno and // Supp_Nbr = .sno and
    Quantity = .q where( .qtd = ((FLT).q)/1000 ))
    // and there_isa SUPPLIER where( Number = .sno and Name = .supplier )
    and there_isa PART where(
        Number = .pno and Name = .part /* and Weight = .wt */ )
    // and .tw = .qtd * .wt
    and .qtd > 4.32
    and .part Matches "^[a-e]" );
```

Note that even though `qtd` is not printed out, it is used (in the test comparing it to 4.32) and so it must be kept.

Removing useless VBLs is not only valuable from the standpoint of efficiency, but it is also necessary to ensure that view tables have the desired semantics. Recall that Cymbal missing value semantics implies that a record will *not* be skipped over on account of a FIELD whose value is missing *but* which is *not* referenced in the query's _there isa_ for that RECORD CLASS. So, to support the same semantics for a view table, Daytona must make sure that FIELDS in the view definition that are not referenced in a _there isa_ use of the view do not somehow cause view records to be skipped over due to missing values in the underlying base data: clearly, that is accomplished by removing the associated useless VBLs and their FIELD references. Finally, the assertions created during view processing are of course subject to Daytona’s usual assertion manipulation which includes equality removal and conjunct commuting.

### 15.3.5 More On Views

For the record, here is the grammar for view definitions:
**ViewDef** ::= define RECORD_CLASS upper as_a_view_where [ViewEquivSeq ]
   I alternate_view_spec (ViewEquivSeq ) ]
   I using( 
   I outside [ VblSpecSeq ] ]
   I opcond_somewhere [ VblSpecSeq ] ]
   I as_read_only ] has_variant_fields ]
   I all_keyed_record_existence_tests_pass ]
) ] ; ]

**ViewEquivSeq** ::= ViewEquiv ] and ViewEquiv ]
 ;

**ViewEquiv** ::= for_each SomeVblSpecs conclude (Desc iff (Asn ))
 ;

**alternate_view_spec** ::= with_this_isas_using | with_from_sections_using
   | with_deletes_using | with_this_is_no_deletes_using
   | with_adds_using | with_updates_using
 ;

In addition to placing a view definition in the same file as the uses of the view, it is also possible to put it in the environment in some *.env.cy so that it can be included automatically; this is a practice that is available for all of Daytona’s non-OPCOND declarative predicates. To find the aar or *.env.cy file for a RECORD_CLASS, whether it be a realized/conventional table or a virtual table (view), just use the descriptively, if awkwardly, named aar_or_env_fl_for command. For example, aar_or_env_fl_for DETAILED_PART. Even better, to edit a view definition, just use the likes of DS Vi -env DETAILED_PART. Also, both DS Synop and DS Show can be used with views.

Furthermore, since views are based on the macro PRED concept and implementation, they can use Ground_In_Use as well. Here is an example of a view definition using Ground_In_Use to prepare a STR value for use as a HEKA(5) key when it is appropriate.
define RECORD_CLASS SMORGAS_PARTIAL_VIEW
as_a_view_where(
    for_each [ STR .str_val, UINT .uint_val ]
    conclude(
        there_is_a SMORGAS_PARTIAL_VIEW where(
            Str_Val = .str_val
            and Uint_Val = .uint_val )
        iff ( 
            there_is_a SMORGAS where(
                F_Heka_Fixed = .heka_val
                and F_Uint = .uint_val )
            and if( Ground_In_Use ) // i.e., the view vbl .str_val
                then( .heka_val = (HEKA(5)) .str_val )
            else( .str_val = (STR).heka_val )
        )))
    do Display each [.uval ] each_time( 
        there_is_a SMORGAS_PARTIAL_VIEW
        where( Str_Val = "32415" and Uint_Val = .uval ));
)

In this case, the implied subject of Ground_In_Use is .str_val which has to be found declared in the for_each LIST for the view. The point with this use of Ground_In_Use is that it enables the view SMORGAS_PARTIAL_VIEW to be used successfully both in situations when Str_Val is being generated and in other situations where the query has provided value(s) for Str_Val with the intention that they will be used as key values for indexed lookup. In the first situation, .str_val will be computed from .heka_val using one algorithm and in the second, the reverse using a different algorithm -- and all this is hidden from the user of the view.

There is an important caveat though. In the more typical box-of-key-field-values situation using a view FIELD that is defined using Ground_In_Use, the user must in effect do the transformation themselves in order for a KEY/INDEX to be used. In other words, for this SMORGAS_PARTIAL_VIEW, in order for any KEY/INDEX on Str_Val to be used, the user must explicitly put the Is_In predication before the _isa (vu.grndNuse.2.Q):

    do Display each [.str_val, .uint_val]
each_time( 
        .str_val Is_In [ "45", "55" ]
        and
        there_is_a SMORGAS_PARTIAL_VIEW where(
            Str_Val = .str_val and
            Uint_Val = .uint_val
        ));

In other words, while the use of the convenient phrasing:

    Str_Val = .str_val which Is_In [ "45", "55" ]

will be accepted and processed into the correct answers, it cannot ever result in the box-of-key-field-values transformation being employed to use the assumed KEY/INDEX on F_Heka_Fixed. The technique for having
that occur is illustrated by the Display query immediately above.

However, there is some good news here in that Is_In_Range will work on a view FIELD defined by using Ground_In_Use. This is good news because some Is_In queries can be expressed alternatively and perhaps even more efficiently by using Is_In_Range. Recall the Ground_In_Use assertion used above in defining SMORGAS_PARTIAL_VIEW:

\[
\begin{align*}
\text{if} ( \text{Ground_In_Use} ) \\
\text{then} ( \text{.heka_val} = (\text{HEKA}(5)) \cdot \text{str_val} ) \\
\text{else} ( \text{.str_val} = (\text{STR}).\text{heka_val} )
\end{align*}
\]

The following query will then work:

\[
\text{do Display each } [\cdot \text{str_val}, \cdot \text{uint_val}] \\
\text{each-time (}
\quad \text{there is a SMORGAS_PARTIAL_VIEW where (}
\quad \quad \text{Str_Val} = \cdot \text{str_val which Is_In_Range [ "35" -> "459" ] and}
\quad \quad \text{Uint_Val} = \cdot \text{uint_val}
\quad )
\);
\]

Is_In_Range of course works with a variety of types like DATE_CLOCK and IP6. Anyway, there is a requirement for this use of Is_In_Range to work. Recall that this Ground_In_Use is defining a view VBL in terms of a FIELD VBL and vice-versa. The requirement is that the then and else phrases that define one VBL in terms of the other must do so in such a way that each VBL is a non-decreasing function of the other. This is typically the case.

15.3.6 Applications Of Views: Data And Query Abstraction

In this setting, an abstraction is defined as the result of removing distracting detail from a situation so as to reveal its simple and essential nature. Views are table abstractions that enable DBAs to hide the possibly changing implementation of a table from the query writers -- as evidenced by the vertical partitioning example above. Indeed, views insulate the end-user from all manner of (changing) lower-level detail including changes in the names of the fields, changes in their types, changes in how they are stored and computed, etc.

Furthermore, views enable users to provide a simpler query interface by hiding complicated query logic -- as evidenced by the NEW_ORDER query above.

15.3.7 Applications Of Views: SQL And Cymbal Parallelization

Views provide a simple way to enable Daytona’s SQL to make use of Cymbal’s parallelization capabilities. In this way, by using predefined views written possibly by someone else, SQL users can obtain the benefits of Cymbal parallelization without writing any Cymbal, i.e., by writing 100% SQL queries. Of course, these kinds of views can also simplify Cymbal queries in the same way. This is accomplished by writing view definitions that parallelize the access to individual tables. Here’s an example (vu.sql.3.Q):
define RECORD_CLASS PARTED_P as_a_view_where(
    for_each [ INT .nbr, STR .color, FLT .wt ]
    conclude (
        there_is_a PARTED_P where(
            Number = .nbr and
            Color = .color and
            Weight = .wt )
        iff(
            [.nbr, .wt, .color ] Is_The_Next_Where(
                .sect Is_In [ 1 -> 10 ]
            and parallelizing(
                there_is_a PARTED from_section [ .sect, 10 ] where(
                    Number = .nbr and
                    Color = .color and
                    Weight = .wt
                )
            )) in_lexico_order parallel_for .max_clone_nbr
        )
    )
) using( outside[ INT .max_clone_nbr ] );

select Number, Color
from PARTED_P
where Number < 110
parallel for 3;

select PARTED_P.Color, avg("ORDER".Quantity)
from PARTED_P, "ORDER"
where "ORDER".Part_Nbr = PARTED_P.Number
group by PARTED_P.Color
order by PARTED_P.Color
parallel for 3;

This view definition illustrates the general strategy, which is to define the view table to be a parallelized box, or more accurately, to say that a view record exists if and only if the TUPLE of corresponding field values is a member of a parallelized box. In general, of course, the assertion defining this parallelized box can be anything -- e.g., it doesn’t have to confine itself to working with just one base table (and could actually include SQL itself!). In this case though, the view table is defined to be the box formed by parallelizing a sequential access to the horizontally partitioned table PARTED.

As illustrated by the queries, PARTED_P is used just like any other table in SQL queries. However, in order to be able to specify the parallelization factor in an SQL query by using the non-standard parallel for keyword phrase, the user must do two things. First, of course, the query must use the parallel for SQL phrase, which is a Daytona customization for SQL selects. Second, the view definition must use the VBL max_clone_nbr by that exact name to specify the parallelization factor via Cymbal’s parallel_for syntax, including using the outside keyword to import it into the
view, which makes sense of course because the view definition is not providing the value for the parallelization factor, it has to come from outside. (Views use the outside keyword just like macro definitions do; outside VBLs may be either locals, exports, imports or declarative.)

In a 100% SQL query file, if the first requirement is met but there is no parallel for argument, then Tracy will complain about not being able to determine a non-OBJ type for max_clone_nbr or max_clone_nbr_sCPn for some integer n:

error: have not been able to determine a non-OBJ Datatype for variable `max_clone_nbr'

This is just a special case of the general error condition, which is that any VBL import of a parallelization factor into a view definition must be backed up by an external definition someplace else in the query. So, for example, that same error will occur when such a view definition is used in Cymbal and no value has been set for max_clone_nbr.

Note that if the SQL use of the view is embedded in Cymbal, then instead of being a constant INT, the argument to parallel for can be a dereferenced VBL (like .x) defined in the surrounding Cymbal (but not a general term). Furthermore, such an explicit SQL parallel for specification overrides any definition of max_clone_nbr outside of the SQL (sub)query. Here is an example:

```
set [.para_deg] = read( from _cmd_line_ bia[ 3 ] );

select Number, Color
from PARTED_P
where Number < 110
parallel for .para_deg;
```

However, suppose one wants to stick to the SQL standard by avoiding the use of parallel for. Then, the hard-wired specification of a constant parallelization factor can be done in the (perforce Cymbal) definition of the view so as to keep strictly to the SQL standard by omitting any parallel for in the SQL select. Alternatively, when writing standard SQL embedded in Cymbal, the view definition can import the parallelization factor whose value would then be defined in the Cymbal preceding the SQL that uses the view: once again, no parallel for is being used in the SQL here either but rather the Cymbal parallel_for is being used in the view definition.

There is another consideration which is that the view definition can only be included in the file that has the SQL query if Daytona does not believe it is necessary to read in the rcd for the parallelized table during SQL parsing (by the Squirrel program). Such a need can be obviated by the simple expedient of qualifying all column names in the query with their corresponding table names (as in <table>.<column>), noting that that is merely sufficient, not necessary, meaning that there are exceptions that will work as well. In fact, one exception happens when the FROM clause has exactly one table (i.e., the view table). For emphasis though, note that using select * does not pass muster here. Violating this requirement will lead to an error message like:

```
error: failed to find &name symbol table information on RECORD_CLASS 'SUPPLIER_V';
could it be that this is a view RECORD_CLASS whose definition \n is in the same Cymbal file as the query?
```
However, for users who wish to parallelize SQL in this way and not use any Cymbal at all in their query files or more generally, for those who wish to use the same view definition for many queries without repeating it in each query file, this consideration is happily irrelevant! The solution instead is to put the view definition in one of the *.env.cy files, as can be done in general for any declarative PRED definition.

Note that Cymbal can be used just as well as SQL to query these parallelized view tables:

```cymbal
do Display each[ .nbr, .color, .wt ]
  each_time( there_isa PARTED_P where( 
    Number = .nbr where( .nbr < 110 ) and 
    Color = .color where( .color Matches "a" ) and 
    Weight = .wt )
  );
```

Here is an example showing how to parallelize SQL and/or Cymbal random access to dynamically horizontally partitioned RECORD_CLASS (vu.sql.4.Q):
define RECORD_CLASS ORDERI_P
as_a_view_where(
  for_each [ INT .bin_nbr, INT .number, INT(_short_) .sno, INT .pno, INT .qty
    ] conclude(
      there_is_a ORDERI_P where(
        Bin_Nbr = .bin_nbr and
        Number = .number and
        Supp_Nbr = .sno and Part_Nbr = .pno and Quantity = .qty )
      iff(.
        .tot_sects = 10
      and .hparti_box = { [ .bin_nbr ] :
        there_is_a_bin_for ORDERI where( Twenty_Cohort = .bin_nbr )
      : with_random_indices_stored } and
      [ .bin_nbr, .number, .sno, .pno, .qty ] Is_The_Next_Where(
        .sect_nbr Is_In [ 1 -> .tot_sects ]
      and
      parallelizing( [ .bin_nbr ] Is_In .hparti_box in_random_order
      from_section[ .sect_nbr, .tot_sects ]
      and
      there_is_a ORDERI where(
        Twenty_Cohort = .bin_nbr and
        Number = .number and
        Supp_Nbr = .sno and Part_Nbr = .pno and Quantity = .qty
      )
    ) in_lexico_order parallel_for .max_clone_nbr
  )
)) using( outside[ INT .max_clone_nbr ] );

select * from ORDERI_P
where Supp_Nbr in [ 410 -> 415 ]
parallel for 4;

The view definition computes and stores all of the bin_nbrs into .hparti_box and then offers 10 random sections of those bin_nbrs as jobs. In the query, the INTERVAL on Supp_Nbr causes each BIN accessed by a clone to be accessed using the Supp_Nbr KEY and INDEX.

In keeping with Daytona’s use of the SPMD (single-program-multiple-data) paradigm, parallelization is typically done by parallelizing the access to the first table in a join situation. In SQL, this means putting the parallelized table first in the FROM clause. This is very important: the system will accept a parallelized table reference later in the FROM clause but the resulting execution is likely to be very slow.
15.3.7.1 Using from_section To Parallelize A View

The from_section concept underlies and supports many parallelization strategies. Since from_section can be used to refer to portions of both BOXes and physical RECORD_CLASSES, why not likewise for virtual RECORD_CLASSES, i.e., views? This is a good idea and it is accomplished by defining a view with a with_from_sections_using for_each assertion argument of the same form as an argument for as_a_view_where. Here is an augmented definition of VPARTV from before that supports using a view from_section to divide up the implementation into pieces based on sequential access (vu.vparti4.Q).

```
define RECORD_CLASS VPARTV as_a_view_where( ... )
with_from_sections_using(
  for_each [ INT .nbr, STR .name, STR .color, FLT .weight ]
  conclude(
    there_isa VPARTV from_section[ .sect, .tot_sects ] where(
      Number = .nbr and
      Name = .name and
      Color = .color and
      Weight = .weight )
    iff(
      there_isa PART_1 from_section[ .sect, .tot_sects ]
      where(
        Number = .nbr and
        Name = .name )
      and
      there_isa PART_2 where(
        Number = .nbr and
        Color = .color and
        Weight = .weight ) )))
  using( all_keyed_record_existence_tests_pass )
```

Note that sect and tot_sects are local VBLs and therefore, it would be wrong to import them in the using clause. Note also that from_section is only used once in the RHS of the iff. Here is this functionality in use in such a way that the implementation parallelizes sequential access to the base table PART_1:
in_lexico_order
  parallel_for 2
do Display each [ .name ]
each_time(
    .sect Is_In [ 1 -> 4 ] and
    parallelizing ( 
      there_isa VPARTV from_section[ .sect, 4 ] where( 
        Name = .name which Matches "^[a-c]$"
      ));

Interestingly enough, the from_section argument can be applied to a box in the RHS of the iff so as to achieve parallelization of random access.

define RECORD_CLASS VPARTV_2 as_a_view_where( ... )
with_from_sections_using(
  for_each [ INT .nbr, STR .name, STR .color, FLT .weight ]
  conclude( 
    there_isa VPARTV_2 from_section[ .sect, .tot_sects ] where( 
      Number = .nbr and
      Name = .name and
      Color = .color and
      Weight = .weight )
    iff( 
      .nbr Is_In .box_o_keys from_section[ .sect, .tot_sects ]
      and
      there_isa PART_1
      where( 
        Number = .nbr and
        Name = .name )
      and
      there_isa PART_2 where( 
        Number = .nbr and
        Color = .color and
        Weight = .weight ) )))
using( all_keyed_record_existence_tests_pass
  outside[ LIST[ INT ] .box_o_keys ]
)
local: LIST[ INT ] .box_o_keys
set .box_o_keys = [ 100 -> 200 by 5 ];
in_lexico_order
parallel_for 2
do Display each [ .nbr, .name ]
each_time( .sect Is_In [ 1 -> 4 ] and
parallelizing( there_is_a VPARTV_2 from_section[ .sect, 4 ] where( Number = .nbr and
Name = .name which Matches "^[a-c]" )
));

Note that the form of this query is very much like its predecessor but its principal mode of access is random, not sequential.

15.3.8 Applications Of Views: Chain Variants

A chain variant record situation occurs when there is a totally ordered sequence of RECORD_CLASSES for which any given successor pair in the sequence is such that the FIELDS of one comprise an initial sequence of FIELDS for the other, implying that the second has more FIELDS and those appear at the end of the FIELD list. This is precisely what happens as schemas evolve by adding more and more FIELDS to a table: the sequence of FIELDS is strictly monotone increasing from one evolution to the next. The ideal outcome in this situation is when, in order to do schema evolution, all that is necessary is to do a little "logical"/metadata work (as in defining views) without doing any "physical" adjustments at all, i.e., without doing any reformatting of the data, rebuilding of indices, or rewriting of queries. After the necessary metadata changes, one would just add the new longer records to what is already there and that would be the end of it.

This ideal is achievable by using Daytona’s chain variant views. What’s more, records of differing numbers of FIELDS can be co-located in the same data files. Users just query using Cymbal or SQL that makes reference to whatever FIELDS they are interested in, regardless of whether all records have those FIELDS or not. Indices can be built on any of the FIELDS for whatever vintage/variant.

To realize this happy situation, the DBA takes advantage of the fact that it is possible to have more than one ViewEquiv specified in view definitions. This is illustrated by the following simplified example (vu.chain.2.Q) where PART_CV_1 and PART_CV_2 form the chain of variant RECORD_CLASSES serving to define the PART_CV view:
define RECORD_CLASS PART_CV as a view where(
    for_each [ INT .nbr, STR .name // , STR .color ]
    conclude (
        there_isa PART_CV where(
            Number = .nbr
            and Name = .name
            // and Color = .color
        )
        iff(
            there_isa PART_CV_1 where(
                Number = .nbr which_is >= 100
                and Name = .name
            )
        )
    )
    or
    for_each [ INT .nbr, STR .name, STR .color ]
    conclude (
        there_isa PART_CV where(
            Number = .nbr
            and Name = .name
            and Color = .color
        )
        iff(
            there_isa PART_CV_2 where(
                Number = .nbr which_is >= 177
                and Name = .name
                and Color = .color
            )
        )
    )
);
    select Name from PART_CV where Name Matches "n" ;
    select Name from PART_CV where Color Matches "a" ;

Note the disjunction of two for_each assertions. When processing a PART_CV there_isa in a query, Daytona looks at all the FIELDSs that are mentioned in the there_isa and uses as the definition of the view there_isa, the first for_each disjunct that matches the there_isa, i.e., the first one that has all of the FIELDSs used in the there_isa. For this all to work, the base RECORD_CLASSes (like PART_CV_2) that are the potential matches must satisfy these conditions:

- Their sequences of FIELDSs must form a chain as previously defined!
- The data files of any member in the chain must be data files in any successor.
- The INDEXes of any member in the chain must be INDEXes in any successor.
- In object-oriented terms, the chain of (base) RECORD_CLASSes in this situation is associated with a corresponding chain of superclass relationships for the associated CLASSes of OBJECTs being described by the RECORD_CLASSes. For example, the first CLASS of OBJECTs is a superclass of all of them and each predecessor OBJECT CLASS is a superclass of each successor CLASS. Note that each successor CLASS has more attributes describing its members.
At any rate, each (base) RECORD_CLASS in the chain must be definable via its own Cymbal condition that is a function of one or more of those fields that are in common with all members of the chain. If a record satisfies such a condition, then the object it is describing is necessarily a member of the associated OBJECT CLASS and each of its successors if any. Therefore, the membership criterion for any RECORD_CLASS in the chain is that its Cymbal condition must be true and that for its successor RECORD_CLASS (if any) must be false. For example, above, a record is in PART_CV_1 if and only if its Number >= 100 and its Number < 177; a record is in PART_CV_2 if its Number >= 177.

Consequently, the condition for a predecessor RECORD_CLASS in the chain must subsume that of any successor: this means that whenever a successor’s condition is true, then so must be any predecessor’s condition. In the example, note that whenever Number >= 177, Number >= 100.

- The order that the ViewEquiv appear in the view definition must be the same order that the corresponding base RECORD_CLASSes appear in in the chain, i.e., from smallest number of FIELDs to the largest.

- The FIELDs section of the base rcd for each member of the chain has a Last For Recls note for each feasible member of the chain including itself. For example, the FIELDs for PART_CV_2 are described by:

```yaml
#{ FIELDS
    #( FIELD ( Number )
        <Type, INT(_long_)>  
        <__Position, 1> )#
    #( FIELD ( Name )
        <Type, STR(*)>   
        <Last_For_Recls, PART_CV_1>
        <__Position, 2> )#
    #( FIELD ( Color )
        <Type, STR(25)>
        <Last_For_Recls, PART_CV_2>
        <__Position, 3> )#
}
```

The assumption is that only the biggest/last member of the chain will be used by Sizup to build the indices.

### 15.3.8.1 Using Chain Variants With Dynamic Horizontal Partitioning

The simple example above does not utilize the widely used dynamic horizontal partitioning feature. When using dynamic hparti, all of the rules above continue to be in effect except that a simplification is permitted. As shown in the test suite by the definition for the view NETFLOW_CV in
daytona.env.cy and its base rcd.NETFLOWED3 in aar.misc, it is permitted for the same RECORD_CLASS (NETFLOWED3 here) to be used as the base RECORD_CLASS for all of the for each alternatives in the view definition (vu.chain.hparti.1.Q). Consequently, the same FILE_INFO_FILE(s) are used by all the variants.

As before, in general, it doesn’t have to matter which kind of data records are in which data files as long as the chain conditions are consistent with the Last For Recls notes in accurately characterizing the alternatives. However, in this example, note that the chain conditions happen to be expressed exclusively in terms of the hparti partitioning attribute Date_Collected as illustrated by:

\[
\text{Date}_\text{Collected} = .\text{randt which is } >= '2008-03-12'.
\]

Consequently, between consecutive (increasing) specified Date_Collecteds, all the records in the associated data files have to have exactly the same FIELDs and hence the same number of FIELDs. This just reflects the common situation where the later data files in terms of Date_Collected are the ones that have the additional FIELDs. The point is that this particular reliance on the hparti attribute Date_Collected effectively forbids having shorter records in files associated with later specified Date_Collecteds but such a stricture is not necessary in general.

The fact that the value for the attribute Last_For_Recls is the same (i.e., NETFLOWED3) just reflects that the same base RECORD_CLASS is being used in all cases. The advantage of using this single base RECORD_CLASS is of course simplicity.

15.3.9 Applications Of Views: Generalized Horizontal Partitioning

Daytona’s built-in horizontal partitioning is specified by identifying each BIN by a unique set of values for its horizontal partitioning FIELD(s). Sometimes though it is more advantageous to do the partitioning on the basis of essentially arbitrary functions of one or more of the true physical FIELDs of the record that map values of those FIELDs to designated BIN identifiers and thus map the associated record to the corresponding BIN. Or to put it simply, generalized horizontal partitioning occurs when the system automatically assigns records to partitions (BINs) on the basis of those records satisfying arbitrary (but fixed) constraints on their FIELDs. For example, consider the ORDER table with its Number FIELD and suppose there are 20 BINs numbered 1 to 20. One might want to put all the ORDERs whose Numbers are in a given LIST in BIN 6 and likewise for other LISTs and BINs. Or one might want to hash on the ORDER Number to determine the BIN (or in this case, do something as simple as \( .\text{nbr} \% 20 + 1 \) to compute the BIN number from the ORDER Number) or perhaps map ORDERs to BINs based on specified INTERVALs containing the ORDER Number. In the general case, all one needs is a function that maps TUPLEs of values for true physical (i.e., non-hparti) FIELDs to underlying built-in hparti FIELD values. In other words, assuming the use of the built-in hparti mechanism underneath, one would like to use a view to put a more convenient, flexible, and general veneer on top of that. Here is an example of how that can be done (vu.hparti.1.Q):
define RECORD_CLASS ORDER_H1
as_a_view_where(
    for_each [ INT .ono, STR .supp, STR .part, DATE .date_rcv, INT .qty ]
    conclude(
        there_is_a ORDER_H1 where(
            Number = .ono and
            Supplier = .supp and
            Part = .part and
            Date_Recd = .date_rcv and
            Quantity = .qty
        )
    iff (  
        there_is_a ORDERI where(
            Number = .ono and
            Twenty_Cohort = (.ono -1)/20 +1 and
            Supp_Nbr = .sno and
            Part_Nbr = .pno and
            Date_Recd = .date_rcv and
            Quantity = .qty
        )
        and
        // OK to use other tables if desired or even computed fields!
        there_isa SUPPLIER where(
            Name = .supp1 where(.supp = (STR).supp1 )
            and Number = .sno
        )
        and
        there_isa PART where(
            Name = .part1 where(.part = (STR).part1 )
            and Number = .pno
        )
    )))

In this view definition, there is exactly one simple line which specifies the mapping of the underlying base table ORDERI Number to its hparti FIELD Twenty_Cohort, where Twenty_Cohort is a number from 1 to 20 indicating a particular underlying BIN. Notice that the view ORDER_H1 doesn't even have a horizontal partitioning FIELD -- all of its FIELDS have direct physical values in records. And yet ORDER_H1 is effectively horizontally partitioned after all by the nature of its implementation via the view. Also, note that the one-line definition of the derived FIELD Twenty_Cohort must follow the (simple) one-line definition of the base FIELD Number in the view definition, so that, once again, concepts are defined before use.

A happy consequence of this is that the query writer using the view does not have to come up with expressions that identify which BIN has the desired data since Daytona view processing will take care of that detail. Specifically, when updating ORDER_H1 or when searching for a particular ORDER_H1 by Number, Daytona will automatically select the correct BIN (and only the correct BIN) by inferring the appropriate equality on Twenty_Cohort needed to identify that BIN. Here are
examples of queries where the BIN selection is automatic:

```sql
select * from ORDER_H1 where Number = 404;
selct * from ORDER_H1 where Number Is_In [ 303 -> 404 ];
update ORDER_H1 set Date_Recd = '2011-11-11' where Number = 303 ;
```

The middle example uses the box-of-key-field-values optimization which implies looking up ORDER_H1 records on the basis of equalities and that implies automatic BIN selection.

Furthermore, to facilitate queries choosing only the appropriate BINS to visit by means of inequalities, it is possible to include the hparti attribute in the view table and use that for this purpose. For an example, consider NETFLOWED4_VU as used in netflo4.1.Q where Starting_DC is the horizontal partitioning FIELD derived from the Date_Hour physical FIELD so that a Starting_DC is the start point of an interval of Date_Hour's. Then the inequalities on Starting_DC in the following query make sure that only the smallest number of BINs are opened consistent with satisfying the inequalities on Date_Hour:

```sql
select count(*) from NETFLOWED4_VU
where Starting_DC between '2006-06-06@8:05am'DC and '2006-06-06@3:00pm'DC
and Date_Hour between '2006-06-06@8:05:30am'DC and '2006-06-06@3:30pm'DC
```

However, even better than that is the ability to instruct Daytona to automatically generate appropriate inequalities for derived FIELD values based on inequalities in the query constraining values of that (presumed only) base FIELD that is used to define the derived FIELD. For example, the values of the derived FIELD Twenty_Cohort above are computed from values of the base FIELD Number by using the function defined by \((.ono -1)/20 +1\). Suppose a query constrained Number to be between 15 and 55. Then the user both needs and wants Daytona to infer that Twenty_Cohort needs be constrained between 1 and 3. In other words, if \(f\) is the function that maps Number to Twenty_Cohort, then when Number Is_In [ .a -> .b ], then it would be good if Daytona would infer that Twenty_Cohort Is_In [ f(.a) -> f(.b) ]. This is only true, of course, if \(f\) is non-decreasing.

Fortunately, Daytona can be so instructed by means of using a keyword+argument for the keyword `infer_bounds_for` in the `using` argument to the view definition, as in:

```sql
using( infer_bounds_for ^Twenty_Cohort^ )
```

The argument to `infer_bounds_for` must be a THING that is either the name of a derived FIELD or else a table-qualified name such as ORDERI.Twenty_Cohort . The latter possibility is of value in disambiguating FIELD names in view definitions involving multiple tables. Any table-qualified name must refer to a base table of the view, not the view table. The derived and base FIELDs and their names must be for base tables, although it is also OK for them to be used in the view table. Obviously, the respective datatypes must be scalars, not COMPOSITEs like LISTS. And very important: the mapping function defining values of the derived FIELD in terms of the base FIELD must be explicitly given (in Cymbal) in the view definition. For an example, see the `which_is =` portion of the definition of Starting_DC two paragraphs down. Also, keep in mind that just because a derived FIELD is computed (by a function on a base FIELD) does not mean that it has to be a "computed FIELD" in the sense of a virtual view FIELD that has no actual values in base tables; on the contrary, the typical way `infer_bounds_for` is used is for non-hparti base table FIELDs to constrain derived hparti base table FIELDs so that only the
minimal number of BINs get opened by a query.

This feature will only work when the function defining the derived FIELD is non-decreasing, but that is typically the case. (But it is certainly not true for a hash function!) Furthermore, the derived FIELD can only be a function of just one other FIELD, although that other FIELD could be defined in the view as a suitable function of other FIELDs. Then, however a bound on the base FIELD is expressed in the query, a corresponding bound on the derived FIELD will be generated as illustrated by Field_3 < .y causing Daytona to generate Hparti_1 <= f(.y). Note that, in recognition of the non-decreasing nature of the mapping function, the only inequalities used for the derived FIELD are >= and <=.

The definition of Starting_DC in NETFLOWED4_VU is of interest because it defines BINs that are `1m`TIME duration first in the sequence and then finish with a sequence of hourly BINs:

\[
\text{Starting\_DC} = .\text{start}\_dc \\
\text{which}\_is = \text{if\_else}( \text{.dthr} <= \text{"2006-06-06@8:10am"}\text{DC} ), \\
\text{round\_down\_to}( \text{.dthr}, \text{`1m`TIME} ), \\
\text{round\_down\_to}( \text{.dthr}, \text{`1h`TIME} ) 
\]

Here are examples of SQL queries where Daytona will automatically infer bounds for Starting_DC:

```
select Seq_Nbr, Src_Addr, Starting_DC, Date_Hour
from NETFLOWED4_VU
where Date_Hour >= .dcl1 and Date_Hour <= .dcl2 ;
select Seq_Nbr, Src_Addr, Starting_DC, Date_Hour
from NETFLOWED4_VU
where Date_Hour between .dcl1 and .dcl2 ;
select Seq_Nbr, Src_Addr, Starting_DC, Date_Hour
from NETFLOWED4_VU
where Date_Hour Is_In_Range [ .dcl1 -> .dcl2 ] ;
select Seq_Nbr, Src_Addr, Starting_DC, Date_Hour
from NETFLOWED4_VU
where Date_Hour in [ .dcl1 -> .dcl2 by `1s`TIME ] // box-of-key-field-values
```

Of course, it does this for Cymbal too! (All Daytona SQL is translated into Cymbal.)

Informative examples are contained in inferbounds.?Q, netflo4.1.Q, vu.hparti.netflo4.IQU .

As mentioned elsewhere in this chapter, to support doing SQL deletes and other operations using this_is, it is important that with_this_is_no_deletes_using and with_this_isas_using both be specified in the view definition and that necessarily they do not make (pointless) references to horizontal partitioning attributes in the RHS of the iff.

At any rate, the general principle behind generalized horizontal partitioning is illustrated by:
define RECORD_CLASS WIDGET_GHP
as_a_view_where(
    for_each [ SomeVblSpecs ]
    conclude(
        there_is_a WIDGET_GHP where(
            Field_1 = .f_1 and · · ·
            Field_n = .f_n and · · ·
            other Field equalities
        )
    )
    iff (
        there_is_a WIDGET where(
            Field_1 = .f_1 and · · ·
            Field_n = .f_n and · · ·
            Hparti_Field_1 = g_1(.f_1, . . . , .f_n) and · · ·
            Hparti_Field_k = g_k(.f_1, . . . , .f_n) and · · ·
        )
    )))

Here the base hparti fields are the \textit{Hparti\_Field\_i} and the \textit{g_i} are the functions that map the values of physical \textit{FIELDs} to the \textit{Hparti\_Field\_i} values. The \textit{Hparti\_Field\_i} equalities must appear lexically after the ones that define the \textit{f_i}.

15.3.10 Applications Of Views: IP2/IP6 Coexistence And this\_isa

The introduction of IPv6 to networking has given rise to the problem of how to evolve existing table schemas so that they handle both IPv4 and IPv6 data at the same time using the same \textit{FIELD} in the data records for both old and new data.

This problem can be solved by using views. In so doing, fortunately, one can leverage the fact that there are two standard ways for an IPv4 address to be embedded in a corresponding IPv6 address, implying that those pseudo IPv4 addresses can coexist in this form with other fully general IPv6 addresses. In practice, typically, there is a known date when the application begins to store IP6 data. Obviously, prior to that date, the data stored in the specified field has the IP2 datatype. After that date though, the data is going to be stored in the IP6 datatype although if it arrives for ingest as IPv4, then it will be rewritten as an IPv6 address using the chosen embedded IPv4 representation and so stored in Daytona using the IP6 datatype in any case. The application of course can test any IP6 object to determine if it is one or the other of the embedded IPv4 forms.

The following definition for the view table NETFLOW\_ANY is based on the old "physical" NETFLOW table containing IP2 data as stored prior to "2006-06-06@08:02" and the new "physical" NETFLOWED table which stores IP addresses as IP6 for data loaded after "2006-06-06@08:02". The rcds for NETFLOW and NETFLOWED are contained in aar.misc in the test suite. Of course, any IP2 data stored after "2006-06-06@08:02" must be converted (not by this view) to the appropriate embedded-IPv4 IP6 form. Indeed, Sizup and any update queries for data must be run against the appropriate base table, not the view table. NETFLOW\_ANY’s utility is for querying.
From a theoretical standpoint, this view is of interest because it illustrates the simultaneous use of a disjunctive choice of base tables based on a categorization criterion, as well as Ground_In_Use and a special idiom supporting the use of this _isa for a disjunctively defined view. Consider then the definition of NETFLOW_ANY (vu.grndNuse.5.Q).

```
define RECORD_CLASS NETFLOW_ANY
as_a_view_where(
  for_each[
    INT .seq_nbr,
    HEKDATE_CLOCK(_yyyymmdd_24_hhmssf_) .date_hour,
    IP6 .src_addr,
    HEKINT .src_port,
    IP6 .dest_addr,
    HEKINT .dest_port,
    HEKTIME(_dhmsf_) .duration,
    HEKINT .pkts_sent,
    HEKINT .octets_sent,
    INT .src_mask,
    INT .dest_mask
  ]
) conclude(
  there_is_a NETFLOW_ANY where (  
    Seq_Nbr = .seq_nbr and
    Date_Hour = .date_hour and
    Src_Addr = .src_addr and
    Src_Port = .src_port and
    Dest_Addr = .dest_addr and
    Dest_Port = .dest_port and
    Duration = .duration and
    Pkts_Sent = .pkts_sent and
    Octets_Sent = .octets_sent and
    Src_Mask = .src_mask and
    Dest_Mask = .dest_mask
  )
) iff(
  .found_netflowed = _true_ and
  there_is_a NETFLOWED where (  
    Seq_Nbr = .seq_nbr and
    Date_Hour = .date_hour which_is <= ^2006-06-06@08:02^ and
    Src_Addr = .src_addr and
    Src_Port = .src_port and
    Dest_Addr = .dest_addr and
    Dest_Port = .dest_port and
    Duration = .duration and
  )
)
```
Syntactically, the only thing new here is the `opcond_somethow` keyword whose argument is a TUPLE of `VblSpecs` just like that used by `outside`. The purpose of `opcond_somethow` is to cause the associated `VblSpecs` to become existentially quantified VBLs just under the smallest OPCOND including the given use of the view. In other words, they become OPCOND_SOMEHOW VBLs. This seemingly arcane concept is exactly what is needed to enable Daytona to process `this_isa`'s for disjunctively defined views as illustrated by the following query (vu.grndNuse.5.Q):
local: IP6 .in_src_addr, .in_dest_addr, .in_addr

fet .choice Is_In [ 1, 2 ]
do {
  when (.choice = 1) set .in_src_addr = "::192.168.10.10"IP6;
  else set .in_dest_addr = "::0A0A:0A02"IP6;
_Show_Exp_To(.choice)

fet [ .seq_nbr, .date_hour, .src_addr, .src_port, .dest_addr, .dest_port,
  .duration, .pkts_sent, .octets_sent, .src_mask, .dest_mask ] ist {
  if (.choice = 1) then {
    there_isa NETFLOW_ANY where(
      Src_Addr = .in_src_addr )
  } else ( /* .choice = 2 */
    there_isa NETFLOW_ANY where(
      Dest_Addr = .in_dest_addr
      and Src_Port = 3000
      and Dest_Port >= 20 & <= 21 )
  )
  and this_isa NETFLOW_ANY where(
    Seq_Nbr = .seq_nbr and
    Date_Hour = .date_hour and
    Src_Addr = .src_addr and
    Src_Port = .src_port and
    Dest_Addr = .dest_addr and
    Dest_Port = .dest_port and
    Duration = .duration and
    Pkts_Sent = .pkts_sent and
    Octets_Sent = .octets_sent and
    Src_Mask = .src_mask and
    Dest_Mask = .dest_mask )
} { with_sep "|" do Write_Line( .seq_nbr, .date_hour, .src_addr,
  .src_port, .dest_addr, .dest_port, .duration, .pkts_sent,
  .octets_sent, .src_mask, .dest_mask );
}
}

This query illustrates the many-there_isa-per-this_isa paradigm. This paradigm is used to create executables whose runtime arguments amount to specifying which index to use in looking up records, which in the query above is expressed by the if-then-else containing there_isa’s. Then since regardless of the index chosen, it is desired to print out values for the same FIELDs, the single this_isa present accomplishes that goal since it is in effect matched to whichever there_isa is being used at runtime. Were it not for this paradigm, one would have to use two there_isas which would have quite a bit of
Cymbal in common. The benefits of using this paradigm are a reduction in the amount of Cymbal that needs to be written and maintained as well as the amount of C code that will be generated and compiled. However, note that actually due to the disjunctive nature of the view, once expanded, the query becomes many-

Recalling that the view mechanism supports using this isa for views, one might think that the above query should run fine without the use of the opcond_somehow VBL. To the contrary, its disjunctive nature presents a special case that requires special handling. Recalling that each use of there isa NETFLOW ANY expands into a disjunction of there isa’s for different record classes, then the problem that arises is that the flow of execution can go through a there isa for NETFLOWED and a this isa for NETFLOW. This will not work because these two record classes are different: one cannot process successfully a this isa NETFLOW for an associated there isa NETFLOWED.

Fortunately, the problem is solved due to the declarative nature of the opcond_somehow VBL found_netflowed and its use in identifying which disjunctive branches are active during the satisfaction process (i.e., during runtime). First, remember that by default, in order to handle this isa NETFLOW ANY, the view definition for there isa NETFLOW ANY is modified simply by replacing any there isa with this isa. Then since any this isa is satisfied after some there isa in the query, equalities like .found_netflowed = _true_ are generators when associated with a there isa and tests when associated with a this isa. So if a disjunct branch for there_is_a NETFLOW has been satisfied, then .found_netflowed will be defined to be _false_ and so the only this isa disjunct branch that can be satisfied is one for this_is_a NETFLOW; the others will be ignored because their truth value is false. Note that the outside keyword is useless for this endeavour because its VBLs are procedural, not declarative, implying that their values are constant throughout the satisfaction processing of the assertion: hence the introduction and utility of opcond_somehow.

The use of this paradigm for handling this isa’s requires special care on the part of the user and so it comes with warnings like this:

```
warning: there are one or more best NETFLOWED there_isa descriptions
that correspond to a given this_isa from a view and
they do not properly exhaust all the cases needed to determine for sure
at compile-time which one corresponds to the this_isa at runtime;
consequently, your use of the ‘opcond_somehow’ VBLs argument to the view
definition had better be correct because Dayton cannot prove it to be so.
```

The price for the user misusing this paradigm is inscrutable runtime failures.

### 15.3.11 View Updates

View updates are generally recognized as being problematical. Daytona handles them by offering the user the ability to customize any and all of them. To see how easy it is to find a case needing customization, take the case of doing deletes for the first and simplest vertical partitioning example where:
define RECORD_CLASS VPARTV as_a_view_where(
  for_each [ INT .nbr, STR .name, STR .color, FLT .weight ]
  conclude(
    there_isa VPARTV where(
      Number = .nbr and
      Name = .name and
      Color = .color and
      Weight = .weight )
    iff(
      there_isa PART_1 where(
        Number = .nbr and
        Name = .name )
      and
      there_isa PART_2 where(
        Number = .nbr and
        Color = .color and
        Weight = .weight )
    )))
  using( all_keyed_record_existence_tests_pass )

Here is an associated delete with subsequent logically equivalent statements:
do Change so_that( there_is_no VPARTV where( Number = 103 ));

do Change so_that( not( there_is_a VPARTV where( Number = 103 ) ));

do Change so_that( not(  
    .nbr = 103 and  
    there_is_a VPARTV where(  
      Number = .nbr and  
      Name = .name and  
      Color = .color and  
      Weight = .weight  
    )  
  ));

do Change so_that( not(  
    .nbr = 103 and // such equalities are moved to the front  
    there_isa PART_1 where(  
      Number = .nbr and  
      Name = .name  
    )  
    and  
    there_isa PART_2 where(  
      Number = .nbr and  
      Color = .color and  
      Weight = .weight  
    )  
  ));

do Change so_that( not(  
    there_isa PART_1 where( Number = 103 )  
    and  
    there_isa PART_2 where( Number = 103 )  
  ));

do Change so_that(  
    there_is_no PART_1 where( Number = 103 )  
    or  
    there_is_no PART_2 where( Number = 103 )  
  );

This is probably not what is desired here: surely the idea is to get rid of both of the records, not just one of them (non-deterministically), that being the minimum-effort way to make this Change happen. Instead, the solution is to specify exactly how to deal with this kind of deletion:
define RECORD_CLASS VPARTV as_a_view_where(...)
with_deletes_using(
  for_each [ INT .nbr, STR .name, STR .color, FLT .weight ]
  conclude(
    there_is_no VPARTV where(
      Number = .nbr and
      Name = .name and
      Color = .color and
      Weight = .weight)
  iff(
    there_is_no PART_1 where(
      Number = .nbr and
      Name = .name)
  and
    there_is_no PART_2 where(
      Number = .nbr and
      Color = .color and
      Weight = .weight))
) using(all_keyed_record_existence_tests_pass)

So, in addition to the required as_a_view_where assertion, there is an optional with_deletes_using assertion argument (of the same general form as as_a_view_where arguments) that is used to specify exactly how to translate do Change so that( there_is_no ...);. In this example then, the specified transformation is to delete both component records underlying the view record.

Here again is the grammar for view definitions, including the current topic of discussion, alternate_view_spec:

\[
\text{ViewDef} :: \text{define RECORD_CLASS upper as_a_view_where( ViewEquivSeq )} \\
\text{alternate_view_spec ( ViewEquivSeq )} \star \\
\text{using( outside [ VblSpecSeq ]? } \\
\text{infer_bounds_for thing }? \\
\text{as_read_only }? \\
\text{has_variant_fields }? \\
\text{all_keyed_record_existence_tests_pass }? \\
\star \\
\text{ViewEquiv} \star \\
\star \\
\text{ViewEquivSeq} :: \text{ViewEquiv } \and \text{ViewEquiv } \star \\
\star \\
\text{ViewEquiv} :: \text{for_each SomeVblSpecs conclude( Desc iff( Asn ) )} \\
\star \\
\text{alternate_view_spec} :: \text{with_this_isas_using | with_from_sections_using} \\
\text{with_deletes_using | with_this_is_no_deletes_using} \\
\text{with_adds_using | with_updates_using }
\]

Here is the algorithm used to select which alternate_view_spec to use in processing a use of the
corresponding view table.

1. In the non-update situation, a query `there_isa` that uses `from_section` is resolved by `with_from_sections_using`, if there is one, or else it fails.

2. In the non-update situation, a query `this_isa` is resolved by `with_this_isas_using`, if there is one, else by `as_a_view_where`.

3. In a literal Cymbal update situation (i.e., `do Change so that( this_isa )`), then a query `this_isa` is resolved by `with_updates_using`, if there is one, else by `with_this_isas_using`, if there is one, else by `as_a_view_where`.

4. In an adds situation, a query `there_isa` is resolved by `with_adds_using`, if there is one, else by `as_a_view_where`.

5. In a delete situation, a query `there_is_no` is resolved by `with_deletes_using`, if there is one, else by `as_a_view_where`.

6. In a delete situation, a query `this_is_no` is resolved by `with_this_is_no_deletes_using`, if there is one, else by `with_this_isas_using`, if there is one, else by `as_a_view_where`.

Keep in mind though that this algorithm can well settle on a strategy that will not work and thus that situation will serve as a prompt to find one that will work. This is subtle business. For examples, see `vu.*.IQ*` in the test suite.

IMPORTANT: Both `with_this_isas_using` and `with_this_is_no_deletes_using` are required when defining a view whose base tables are horizontally partitioned. Furthermore, when writing the assertion argument of `with_this_isas_using` and `with_this_is_no_deletes_using` in this situation, do NOT mention any hparti attributes in the RHS of the `iff`: they are simply not relevant and if mentioned, will likely cause hard-to-understand error messages.

### 15.4 In-line FUNCTIONS

At times, it can be very convenient when applying a function to some arguments, to have the definition of the function right there in-line at the point of invocation. Cymbal offers that functionality in two flavors: first, the case (which is called lambda OPCOFUN) wherein the in-line function definition offers a term pattern to be instantiated to create the result of the function invocation and secondly, what is simply more general, the case (either called definite description or arbitrary choice) wherein an in-line arbitrary assertion is used to compute the function return value.

#### 15.4.1 In-line FUNCTIONS: lambda OPCOFUNS

The lambda OPCOFUN is a declaratively defined FUNCTION in the same spirit that an OPCOND is a declaratively defined PREDICATE: the OPCO designation is short for open-coded in that the definition of the FUNCTION appears out in the open at the point of function invocation. (And the "lambda" adjective reflects the similarity to the lambda notation central to functional programming languages.) The essence of a lambda OPCOFUN is that it is an anonymous FUNCTION that is defined like a macro by saying how to form a term from its arguments that is in fact equal to the value of the function (opcofun.1.Q):
set .z = (3,2);  
_Show_Exp_To(.z)  
when( (3,2) > .z )
{
  do Write_Line( "Success" );
}

The output is:

  .z = 6
  Success

In this example, the first OPCOFUN is (3,2) which is the anonymous FUNCTION (as signified by \(\sim\) indicating a missing function name) that maps its pair of arguments to their multiplicative product. The assignment shows how this lambda OPCOFUN is applied to a couple of arguments: note that in this application (or invocation), the OPCOFUN appears in the same place that a FUNCTION name would.

The evaluation of a lambda OPCOFUN proceeds essentially by macro expansion: the invocation is replaced by the term that results by replacing all values of the parameter VBLs with their corresponding argument terms in the term that defines the lambda OPCOFUN value. Note the second example illustrates that the OPCOFUN can optionally be enclosed in parentheses when being applied to arguments. Please note that the term identified as being the (prototype) value for an OPCOFUN must be enclosed in parentheses.

There are a few caveats. First, lambda OPCOFUNs are not first-class citizens because with the exception of the apply function described next, a lambda OPCOFUN cannot appear in Cymbal unless it is part of a function application. For example, it cannot be assigned as the value to a VBL or passed as the argument to a FUNCTION. This is necessarily the case because lambda OPCOFUNs are handled/expanded during Tracy’s compilation of Cymbal code and so, they do not exist at run-time. This is part of the price of working with a compilation-based system as opposed to an interpreted language system. Secondly, when used in a declarative context, all of a lambda OPCOFUN’s arguments must be ground -- it cannot generate any values for variables. Lastly, Daytona does not support the user providing types for lambda OPCOFUN parameter VBLs.

To illustrate how a lambda OPCOFUN might be used to provide a programming convenience for the user, consider the following example which shows how to use a ds_m4 macro to introduce a function-composition function into Cymbal:

    // from sys.macros.m (the user gets this definition automatically)
    _Define_(compose, @<\(\sim\)(.x)=($1( $2( .x )))>@)
    // example use from a user Cymbal program
    do Write_Line( compose(str_for_heka, heka_for_str)( "54321" ) );

lambda OPCOFUNs may seem a little arcane at first but there are occasions when they are the right solution for the problem at hand, as will be seen when they are used with apply.
15.4.1.1 apply

In languages like LISP where functions are first-class citizens, the apply (or map) function provides a convenient service by enabling the user to specify the application of a function to lists of arguments without actually writing out all those applications out explicitly. For example, observe how tedious it is to write the following concat and how much more pleasant it is to write the equivalent apply invocation (apply.1.Q), especially if the list of arguments is long:

```
concat( [ rb(.a,3,"0"), rb(.b,3,"0"), rb(.c,3,"0"), rb(.d,3,"0") ], "." )
concat( apply( ~(.x)=(rb(.x,3,"0")), [.a,.b,.c,.d] ), "." )
```

Once again, the macro flavor is apparent: during query translation, Tracy literally expands apply invocations by taking the FUNCTION first argument and replacing each element of the TUPLE second argument with a call to this FUNCTION with arguments provided by that element. The import for apply reveals exactly what kind of creature it is.

```
TUPLE FUN( OBJ FUN( OBJ ), TUPLE| OBJ ARRAY| VBL ) apply
```

As illustrated by the next examples, apply works with FUNCTIONs that take several arguments and with FUNCTIONs that are known by their names alone. It is also able to apply itself to dereferenced TUPLE-valued and conventional ARRAY-valued VBLs.

```
define INT FUN( TUPLE[ INT, INT ] .a ) plus_tuple
{
  return( .a#1 + .a#2 );
}
{ local: INT ARRAY[3] .totals
  TUPLE[ INT, INT ] ARRAY[3] .summands = [ [ 1, 2 ], [ 3, 4 ], [ 5, 6 ] ]
  set .totals = apply( ~(.x,.y)=(.x+.y), [ [ 1, 2 ], [ 3, 4 ], [ 5, 6 ] ] );
  do Write_Words( .totals );
  set .totals = apply( "plus_tuple", .summands );
  do Write_Words( .totals );
}
```

To offer a FUN name as the first argument to apply, a constant like `plus_tuple` or `plus_tuple`FUN will always work; the user may be able to get away with just the unhatted FUN name itself depending on the mood of the parser.

15.4.2 In-line FUNCTIONS: definite description via arbitrary choice

In general, a definite description is a construct that denotes a unique individual who is identified by a characterizing assertion; consider, for example, "the 44th President Of The United States". The initial development of this concept goes back to a paper by Bertrand Russell in 1905 called "On Denoting". Here is what a definite description looks like in situ in Cymbal:
Specifically, the definite description consists of a LIST-former preceded by `?!`. It evaluates to the presumed sole member of the LIST, which of course could be a TUPLE as well. The example above can be read as, some unique `.sname` that satisfies the assertion given.

Surprisingly, definite descriptions are not yet implemented. However, a close sibling, arbitrary choice, has been implemented. The syntax for arbitrary choice just elides the `!` as in:

```java
when(
    ?![ .sname : there_isa SUPPLIER where( Name = .sname and City = "Atlanta" ) ]
    = "Standard AG" )
{
    do Write_Line( "Yes, Standard AG is a SUPPLIER in Atlanta" );
}
```

The idea here is that the `?[ : ]` construct evaluates to some element of the indicated LIST. Since there is no point in allowing the selection of the most expensive-to-compute element, Daytona chooses the first one that it can generate (and so it does so without generating the entire LIST). The user may or may not be able to see or know which one that is. Consider these two examples from arbchoice.1.Q, as all of these are:

```java
do Write_Line( ?[ .sname : there_isa SUPPLIER where( Name = .sname ) ]
set .x = ?[ .part : there_isa PART
    where( Name = .part and Number = rand_int( 100 ) + 100 ) ]
```

Observe that the second example illustrates the fact that "arbitrary choice" does not mean "random choice". If the user wants "random choice", then the user needs to explicitly ask for it as is done in the second example. Both examples show how these terms can be used in procedural settings whereas the previous `when` example illustrates a declarative setting.

What should (and does) happen when the LIST associated with an arbitrary choice is empty? In the procedural case, an arbitrary choice becomes the value of a procedural VBL. So, a fortiori, if it cannot be assigned a value because its LIST is empty, then its value will be whatever its procedural defaults are. So, the result of executing:

```java
set .u3 = "no-one";
set .u3 = ?[ .s : there_isa SUPPLIER where( Number = 101 and Name = .s ) ];
do Write_Line(.u3);
```

is not `no-one` on a line but rather an empty line.

In the declarative case, an inability to compute an arbitrary choice results, as one should expect, in the containing satisfaction claim being false.
when( "Horton" =
    ?[ .s : there_isa SUPPLIER where( Number = 101 and Name = .s ) ] )
    { do Write_Line( "error: Horton can’t happen" ) }
else { do Write_Line( "yep, no arbitrary choice possible" ) }

The output says that no arbitrary choice is possible.

The next example shows that an arbitrary choice can be a TUPLE:

fet .tu ist( .tu = ?[ [ .s, .c ] :
    there_isa SUPPLIER where( Number >= 432 and Name = .s and City = .c )
] )
    { do Write_Words( .tu#1, "from", .tu#2 );}

Happily, arbitrary choice even works on SQL LISTs. Here it is capturing a maximum SUPPLIER Number and producing an arbitrary TUPLE from an SQL SELECT:

    set .maxnbr= ?[select max(Number) from SUPPLIER]$;
    do Write_Line(.maxnbr);
    do Write_Words( "#100: ", ?[ select * from SUPPLIER ]$ );
16. Path Recursive Queries Including Transitive Closure

Traditionally, transitive closure provides essentially a mechanism for doing indefinitely many joins of a table with itself. In Daytona, it is instead an arbitrary first-order logic assertion that is being repeatedly "joined" with itself and for which in addition, the user has considerable control over the manner in which this "joining" is done. Actually, while Daytona can be directed to compute the true transitive closure, typically it is asked to compute something related but different, i.e., selected nodes on specified paths associated with linear recursive predicates, but people tend to lump all this kind of functionality under the rubric of transitive closure anyway, probably just because it has such a catchy name. As a tool, linear recursive path queries are very useful for asking questions about such graphs as organization charts and IP networks. And furthermore, it is simply Daytona’s mechanism for doing linear recursions of many but not all kinds.

16.1 Defining Transitive Closure

As an area of interest, transitive closure can be seen to be generalization of the following simple fundamental kind of question: Who is related to who?. The canonical example is that of the child-to-parent binary relationship as in

\[
\begin{align*}
\text{Maybelle Is A Child Of Pandora} \\
\text{Pandora Is A Child Of Ray} \\
\text{Ray Is A Child Of Steve} \\
\text{Maybelle Is A Descendant Of Steve}
\end{align*}
\]

Obviously, the last assertion is getting at the question of who is related to Steve, either directly or indirectly. That’s what the transitive closure AKA linear path recursion apparatus is all about: given a bunch of basic binary relationships, compute who is related to who, either directly or indirectly.

Consider the definition of transitive closure in an abstract mathematical setting, e.g., as relating to a binary relation \( \sim \) on the integers. While seemingly remote from its actual use in databases, this discussion in the abstract will best highlight the essence of the topic. The following table shows what few relationships there are for \( \sim \).

\[
\begin{array}{c|c|c|c}
\sim & 2 & 3 & 4 \\
2 & 3 & 4 & 5 \\
3 & & 4 & \\
4 & & & \\
3 & & & 6
\end{array}
\]

So, for example, \( 4 \sim 5 \) and \( 3 \sim 6 \). A binary relation \( R \) is said to be transitive if and only if for every \( x, y, z \) in the domain of \( R \), if \( x R y \) and \( y R z \), then \( x R z \). \( \sim \) is seriously not transitive as can easily be seen by choosing \( x = 2, y = 3, \) and \( z = 4 \) and noticing that it is not the case that \( 2 \sim 4 \).
So, the natural question arises as to what would have to be added to a non-transitive relation \( R \) to make it transitive? Certainly the containing binary relation that relates each integer to every other integer would be going too far for a binary relation \( R \) since it has so little of \( R \)'s character. In fact, what is desirable is to ask what is the \textit{transitive closure} of \( R \), i.e., what is the smallest relation \( R^* \) that is transitive and contains \( R \)?

For \( \sim \), that relation is:

\[
\begin{array}{c|c}
\sim^* \\
\hline
2 & 3 \\
3 & 4 \\
4 & 5 \\
3 & 6 \\
2 & 4 \\
2 & 5 \\
2 & 6 \\
3 & 5 \\
\end{array}
\]

Note that \([4, 6]\) and \([5, 6]\) are not in the transitive closure.

It turns out that (as is proved below in an equivalent context)

\[ R^* = \bigcup_{k=1}^{\infty} R^k \]

where \( R^1 = R \) and \( R^k = R^{k-1} \ast R \), where for two binary relations \( P \) and \( Q \) on the integers

\[ P \ast Q = \{ [i, j] : \text{there exists } z \text{ with } i P z \text{ and } z Q j \} \]

In fact, \( P \ast Q \) is the (relational equi-) join of the two tables for \( P \) and \( Q \) on the second coordinate of \( P \) and the first coordinate of \( Q \) plus a subsequent projection onto the two non-join attributes. Consequently, the indefinitely-many-joins concept is suddenly made apparent here since it can now be seen that the transitive closure \( R^* \) is the union of \( R \) with a sequence of join results formed by repeatedly joining (and projecting) the previous result with \( R \).

By identifying the presence of a certain graph, a graph-theoretic way to understand this indefinitely-many-joins result comes to light. Note that a binary relation can be thought of as representing the directed edges in a graph where the nodes of the graph are the elements of the two columns of the binary relation (seen as a two-column table) and a directed edge is said to exist from node \( a \) to node \( b \) iff \([a, b]\) is in the relation. Then, for binary relation \( R \), \( R^k \) is the set of all ordered pairs of integers which are linked by a path of length \( k \) through \( R \). For example, \( 2 \sim^3 5 \) since \( 2 \sim 3 \) and \( 3 \sim 4 \) and \( 4 \sim 5 \). This understanding of \( R^k \) is provable by a simple inductive argument: clearly, \( R^1 = R \) contains paths of length one between the nodes. Suppose for any \( k \) that the assertion holds. Then since \( R^{k+1} = R^k \ast R \), \([a, b]\) is in \( R^{k+1} \) if there is some \( c \) such that \([a, c]\) is in \( R^k \) and \([c, b]\) is in \( R \), i.e., if there is a path of length \( k + 1 \) between \( a \) and \( b \).

When \( R \) is finite, this union will not in fact be truly infinite because there will exist some \( k \) for
which all future $R^k$ do not contribute any new ordered pairs to the union (and may indeed be empty but not necessarily so). Specifically, if the underlying graph is acyclic, then at some $k$, there simply aren't any paths that are longer than $k$ (through a fixed finite number of nodes): hence all subsequent $R^k$ are empty. On the other hand, if the graph has cycles, then while the path length is unbounded, the number of nodes on any path is still bounded (like by the total number of nodes) and so at some point $m$, for all subsequent $k \geq m$, the two-tuples in $R^k$ will have already appeared in some $R^j$ for $j < m$, because otherwise, there would be an unbounded number of nodes. All this begins to hint as to why transitive closure is good for exploring graphs.

Actually, transitive closure, which has such a catchy name, is only a small part of what Cymbal offers. Cymbal begins by offering support for defining (some) linear recursive predicates using the full first-order language of logic (and then it goes far beyond that to add even more power). This is all related to computing the transitive closure, which actually is something users are typically not interested in. This is because the huge size of typical transitive closures indicate that they include a lot of relationships that are just not of interest for many practical problems. Indeed, recall, in the graph-theoretic context, the transitive closure contains precisely all those $[a, b]$ for which there is some directed path between them. So, if the user is only interested in reachability, e.g., what are all the nodes that terminate paths starting from given node $a$, then computing the full transitive closure would be computing quite a bit of irrelevant information.

### 16.2 Linear Recursive Predicates

The purpose of this mathematical section is to lay the groundwork for characterizing exactly what it is that Cymbal path PREDs are capable of computing. Here is a linear recursive predicate definition from logic programming where the object is to determine a binary relation $Vv$ such that for given finite, non-empty binary relations $Ra$ and $Sb$ satisfies this property called $\Phi$:

```plaintext
for_each [.x, .y] conclude(
    Vv[.x, .y] iff
    ( Sb[.x, .y]
        or there_exists .z such_that( Ra[.x, .z] and Vv[.z, .y] ))
)
```

As will be seen shortly, $\Phi$ is a meaningful property because it captures the notion of paths through graphs. For the moment though, notice that one $Vv$ that satisfies this $\Phi$ property is the complete graph on the set of all nodes referenced in the columns of $Ra$ and $Sb$. (The complete graph is the binary relation that contains both directed edges between any pair of nodes in the set of nodes.) Obviously, this is not a very useful solution but it is nice to have at least one solution. Any $Vv$ that has property $\Phi$ is called a fixed point since by the definition, $Vv = Sb \cup Ra \ast Vv$, $Vv$ is seen to be mapped to itself by the linear transformation analog that combines $Sb$ and $Ra$. So, in the interests of economy, one would want the smallest such binary relation, call it $Ud$, that satisfies $\Phi$; this would be called the least fixed point (LFP) of $\Phi$ and is considered to be the solution to the recursive equation $Vv = Sb \cup Ra \ast Vv$.

Does such a relation exist? It does, it is unique, and it can be written as:
Call this intersection $A$. Note that it is an intersection of a finite number of relations and that it exists because the complete graph is one such relation. Also, note that $A$ is non-empty because $A$ includes the assumed non-empty $S_b$. Here is a proof sketch as to why $A$ satisfies $\Phi$:

$$S_b \bigcup Ra^* A = S_b \bigcup Ra^* (V_v \text{ satisfies } \Phi)$$

$$= S_b \bigcup (V_v \text{ satisfies } Ra^* V_v)$$

$$= (S_b \bigcup Ra^* V_v)$$

$$= V_v \text{ satisfies } \Phi = A$$

And $A$ is clearly the least fixed point. Other than standard set-theory, this proof relies on the fact that:

$$(P \bigcap Q) \ast R = (P \ast R) \bigcap (Q \ast R)$$

the proof of which is left to the reader.

### 16.2.1 Relating The LFP To The Set Of All Paths

Now to develop the relationship between the LFP $Ud$ and paths in graphs. The claim is that:

$$Ud = \bigcup_{k=0}^{\infty} Ra^k \ast S_b = (I \bigcup \bigcup_{k=1}^{\infty} Ra^k) \ast S_b$$

where $I$ is the identity binary relation that $Ra^0$ is defined to equal. This implies that $Ud$ contains precisely all the nodes that can be reached from the nodes in the second column of $S_b$ by following a path of some finite length defined by joining $Ra$ to itself some finite number of times. Here is the form of such a path for $k = 2$:

$$Ra[ r_2, r_1 ] \text{ and } Ra[ r_1, s_2 ] \text{ and } Sb[ s_2, s_1 ]$$

To prove the claim, to show one direction, it suffices to show that this union of paths satisfies $\Phi$, i.e.,

$$\bigcup_{k=0}^{\infty} Ra^k \ast S_b = S_b \bigcup (\bigcup_{k=1}^{\infty} Ra^k \ast S_b)$$

To show the other direction, suppose $V_v$ is an arbitrary relation satisfying $\Phi$ and suppose that $[a, b]$ is in $Ra^k \ast S_b$ for some $k \geq 1$. Then there exists constants $c_i$ such that:

$$Ra[ a, c_k ] \text{ and } \ldots \text{ and } Ra[ c_2, c_1 ] \text{ and } Sb[ c_1, b ]$$

Then since $V_v$ satisfies $\Phi$, $V_v[ c_1, b ]$ and since that is true and $Ra[ c_2, c_1 ]$, then $V_v[ c_2, b ]$. Continue on in this fashion to conclude that $V_v[ a, b ]$. QED.
Notice how $Ud$ also has the flavor of mathematical induction: there is a base case ($Sb$) and then there is the rule for getting from one state to the next (via $Ra$). Indeed, an alternative approach here would be to use induction to define the transitive closure in a constructive way and relegate the least-fixed-point characterization to a curiosity.

### 16.2.2 Relating The LFP To The Transitive Closure

Observe that if $Sb$ is $Ra$, then $Ud$ is actually transitive because if $Ud[b,a]$ and $Ud[c,b]$, then the length $j$ path between $a$ and $b$ can be joined to the length $k$ path between $b$ and $c$ to form a length $j+k$ path between $a$ and $c$. This suggests that $Ud$ is in fact the transitive closure $Ra^*$. To prove this, take any relation $Q$ that contains $Ra$ and is transitive. Take any element $[b,a]$ of $Ud$. Then from the preceding result, there exists constants $c_i$ such that:

$$Ra[b, c_k] \text{ and } \cdots \text{ and } Ra[c_2, c_1] \text{ and } Ra[c_1, a]$$

Since $Q$ is transitive and contains $R$, one concludes immediately that $c_2 Q a$. Using that fact and continuing to apply transitivity, one concludes eventually that $b Q a$.

Thus for the class of problems that Daytona solves, the appellations transitive and linear recursive are equivalent.

### 16.2.3 Implicit Generality

Furthermore, all these proofs are sufficient to prove something much more general. First of all, all that is required of the types of $x$ and $y$ in the definition of $\Phi$ is that they be the same: in particular, they could both be TUPLES (of the same type). And secondly, there is nothing in the proofs that relies on $Sb$ and $Ra$ being tables; indeed the requirement was only that they be (finite) binary relations whose two columns have the same type. These relations could be implicitly defined by a Cymbal OPCOND:

$$\sim[.p,.q] \text{ such that } (\text{Logic}_\text{Asn}(.p,.q))$$

The Logic_Asn of course is any Cymbal assertion (that Daytona can generate values for), including perhaps other uses of linear recursion. Indeed, many single-recursive-predicate Prolog programs fall into this category. As an illustrative example, consider:

$$
P(x,y) \leftarrow S(x,y).
P(x,y) \leftarrow A(x,u), P(u,y), B(x,u).
P(x,y) \leftarrow P(u,y), C(u,y), D(x,u).
P(x,y) \leftarrow E(x,u), F(u,y), P(u,y).
$$

can be rewritten in this form:

$$P(x,y) \text{ iff } S(x,y)$$

$$\text{ or } (A(x,u) \text{ and } B(x,u) \text{ or } C(u,y) \text{ and } D(x,u) \text{ or } E(x,u) \text{ and } F(u,y)) \text{ and } P(u,y)$$

### 16.3 Linear Path Recursion / Transitive Closure In Daytona

As a top-level summary, Daytona’s linear path recursion hence transitive closure works with completely general Cymbal assertions instead of database tables and in its most general form, it works
with relations on pairs of TUPLEs instead of pairs of just scalars. Also, Daytona does not typically compute and store the whole transitive closure in one gulp as some algorithms do (although it can) but rather it answers the user’s questions about the transitive closure by exploring it in a selective depth-first-search way that is under substantial control of the user as to what to select, when to backtrack, when to stop, and how and what to sort. It is in this way that Daytona can be used to intelligently explore and answer questions about networks and graphs as well as to compute the results of linear path recursions. This is done by defining in Cymbal a path PREDICATE, which by the preceding discussion will be seen to have earned the name.

Specifically, given \( Ud \) as the linear recursive \( LFP \) of \( Vv = Sb \cup Ra \ast Vv \), the appropriate Cymbal for computing all of the pairs \([.x, .y]\) in \( Ud \) is:

\[
\text{define path PRED: Tc[ .u, .v ]}
\]

\[
\text{by_stepping_with( .u Ra .v )}
\]

\[
\text{using( with_identity );}
\]

\[
\text{with_no_duplicates}
\]

\[
\text{do Display each[ .x, .y ] each_time( .z Sb .y and .x Tc .z );}
\]

As per Chapter 5, the each_time argument above is shorthand for either of the following:

\[
\text{each_time(somehow( .z Sb .y and .x Tc .z ))}
\]

\[
\text{each_time( there_exists .z such_that( .z Sb .y and .x Tc .z )))}
\]

It’s necessary for the \( Sb \) conjunct to precede the \( Tc \) conjunct in the Display assertion above because in order for Daytona to compute \( Tc \), any use of \( Tc \) must have its second argument ground. This is in keeping with the convention used for other satisfaction claims that generate values for variables:

\[
.x = 7
\]

\[
.y = [ 4, 5 ]
\]

\[
.z \text{ Is_In } [ 1, 2, 3 ]
\]

\[
[ .a, .b ] = \text{tokens( for "11 13 17 19 23 29" upto " " )}
\]

\[
.c \text{ Is_A_Descendent_Of "Steve" // path PRED}
\]

Nonetheless of course Daytona considers the next two equalities to be equivalent, whether for generation or for testing:

\[
.x = [ 4, 5 ]
\]

\[
[ 4, 5 ] = .x
\]

However, in terms of conventional English, the first is preferred because the subject of the assertion is really \(.x\) and so it should appear first, before the verb phrase. That’s what’s happening with the convention for using path PREDs: when generating values, it’s the variable that is the subject of the assertion that is being defined (as a generator) and so it appears first.

The with_no_duplicates keyword is also necessary because the transitive closure is a set and Daytona’s mechanism for processing path PREDs generates all paths through the graph (from the set of starting nodes) and so those paths may find their way through the same node more than once.

Furthermore, in order for Dayton to compute with \( Tc \), \( Tc \) must be defined with an \( Ra \) assertion.
that, given \( u \) Ra \( v \), will be able to generate one or more values for \( u \) given a value for \( v \). And as before, \( u \) and \( v \) must have the same types, possibly TUPLES. These two restrictions characterize the linear recursions that Daytona is able to process. The \textit{with\_identity} keyword is what will get \( Sb \) printed. Note that if \( Sb \) were \( Ra \), this is the transitive closure \( Ud = Ra^* \). The point to take away is that Daytona realizes that all that is necessary to specify transitive closure is to specify \( Ra \) alone, which must satisfy the constraints above.

Here is how this paradigm is used to compute the transitive closure of the relation \( \sim \) that introduced this chapter, where the directed edges of the \( \sim \) graph go from the first column to the second (which is the opposite for \( Ra \), \( Sb \), and \( Ud \) above):

\[
\begin{align*}
\text{define path PRED Tc_til[ .v, .u ]} \\
& \text{// note reversal of } u/v \text{ from line above} \\
& \text{by_stepping_with( [.u, .v] Is_In .tilde );} \\
& \text{set .tilde = { [2,3], [3,4], [4,5], [3,6] :: with_sort_spec[1] };} \\
& \text{// prints the entire transitive closure} \\
& \text{with_format _table_} \\
& \text{in_lexico_order} \\
& \text{with_no_duplicates} \\
& \text{do Display each[ .a, .b ]} \\
& \text{each_time( [.a, .z] Is_In .tilde and .b Tc_til .z with_identity );} \\
& \text{// just prints the new entries added to get transitivity} \\
& \text{with_format _table_} \\
& \text{in_lexico_order} \\
& \text{with_no_duplicates} \\
& \text{do Display each[ .a, .b ]} \\
& \text{each_time( [.a, .z] Is_In .tilde and .b Tc_til .z );}
\end{align*}
\]

Note that the keyword \textit{with\_identity} can appear in the call to the PRED \( Tc\_til \), instead of its definition, with the same effect. This query, which is in proof.trans.2.Q, produces this confirmatory output:
Note that in this example, the \texttt{with\_identity} keyword has been removed from the definition of \texttt{Tc\_til} so that it can be optionally included with the same effect in any \textit{satisfaction claim} that uses \texttt{Tc\_til}. Note that when \texttt{with\_identity} is used, it results in producing the entire transitive closure whereas without it, only the new tuples that are found necessary for creating transitivity are printed.

And here is a simple query that demonstrates that \texttt{Tc\_til} is transitive (proof.trans.2.Q):

\begin{verbatim}
set .tilde = { [2,3], [3,4], [4,5], [3,6] :: with_sort_spec[1] };
fet .middle ist(
    [.a, ?] Is_In .tilde and
    .middle Tc_til .a
    and .last Tc_til .middle
    and ! .last Tc_til .a
){
    do Write_Line(
        "error: found intransitivity for Tc_til node .middle"ISTR );
}
\end{verbatim}

Of course, there is no output from this query!

So then, consider how Daytona would generate all \texttt{x} such that \texttt{x Ud 5} for \texttt{Ud} as above.
define path PRED: Tc[ .u, .v ]
    by_stepping_with( .u Ra .v )
    using( with_identity )
)
    do Display each[ .x ] each_time( .z Sb 5 and .x Tc .z );

The Daytona predicate definition says that $Tc$ is the transitive closure $Ra^*$ of $Ra$ with identity, meaning $Tc = I \cup Ra^*$. If the with_identity keyword was omitted, then $Tc$ would equal the true transitive closure $Ra^*$.

The appropriate Cymbal for testing if $8 Ud 23$ is:

define path PRED: Tc[ .u, .v ]
    by_stepping_with( .u Ra .v )
    using( with_identity )
    do Display each[ .tval ]
        each_time( .tval = truth( .z Sb 23 and 8 Tc .z );

The point of these three $Ud$ examples is to show that the same basic Cymbal expression can be used to generate or test values, depending on which terms are ground. This is typical declarative flexibility. There is still a fourth example to discuss.

However, in order to keep from having to repeat the definition of $Tc$ every time it is used in a query, it suffices to put the definition in \texttt{usr.env.cy} (or one of the other \texttt{*env.cy}) with the user's other global Cymbal definitions.

So, given that the definition of $Tc$ is in \texttt{usr.env.cy}, consider the fourth grounding $Ud$ possibility, the question of which values of $x$ are such that $5 Ud x$ is phrased as:

$$\text{do Display each[ .x ] each_time( .z Sb .x and 5 Tc .z );}$$

Notice however that in each of the above 4 cases, the second argument to the transitive closure predicate $Tc$ does not contain the first lexical occurrence of any variable, i.e., that it is ground, it is not a defining occurrence. This means that by the time Daytona begins to process the $Tc$ conjunct, it has a known set of candidates to consider for being the second argument to the $Tc$ satisfaction claim.

This is absolutely critical since Daytona processes transitive closure satisfaction claims by what is called "forward-chaining", which in this case, is done by depth-first search (as opposed to breadth-first or whatever). In effect, for each second argument to $Tc$ that needs to be considered, Daytona creates a certain tree and then searches it depth-first, visiting the parents before each of their children. This tree is constructed as follows: For each $a$, define $f$ such that

$$f( a ) = \{ x : x Ra a \}$$

Then for each initial value $a$, the first level of the tree consists of the elements of the LIST $f(a)$. The children in the tree of each of those $b$ in $f(a)$ are the elements of $f(b)$. This is continued recursively until applying $f$ to given values produces the empty set or else when the user has arranged for Daytona to determine it should stop. (Incidentally, the process may never stop; the user must know enough about the problem to guarantee, with or without Daytona's help, that there will be no infinite loops.)
Since it is just doing a depth-first search, it should be clear that Daytona is 100% efficient at enumerating paths in this abstract graph; however, it may visit a single node in the graph more than once if that node is on more than one path in the tree leading from the starting nodes. Anyway, it is the construction and exploration of this tree that proves that Daytona is computing $Ud$.

Since clearly this tree exploration cannot begin until there is a LIST of values to try, the hard and fast rule is that a linear recursive path predicate cannot be used in Cymbal in such a way that its second argument contains the lexically first (defining) occurrence of any variable.

It must be pointed out though that this is primarily a syntactic constraint which can be arranged to hold without loss of generality in many practical situations just by redoing some definitions. Consider the next two examples. The following data is taken from the EXAMPLES/usr/order database with the PERSON RECORD CLASS’s FIELDS being Name and Children, the latter being SET-valued.

Steve:[Ray:Roger:Romy]
Ray:[Paula:Pat:Penny:Pandora]
Paula:[Mindy:Max]
Roger:[Pablo]
Romy:[]
Penny:[Martin:Milton]
Pandora:[Maybelle]

The first query (steve.1.Q) shows how to get the descendants of Steve:

```scheme
do Display each[ .desc ]
  each_time( .desc Is_A_Descendant_Of "Steve" );
define path PRED: Is_A_Descendant_Of[ STR .x, STR .y ]
  by_stepping_with(
    there_is_a PERSON named .y where( one_of_the Children = .x )
  )
```

Suppose though one wanted to get all the ancestors of Maybelle. The syntactic constraint above forbids trying:

```scheme
do Display each[ .anc ]
  each_time( "Maybelle" Is_A_Descendant_Of .anc );
```

However, by doing a redefinition, the following (maybelle.3.Q) works:

```scheme
do Display each[ .anc ]
  each_time( .anc Is_An_Ancestor_Of "Maybelle" );
define path PRED: Is_An_Ancestor_Of[ STR .x, STR .y ]
  by_stepping_with(
    there_is_a PERSON named .x where( one_of_the Children = .y )
  )
```

The two by_stepping_with assertions are not so similar that they will be processed the same way. In the case of Is_A_Descendant_Of, the processing goes from a parent to their children whereas the opposite is the case for Is_An_Ancestor_Of. So obviously, this rewriting can only work if the assertion can
generate values in both processing directions.

The handling of the preceding "fourth case" above (i.e., $5 \ \ Ud \ \ .x$) offers another technique.

16.4 Path Predicates Are Boxes Too!

Actually, Daytona’s transitive closure is an elaboration of the box concept: as the nodes of the tree defined above are generated, by default, they are put into a box, and indeed, unless specified otherwise, this box is a LIST of TUPLES, where each TUPLE represents a node in the graph. Consequently, Daytona’s transitive closure inherits the use of many if not all of the box keyword arguments. The precise syntax permissible is given for Any_Path_Pred in sys.env.cy. There are keywords such as with_identity that can only be used when defining path PREDs and then there are selected keywords taken from both Is_In and box() that can be used as well.

PRED: Any_Path_Pred[
    VBLSPEC | TUPLE[ ( 1-> ) VBLSPEC ],
    VBLSPEC | TUPLE[ ( 1-> ) VBLSPEC ],
    ( 1 ) by_stepping_with ASN,
    ( 0->1 ) backtracking_when ASN,
    ( 0->1 ) stop_finding_children_when ASN,
    ( 0->1 ) with_distance_vbl manifest alias INT,
    ( 0->1 ) with_child_nbr_vbl manifest alias INT,
    ( 0->1 ) with_outcount_vbl manifest alias INT,
    ( 0->1 ) with_path_vbl manifest alias STR(*),
    ( 0->1 ) with_identity,
    ( 0->1 ) given_acyclic,
    ( 0->1 ) outside TUPLE[ ( 1-> ) VBLSPEC ], // allowed only in defn

    /** from Is_In[] **/
    ( 0->1 ) in_selection_order,
    ( 0->1 ) in_reverse_selection_order,
    ( 0->1 ) in_arbitrary_order,
    ( 0->1 ) in_lexico_order,
    ( 0->1 ) in_reverse_lexico_order,
    ( 0->1 ) in_random_order,
    ( 0->1 ) sorted_by_spec manifest TUPLE[ ( 1-> ) manifest INT
        : with_specifiers( (0->1) as_unique_key,
            (0->1) with_sort_fun INT FUN( alias TUPLE, alias TUPLE ) ) ]),
    ( 0->1 ) sorted_by_directions manifest TUPLE[ ( 1-> ) _3GL_TEXT ],

    ( 0->1 ) with_candidate_index_vbl manifest alias INT,
    ( 0->1 ) with_selection_index_vbl manifest alias INT,
    ( 0->1 ) with_sort_index_vbl manifest alias INT,

    ( 0->1 ) with_selection_index INT, /* could be _last_ */
A VBLSPEC is any optionally typed variable dereference as illustrated by .x and STR .yy. An ASN is any (declarative) assertion in parentheses. The exact effect of with_identity is that it causes Daytona to put the starting node(s) on the paths in the transitive closure box. If a selecting_when assertion is used, then only when the current node satisfies that assertion, does it get added to the box.

Using a box keyword, here is how to sort the descendants of Steve:

```vbnet
do Display each[ .desc ]
   each_time( .desc Is_A_Descendant_Of "Steve" in_lexico_order );
define path PRED: Is_A_Descendant_Of[ STR .x, STR .y ]
   by_stepping_with(
      there_is_a PERSON named .y where( one_of_the Children = .x )
   )
   using( with_identity )
```

Note that keyword arguments used in the PRED definition are grouped together by a using keyword. In this case, the inclusion of with_identity implies that Steve will appear in the answer, which, while counter-intuitive because Steve is not a descendant of himself, nonetheless suffices to illustrate a use of using.

Transitive closure boxes maintain a distinction between the definition of the BOX/PRED and any one of its uses (i.e., calls). While the by_stepping_with keyword can only and must be used in the path PRED definition, the other keywords can appear either in the definition or at the site of use. When the same keyword appears both in the definition and the use, the one in the call takes precedence. Otherwise the keywords accumulate and are treated as an ensemble like they would in a regular box, which implies that they cannot conflict, as would occur, for example, when two different sorting criteria are explicitly given.

PERSON and its variants were the original tables used to test out path predicates. They are very
small and artificial while nonetheless remaining useful for testing purposes. It’s a lot more fun though to work with the bigger and richer MARRIAGE and ROYAL. As introduced in Chapter 15, MARRIAGE is a table of English monarchy marriage info and ROYAL provides additional information about some of the royals referred to in MARRIAGE. This is an extremely interesting dataset with some 47 marriages from William The Conqueror to Elizabeth I. Perhaps even more will be added as time goes by.

The next query shows how to use keywords to select and arrange the ancestors of Elizabeth I (decl.genea.2.Q).

```plaintext
define path PRED[ TUPLE[ STR .x, INT .yom ], TUPLE[ STR .y, INT .yom0 ] ]
Is_A_Royal_Ancestor_With_Yom_Of
by_stepping_with(
    there_isa MARRIAGE where( one_of_the Children = .y and Year_Married = .yom )
    and (this_isa MARRIAGE where( Husband = .x )
        or this_isa MARRIAGE where( Wife = .x ))
)
using( sorted_by_spec[ -2, 1 ] )

fet [ .si, .lizanc, .yom ]
list( [ .lizanc, .yom ] Is_A_Royal_Ancestor_With_Yom_Of [ "Elizabeth I", 0 ]
    with_no_duplicates with_sort_index_vbl si
){
    do Write_Words( .si, .lizanc, .yom );
}
```

Here is the output of this query:
The very first thing to notice about this query is that this path PRED works with TUPLeS for nodes, instead of the scalars as seen earlier. In this case, the node consists of the name of the ROYAL and the year in which that person was married. Notice that keyword arguments are used both in the
call and in the PRED definition. Obviously, the ones in the PRED definition are inherited by any use of the PRED, whereas the ones in the call are local customizations. The cumulative effect here is to sort the ancestors by marriage year going back in time to William I.

The keyword with_no_duplicates changes the character of the default LIST BOX to be that of a SET. Its use is warranted here because of the War Of The Roses: in 1328, the marriage of Edward III Plantagenet and Philippa of Hainault produced the eventual patriarchs of the houses of Lancaster and York. These two houses, whose emblems were a red rose and white rose resp., battled for the monarchy for 30 years. The houses were united again via the marriage of the Lancastrian Henry VII to Elizabeth of York. The point of this digression is that via Henry VIII, Elizabeth I is a granddaughter of Henry VII, thus implying that beginning with Edward III Plantagenet and his ancestors, this query will produce duplicates as it explores all paths back from Elizabeth I. Clearly, by using with_no_duplicates, these duplicates are eliminated from the query output.

The keyword outside plays the same role for path PREDs as it does for macro PREDs, only for path PREDs, its use is optional, although recommended. The keyword outside can only appear in the definition.

As a cautionary tale, consider this query, which takes a run at generating all of the even natural numbers (even.2.Q).

```plaintext
define path PRED [ INT .x2, INT .x1 ]: Is_Even_Reachable_From
    by_stepping_with( .x2 = .x1 +2 )
using(
    with_identity
    stopping_when( .x2 >= 50 )
)
define PRED[ INT .x ] Is_Even iff( .x Is_Even_Reachable_From 2 )
fet .x Is_Even
do {
    _Show_Exp_To(.x)
}
```

Obviously, without the stopping_when ancillary assertion, this query would attempt to run forever. This should make it clear that Daytona allows users to write valid declarative queries which go into infinite loops, as well as invalid declarative queries which can go into infinite loops. Caveat emptor. The only support Daytona offers against such unhappiness is that by default, Daytona will instrument the code with an additional check that prevents infinite loops that would arise due to cycles in the node graph. This check and its overhead can be eliminated by using the given_acyclic keyword. There are no cycles in the even natural numbers though and so, for that case, the user must take explicit steps to control the computation.

For the record, here is a technical constraint on ancillary assertions: Any non-ancillary, implicitly-scoped (hence local and not outside) VBL is fine in the PRED definition but not in the call, where it must be explicitly scoped in order for the query to be compiled. This is not a constraint that is likely to cause anyone any trouble because ancillary assertions tend to be simple. steve.plus.Q has examples.

Also, the ancillary assertions can only make reference to the values of the node variables just
computed, which syntactically means that it is OK to reference the variables in the left-half of the path PRED’s arguments but not the right-half.

16.5 Computing Functions Using Path PREDs

As it turns out, Daytona’s path PREDs are capable of computing quite a few (but certainly not all) recursive functions. This is entertaining, educational, and potentially useful, if the problem to be solved is of interest. There are two classes of recursively-defined functions that the Cymbal path PRED feature is known to be able to solve: the class with tail recursions and the class defined by a moving-window type recurrence relation. The utility of this is increased by the fact that many non-tail recursions can be reformulated as equivalent tail recursions, often by adding additional helper variables. Nonetheless, this mechanism, which was never intended to be able to compute any functions, can still fail to be able to compute some.

Also, as will be seen, the syntax for doing these computations is a bit awkward (although effective) but that is to be expected because as a logic language, Cymbal is designed to work in the context of assertions, the most primitive of which being satisfaction claims. In other words, without adding new syntax (and a new approach) to the language, Cymbal has to approach defining functions by using assertions to define them, not by working with them directly. Actually, Cymbal’s approach is the same taken in set theory where a function is defined as being a constrained binary relation relating domain elements to range elements in such a way that there are no two-tuples that have the same domain element.

On the other hand, functional programming, as would be expected from the name, can concisely express the definitions of a wide range of recursive functions. And therefore, one would not expect a functional language to be particularly good at expressing quantified logic assertions. All this being said though, the examples below are still educational and entertaining, because they illustrate the use of BOX keywords in specifying how to process a chain of nodes (not a tree or forest).

Happily, the factorial function is easily computed both ways. Here is a tail-recursive way to compute it (recur.facto.1.Q).
define path PRED[
    TUPLE[ INT(_huge_) .n2, INT(_huge_) .f2 ],
    TUPLE[ INT(_huge_) .n1, INT(_huge_) .f1 ] ]

Factorial_Of
    by_stepping_with(
        if( .n1 = 1 ) then( [ .n2, .f2 ] = [ 1, 1 ] )
        else( [ .n2, .f2 ] = [ .n1-1, .n1*.f1 ] )
    )
using(
    given_acyclic
    with_identity
    stopping_when( .n2 = 1 )
    selecting_when( .n2 = 1 )
)

fet [.i, .f ] ist(
    .i Is_In [ 1, 2, 4, 8, 12, 16 ] and
    [ ?, .f ] Factorial_Of [ .i, 1 ]
)
{
    do Write_Words( .i, .f );
}

The essence of tail recursion is that the body of the recursive procedure is set up to return either a constant (for the base case) or else just directly the result of a carefully constructed smaller problem of the same form as the original. That’s what’s happening here: in creating and going from one node to the next, Daytona is proceeding like this:

\[
\begin{align*}
[ n, 1 ] &\rightarrow [ n-1, n*1 ] &\rightarrow [ n-2, (n-1)*n*1 ] &\rightarrow \ldots &\rightarrow [ 1, 2*3*\ldots n*1 ]
\end{align*}
\]

Note that the usual definition of factorial, i.e.,

\[ \text{ facto}(n) = n \cdot \text{ facto}(n-1) \]

is not tail-recursive but can be rewritten as above to be tail-recursive by adding a variable to carry along intermediate results.

On the other hand, a moving-window recurrence relationship is characterized by:

\[ f_n = g(f_{n-1}, f_{n-2}, \ldots, f_{n-k}) \]

for fixed \( k \geq 1 \) and for all \( n > k \) and of course, some function \( g \) and initial values for the first \( k \) values of \( f \). Cymbal path PREDs can handle these computations simply by working forwards, instead of backwards as in the preceding example, and by carrying along the \( k \) previous values to use in constructing the next node. That is trivially seen in this alternate mode of computing the factorial (recur.facto.2.Q):
define path PRED[
    TUPLE[ INT(_huge_) .idx2, INT(_huge_) .f2 ],
    TUPLE[ INT(_huge_) .idx1, INT(_huge_) .f1 ] ]

Factorial_From
    by_stepping_with(
        .idx2 = .idx1 +1
        and .f2 = .idx2*.f1
    )
    using(
        given_acyclic
        with_identity
        with_candidate_index_vbl ci
        selecting_when( .ci = .n )
        stopping_when( .ci = .n )
        outside[ INT .n ]
    )

    fet [.n, .f ] ist(
        .n Is_In [ 1, 2, 4, 8, 12, 16 ] and
        [ ?, .f ] Factorial_From [ 1, 1 ]
    ){
        do Write_Words( .n, .f );
    }

Here the construction of the nodes proceeds like this:

    [ 1, 1 ] -> [ 2, 2*1 ] -> [ 3, 3*2*1 ] -> ... -> [ n, n*n-1* ... *3*2*1 ]

Note that for both of these computations, only one node is stored in the path PRED box.

    The Fibonacci sequence illustrates a case when k is 2 (recur.fibo.4.Q):
define path PRED[
    TUPLE[ INT(_huge_) .f_n, INT(_huge_) .f_prev ],
    TUPLE[ INT(_huge_) .f_n_1, INT(_huge_) .f_n_2 ]
] Is_Fibonacci_From
by_stepping_with(
    .f_prev = .f_n_1 and
    .f_n = .f_n_1 + .f_n_2
)
using(
    given_acyclic
    with_candidate_index_vbl ci
    selecting_when( .ci = .n )
    stopping_when( .ci = .n )
    outside[ INT .n ]
)

fet [ .n, .f ]
ist( .n Is_In [ 5 -> 50 ]
    and [ .f, ? ] Is_Fibonacci_From [ 1, 0 ]
)
{
    do Write_Words( ".n -> .f"ISTR );
}

Moving on to a couple of examples of the handling of tail recursion, recall that Euclid’s algorithm for computing the greatest common divisor is expressed by:

\[
gcd(a, b) = \text{if} \ a = 0 \ \text{then} b \ \text{else} \ gcd(b \% a, a)\] 

Here is how to compute the gcd using Euclid’s algorithm and to further use that, by means of a macro PRED, to reduce a few fractions (decl.gcd.5.Q):
define path PRED[
  TUPLE[ INT(_huge_) .a2, INT(_huge_) .b2 ],
  TUPLE[ INT(_huge_) .a1, INT(_huge_) .b1 ] ] Gcd_Euclid
by_stepping_with(
  if( .a1 = 0 ) then( [ .a2, .b2 ] = [ .a1, .b1 ] )
  else( [ .a2, .b2 ] = [ .b1 % .a1, .a1 ] )
)
using(
  with_identity // this is how to handle input of [ -18, 0 ]
given_acyclic
  stopping_when( .a2 = 0 )
  selecting_when( .a2 = 0 )
)

define PRED[ TUPLE [ INT(_huge_) .numer, INT(_huge_) .denom ] .redu,
  TUPLE [ INT(_huge_) .numer, INT(_huge_) .denom ] .orig ]
Is_Reduced_Fraction_For
iff(
  [ ?, .gcd ] Gcd_Euclid [ .orig#1, .orig#2 ]
  and .redu = [ .orig#1 / .gcd, .orig#2 / .gcd ]
)

fet [ .i, .j, .gcd ] ist(
  [.i, .j ] Is_In [ [33, 57], [3,12], [108,258], [0,-14], [-18,0], [1234, 5678],
  [2310, 10010], [ 987654321, 123456789 ] ]
  and
  [ ?, .gcd ] Gcd_Euclid [ .i, .j ]
){
  do Write_Words( .i, .j, "have gcd", .gcd );
}

fet [ TUPLE[(2)INT(_huge_)] .orig_tu, TUPLE[(2)INT(_huge_)] .redu_tu ]
ist(
  .orig_tu Is_In [ [33, 57], [3,12], [108,258], [0,-14], [-18,0], [1234, 5678],
  [2310, 10010], [ 987654321, 123456789 ] ]
  and .redu_tu Is_Reduced_Fraction_For .orig_tu
){
  do Write_Words( .orig_tu, "reduces to", .redu_tu );
}

The last example illustrating the processing of tail recursion shows the use of the Newton-Raphson method of computing the square roots of FLOATs (recur.root.1.Q):
define PRED[ FLT .r, FLT .sq ] Is_Sq_Root_Of
iff( there_exists FLT .accur such_that( .accur = .00001 and
    [ .r, ? ] Is_Next_Sq_Root_Estimate_From [ 1.0, 1.0 ] // #2 can’t be 0.0!! )
)

define path PRED[
    TUPLE[ FLT .r2, FLT .err2 ],
    TUPLE[ FLT .r1, FLT .err1 ] ]
Is_Next_Sq_Root_Estimate_From
by_stepping_with(
    .r2 = (.sq/.r1 + .r1)/2.0
    and .err2 = abs( .r2 - .r1 )/.r1
)
using(
    given_acyclic
    stopping_when( .err2 < .accur )
    selecting_when( .err2 < .accur )
    outside[ FLT .sq, FLT .accur ]
)

fet [ .r, .sq ]
ist(
    .sq Is_In [ 2.0, 3.0, 4.0, 5.0, 7.0, 9.0, 16.0 ]
    and .r Is_Sq_Root_Of .sq
)
{    do Write_Words( "sqrt(", .sq, " ->", .r );
}

16.6 Additional Constructs To Control The Path Search

The remaining example queries show how additional control information can be used to control
the path recursion search. This control information is specified by non-BOX-related, path-PRED-specific
keywords. For trans.2.Q,
define path PRED: Is_A_Remote_Descendant_Of[ .x, .y ]
by_stepping_with(
    there_is_a PERSON named .y where( one_of_the Children = .x )
)
using(
    with_distance_vbl d
    selecting_when( .d > 2 )
)
do Display
each[ .remote ]
each_time( .remote Is_A_Remote_Descendant_Of "Steve" )

Notice how no datatype information is given explicitly for the path recursion variables in this example. This is usually acceptable since Daytona’s type inference mechanism is almost always adequate for the task. Here the with_distance_vbl keyword argument introduces a variable chosen by the user for holding the values of the current distance from the root as the search progresses. This value then can be used in a number of different assertions so as to control the search. For example, the selecting_when keyword argument is an assertion that says which nodes of the tree are to give values for the answers: by default, all the (graph) nodes that the path recursion algorithm visits are stored in a box. When selecting_when is used only those nodes satisfying that assertion are stored in the box. In this case, .remote is only given the values of those descendants which are a distance greater than 2 from the root. Note that with all of these modifications, Is_A_Remote_Descendant_Of is not the transitive closure of anything although it is based on the transitive closure of something.

Raising the stakes some (nearsteve.2.Q):
do Display
each[ .near_leaf ]
each_time( .near_leaf Is_A_Bounded_Leaf_Of "Steve" )

define path PRED: Is_A_Bounded_Leaf_Of[ STR .x, STR .y ]
by_stepping_with(
    there_is_a PERSON named .y where( one_of_the Children = .x )
)
using(
    with_outcount_vbl nbr_succ
    selecting_when( .nbr_succ = 0 or .d = 2 )
    with_distance_vbl d
    backtracking_when( .d = 2 )
)

The with_outcount_vbl keyword argument specifies a variable to hold the number of children of each node in the tree. The backtracking_when keyword argument is an assertion that, when true,
instructs the search mechanism to stop searching down its current path in the tree and to continue on with any remaining search paths. In other words, since Daytona’s path recursion box construction is accomplished by means of using a stack to do depth-first search, the current node \( A \) on the stack (corresponding to the last element put in the box) will be popped off the stack and the next candidate considered for pushing onto the stack and inclusion in the box -- instead of attempting to find a node to push on top of that original node \( A \). Backtracking, if any, occurs after the current node is made available for selection into the box. Contrast this with the *stopping when* keyword which instructs the search mechanism to terminate the entire search immediately when its argument assertion is true. Any stopping occurs after the current node is made available for selection into the box. (The *backtracking when* keyword argument achieves results similar to what the *cut* operator does in Prolog.)

So, the goal of this query is to produce the union of the leaves of Steve’s tree given that they are within a distance of 2 from him together with those non-leaf nodes that are at a distance of 2 from him.

The ancillary VBL designated by the argument to *with child nbr vbl* will contain the ordinal number (from 1) of the current node as a child of its parent in the depth-first search (foretime.8.Q).

By using a *with path vbl* ancillary variable, the complete path information from the root may be obtained as well (steve.9.Q):

```prolog
do Display
   each[ .descendant, .path ]
   each_time(
      .descendant Is_A_Descendant_Of "Steve"
      with_path_vbl path
      selecting_when( "M" Is_A_Substr_Of .path )
   );
```

The answer here is:

```
%msg1)
%msg2)Query File: ./Q/steve.9.Q
%msg3)
%msg4)
%msg5)flds)Descendant|Path
%msg6)
Mindy|<Steve>Ä<Ray>Ä<Paula>Ä<Mindy>
Max|<Steve>Ä<Ray>Ä<Paula>Ä<Max>
Martin|<Steve>Ä<Ray>Ä<Penny>Ä<Martin>
Milton|<Steve>Ä<Ray>Ä<Penny>Ä<Milton>
Maybelle|<Steve>Ä<Ray>Ä<Pandora>Ä<Maybelle>
```

Daytona assumes that any graph given to it to explore may have cycles and therefore it writes special code to test for the existence of such. This overhead can be removed at the cost of the user assuming the risk of an infinite loop if the keyword *given acyclic* is used in the path PRED definition. The loop detection code is simple: Daytona backtracks if the current node candidate for the path recursion box already appears in the path down from the root. This is often adequate but certainly not foolproof: one way to fool it is to define temporary path predicate variables that transmit constantly
recomputed information about the underlying graph down along the search. In that situation, the path PRED nodes that are encountered in the search may all be unique, even though the underlying graph nodes may be visited an infinite number of times. This is illustrated in (backpath.1.Q):

```plaintext
define path PRED: Is_After
[ TUPLE[ .x, .cum_age ], TUPLE[ .y, .prev_cum_age ] ]
by_stepping_with(
    there_is_a PERSONNE named .y where(
        one_of_the Children = .x
        and the Age = .age )
        and .cum_age = .prev_cum_age + .age
)
using(
    with_path_vbl path
    backtracking_when( .path / .x >= 2 )
)

do Write_Line( "here in backpath.1.Q" );
for_each_time [ .person, .cum_age ] Is_After [ "Steve", 0 ]
do {
    do Write_Words( .person, .cum_age )
}
```

Here PERSONNE has been created in such a way that there are several cycles. Since .cum_age will grow without bound as the search continues, the [ .x, .cum_age ] tuples stored in the path recursion box will all be unique and the infinite loop detection algorithm will fail. In order to prevent an infinite loop, backtracking is specified if the number of occurrences of .x in .path is greater than 2.

In some graphs, a number of paths will converge to a single "concentrator" node which will then be the source for a number of continuing paths. If the goal is just to visit all of the nodes of the graph, then it is pointless to visit the graph past the concentrator node more than once. This kind of control is provided by the Candidate_Selected_Before predicate which is _true_ if and only if the current candidate for inclusion in the path recursion box is already there. Here is an example of its use (trans.8.Q):
with_format _table_
do Display
  each .relative
  each_time(
    .relative Is_Reachable_From "Steve"
  )
);

define path PRED: Is_Reachable_From[ .x, .y ]
  by_stepping_with(
    there_is_a PERSONNE named .y where( one_of_the Children = .x )
  )
using(
  with_identity
  in_lexico_order
  backtracking_when( Candidate_Selected_Before )
)

So, in this query, when the current candidate has already been selected for inclusion in the path recursion box, it will be rejected for inclusion this time and Daytona will continue on to the next candidate. Candidate_Selected_Before can only be used for boxes that have a sort defined on them, which, of course, occurs automatically if with_no_duplicates is specified. See also trans.4.Q.

Graphs that have an unexpectedly huge number of children for a given node present another peril to deal with: it can be prohibitively expensive just to compute all the children, let alone to follow all the paths that lead down from them. The stop_finding_children_when keyword argument enables these situations to be identified and dispensed with before the entire mass of children have been computed. Here's an example taken from trans.7.Q:

define path PRED [ TUPLE[ .x, .parents_age ], TUPLE[ .y, .y_age ] ] :
  Is_A_Children_Constrained_Descendant_Of
  by_stepping_with(
    there_is_a PERSONI named .y where(
      one_of_the Children = .x
      and Age = .parents_age
    )
  )
using(
  with_outcount_vbl tot_kids
  stop_finding_children_when(
    if( .parents_age % 2 = 0 )
    then( .tot_kids >= 2 )
    else( .tot_kids >= 1 )
  )
)

As the children of a node are being computed, as soon as the assertion is satisfied, i.e., when .tot_kids reaches its appropriate threshold value of 1 or 2, the computation of the children ceases;
note that the child that first caused the assertion to be true is included in the children that will subsequently be worked with.

So, then, with all of these features, Cymbal offers considerable capability in controlling the search of computed graphs.
17. Updates, Transactions, Logging And Recovery In Cymbal

Transactions allow users to read and change their data in a way which is not only independent of the actions of other users but also independent of such disruptions to the computing process as interrupts and system crashes. There are a number of different definitions for transactions but for Daytona, a transaction is a user interaction with the database that satisfies two conditions:

1. If the transaction specifies changes to the database, then, no matter what happens in the computing environment, the system guarantees that either all of the changes are done or none of them are. In particular, this atomicity must be preserved in the face of user-specified transaction abort, integrity constraint violation, process-caught signal, system crash, and (optionally) media failure.

2. The influence of other users on a given user’s transaction is moderated in such a way that the given user can assume that while their transaction is running on the machine, no other users are present. In other words, regardless of what changes to the database actually occur at what time, the final database state is achievable by either having any other user’s changes occur completely before the given user’s transaction logic or else completely after.

Durability of the data in the face of system crash and the other troubles listed above can be obtained by using transaction logging whereby a queue of a transaction’s data changes is created in-memory and eventually written to disk prior to those changes being applied to the data. With logging turned on, data that may have been left half-changed by a transaction interrupted by a system crash can be restored using the Recover utility to properly apply the logged data changes to the data.

This chapter begins by showing how the basic delete, insert, and modify database operations are expressed in Cymbal. (In the literature, this class of operations is collectively called that of updates about as often as ‘modify’ operations alone are called ‘updates’; ‘update’ will be employed here using both of its meanings as well.) In addition, the non-transaction-based Cymbal blind append facility is discussed. Some transaction examples follow and then an in-depth discussion of Cymbal transaction semantics and usage. The chapter concludes with a section describing the use of logging and recovery.

17.1 Basic Transaction Syntax

Cymbal transactions are simply ordinary function, predicate, or procedure (fpp) tasks that have been explicitly designated to be transactions. Here is an example:

```markdown
define PROC transaction task: Delete_A_Part( with_nbr INT , part_nbr )
{
    do Change so_that( there_is_no PART where( Number = .part_nbr ) )
}
```

Tasks can have several keyword + argument modifiers placed immediately before the task keyword in the fpp definition. In this instance, the keyword transaction, which does not have an argument, is used;
txn can be used instead as a short synonym for transaction.

Also illustrated is the Change procedure call, which is being employed here to delete a record. All Cymbal deletes, adds, and modifies of database records are handled by Change call variants. In other words, whenever Cymbal users wish to change their data in any way, they issue Change calls. Each Change call takes as its sole argument an assertion keyword argument with keyword so that that asserts what the user requires to be true after the database has been changed. Regarding the transaction above, the requirement is that the database is to be changed so that there is no PART where the Number is equal to the part number passed in as an argument.

17.2 Using Cymbal To Delete Records

Here is a small program which, after asking the user to specify a Color and a Weight cutoff value, deletes all PART records which have the given Color and which have a Weight greater than the cutoff value (part_IQD).

locals: STR: .color ; FLT: .wt_cutoff

skipping 1 do Write( "enter color: " );
set [ .color ] = read( upto "\n" but_if_absent[ "blue" ] );

skipping 1 do Write( "enter weight cutoff: " );
set [ .wt_cutoff ] = read( upto "\n" but_if_absent[ 1.0 ] );

when( At_Eoc ) do Exit( 1 );
do Delete_Color_By_Weight( .color, .wt_cutoff );
global_defs:
define PROCEDURE transaction task :
  Delete_Color_By_Weight( STR .color, FLT .wt_cutoff )
  {
    do Change so that(
      there_is_no PART_ where(
        Color = .color
        and Weight > .wt_cutoff )
    );
  }

To specify a Cymbal deletion, simply claim within the so that assertion of a Change call that there is no record of a given RECORD_CLASS that meets certain conditions. As with all Change calls, Daytona processes deletions by changing the database so that the so that assertion is true, i.e., so that each record, if any, of the RECORD_CLASS that meets the there is a description for the there is no is deleted by:

1. Overwriting the first byte of the record with the ^ record-delete character.

2. Removing all references to the record from the associated indices.
3. Adding the space previously occupied by the record to the free tree so that subsequent additions and updates may reuse it, if possible. (This only occurs if the free tree exists, which it will not if <Reuse_Freed_Space, no> is annotating the associated FILE node in the rcd.)

Notice that the delete above could be handled using SQL embedded in Cymbal (part_3.IQD):

```csharp
locals: STR: .color; FLT: .wt_cutoff;
skipping 1 do Write( "enter color: " );
set [ .color ] = read( upto "\n" but_if_absent[ "blue" ] );
skipping 1 do Write( "enter weight cutoff: " );
set [ .wt_cutoff ] = read( upto "\n" but_if_absent[ 1.0 ] );
when( At_Eoc ) do Exit( 1 );
delete from PART_ where Color = .color and Weight > .wt_cutoff ;
```

Since this is more terse, it is to be preferred. However, the point with all Cymbal transactions is that the transaction body can be an arbitrary program. In this case, the program just specifies a simple delete; in general, arbitrary computation and other data modifications could make up the transaction body as will be seen subsequently. This cannot be done in a pure SQL setting because SQL does not contain a programming language.

By using the `with_sudden_death` keyword, only the first record that matches the conditions will be deleted.

```csharp
with_sudden_death do Change
so_that( there_is_no PART where( Color = "red" ) );
```

### 17.3 Using Cymbal To Add Records

#### 17.3.1 Using Cymbal To Add Records Using Transactions

The `there_is_a` variant of a Change call serves to ensure the existence of records having specified values (addsupp.1.IQA).
locals: STR: .supplier

set [ .supplier ] = read( from _cmd_line_ );

do Exclaim_Words( "SUPPLIER", .supplier, "added with Number",
    nbr_for_new_supplier( with_name .supplier ) );

global_defs:
define INT(_short) FUNCTION transaction task:
    nbr_for_new_supplier( with_name STR .supplier )
{
    set .new_nbr =
        (INT(_short_)) max( over .nbr each_time( 
            there_is_a SUPPLIER where( Number = .nbr ) ) ) +1;
    do Change so_that(
        there_is_a SUPPLIER where( 
            Number = .new_nbr 
            and Name = .supplier 
            and Telephone Is _absent_ )
    );
    return( .new_nbr );
}

Recall that Exclaim_Words separates its arguments with spaces and sends the result as a line to stderr.

In general, a Cymbal add will add a new record to a RECORD_CLASS if there is not already one there that satisfies the so_that assertion of the Change call. Consequently, trying to add the same record more than once will not result in multiple appearances of the record in the data file. In this case, the above program is used to add a new entry to the SUPPLIER RECORD_CLASS which has a Number one greater than the largest one present. Such a record, of course, is guaranteed to be new.

There are some restrictions on the usage of there_is_a to effect adds (and this_is_a to effect updates). Each FIELD assertion must either be an equality assertion or an Is_absent_ assertion. Also, no FIELD can be mentioned more than once.

Clearly, in order to add a new record, some decision must be made as to what the value should be (if any) for each FIELD. Suppose a FIELD is not mentioned in the there_is_a. Suppose also that the FIELD is a non-hparti FIELD, i.e., is not a Partitioning Field for horizontal partitioning. If there is no Default_Value specified for that FIELD in the rcd, then no value is written out for that FIELD in the data record. However, if a Default_Value is specified, then the Default_Value will literally be written out if missing values are not allowed; otherwise, nothing will actually be written into the data record for that FIELD, but of course, Daytona will subsequently treat that missing value as an instance of the Default_Value. If the user says to write a FIELD value out to the database that happens to be the Default_Value in the case where missing values are allowed, then nothing will be written out to that FIELD position in the data file record: when the delimiter is a |, this will result in the FIELD being represented by ||.

hparti FIELDS must have Default_Values specified in the rcd if any adds or updates are to be done
against the associated RECORD_CLASS without explicitly specifying values for the hparti FIELDS.

One implication of all this is that the Telephone Is absent assertion above is strictly speaking unnecessary. Also, since the City FIELD has a Default_Value of Murray Hill, that is the value that subsequent queries will see (although that value will not be written out to the corresponding FIELD position in the data file record).

Regarding the implementation of Cymbal adds, the system looks for the smallest free record space on the free list that can hold the new record and then uses that, if it exists, as the slot for holding the new record. Any unused bytes after the end of the new record are filled with comment characters. If no free space can be used, then the new record is appended to the end of the data file.

17.3.2 Handling Bad Records In Cymbal Transactions

Of course, new records are checked for their conformance to the rcd. FIELDS which must have values cannot be said to be absent. Any Validation_RE associated with a FIELD is enforced as are any Min_Value/Max_Value constraints. A check is made verifying that all Unique keys are unique. The new values given for FIELDS are typechecked according to the typechecking done for assignments, i.e., the FIELD is treated like a VBL that is getting a new value by means of assignment and that assignment is typechecked accordingly. This is what requires the INT(_short_) conversion above since the Number FIELD is an INT(_short_).

By default, any integrity error discovered for a record at runtime results in Daytona printing an error message and aborting the transaction. However, the user may wish to take over the error handling themselves as much as possible. Here are the keywords that facilitate this and an example of their use:
PROC: Change( so_that ASN,
    ( 0->1 ) with_sudden_death,
    ( 0->1 ) with_val_err_msg_vbl alias STR(*),
    ( 0->1 ) exiting_on_val_err,
    ( 0->1 ) continuing_on_val_err,
    ( 0->1 ) ensuring_string_sanity,
    ( 0->1 ) ensuring_string_sanity_nonascii_ok,
    ( 0->1 ) not_ensuring_string_sanity )

    { local: STR .emsg
        continuing_on_val_err
        with_val_err_msg_vbl  emsg
        ensuring_string_sanity
        do Change so_that(
            there_isa PART where( Number = .pnbr and Name = .pname) );
        when( .emsg != "" ){
            do Exclaim_Words(
                "Alert: integrity failure caused failure to add record for Part .pname"ISTR);
            do Exclaim_Words(" and the error message is", .emsg);
        } }

The optional keyword ensuring_string_sanity causes Daytona to check the values of STRING, LITERAL, RE, and THING FIELDS to make sure that they don’t contain characters that would invalidate the DC format for the record. Specifically, this check ensures that those field values do not contain the field separator(s), the TUPLE delimiters, unescaped or improperly escaped newlines, ‘\001’ (ctrl-A) (for implementation reasons), bytes with the 8th bit set, the DEL character, or a record-delete character as the first character of the first field value. By default, in order to increase query speed, Daytona does not perform these checks. The optional keyword ensuring_string_sanity_nonascii_ok has the same effect as ensuring_string_sanity with the exception that it allows bytes with the 8th bit set: this allows text containing foreign language characters that are coded by bytes with the 8th bit set.

The optional keyword ensuring_float_sanity causes Daytona to check the values of FLOAT FIELDS to make sure that they contain valid and usable floating point numbers that look like normal floating point numbers. In that regard, the IEEE floating point standard supports a variety of special binary codes that represent numbers that are not numbers. Such codes are called NaNs, for "not a number". Some misuses of floating point arithmetic like division-by-zero will result in a runtime signal that Daytona will catch but others will quietly create one of these NaNs. Here is one according to one compiler: nan.0 . The use of this option will prevent NaNs from being written out to DC data files by do Change. Of course, a careful, disciplined use of floating point arithmetic will also avoid the creation of NaNs in the first place -- and in all scenarios, not just do Change.

The only errors that can be handled using the val_err keywords are the string and float field value sanity errors, Validation_RE errors, Min_Value/Max_Value errors, trying to put a missing value into a
FIELD which cannot have missing values, and trying to use horizontal partitioning field values that do not correspond to a bin as defined in the rcd. Notice that these all correspond to field validation errors.

When a field validation error has occurred, Daytona by default will cause an exit from the program (exiting_on_val_err). However, if continuing_on_val_err is used, then, in the event of an integrity error, Daytona will continue execution with the statement following the do Change statement.

When the user provides an alias variable like emsg for keyword with_val_err_msg_vbl, then instead of Daytona printing an error message to stderr (as it does by default), it will make the message it would have printed available as the value of the user-provided alias variable. An error has occurred for a particular Change invocation if and only if the value of the alias variable is a non-empty string.

These same Change keywords are also valid when updating records.

17.3.3 Using Cymbal To Unconditionally Add Records Using Transactions

there_is_a_new can be used instead of there_is_a in a Cymbal add if the user wishes for the new record to be added unconditionally regardless of whether there is already one out there that meets the description and regardless of whether any unique key constraints might be violated. there_is_a_new (or equivalently there_is_a_new) is faster since it does not do either of the two previously mentioned checks. This is a feature that can be used to gain maximum speed when there is some guarantee in the application that only new, unique records will be presented for addition.

17.3.4 Using Sizup To Append New Records Using Batch Adds

The above Cymbal-based adds are considered to be query-driven which distinguishes them from the batch adds feature of Sizup. In the latter case, Sizup, which in no way uses Cymbal transactions, appends a sequence of new records to a data file in such a way that only the index entries for the new records are added and such that key uniqueness constraints are checked. Sizup running in batch-adds mode adds records to a table significantly faster than a Cymbal transaction would. The batch-adds feature is preferable when wishing to append a sizable amount of data already in the DC format to an existing DC table.

17.3.5 Using Cymbal To Blindly Append Records Outside Of Transactions

Fortunately, Cymbal provides the blind append form of the do Change call that can also be used to generate data file records ready for Sizup to load into a Daytona table, either by running in regular mode or by means of the batch-adds feature. There are many advantages to using blind appends over writing straight Cymbal logic that would depend on Write calls to produce data file records; for example, blind appends automatically:

1. convert field values to the appropriate data file format,
2. perform record-level compression, if specified in the rcd,
3. output the field values in the correct rcd-specified order regardless of their order in the query,
4. perform the above mentioned field validation checks and ensure that the types used are consistent with what is specified in the rcd,

5. insert default and/or missing values for FIELDS not mentioned,

6. apply any Output_Filter_Funs as specified in the rcd

7. handle horizontal partitioning by putting records in the specified BINs, and if needed, according to hparti FIELD Default Values.

In contrast to transactional Changes, blind appends perform no index work whatsoever and perforce incur none of the overhead associated with transactions such as the construction of a Do_Queue and logging. These adds are called blind appends because they append records to a file without the help or use of any indices (which includes siz files and free trees). Here is an example of a blind append (blindapnd.3.Q):

```plaintext
with_val_err_msg_vbl  emsg
continuing_on_val_err
do Change so_that(
    there_is_a_new SUPPLIED
    using_no_index
    using_source .source
    with_lock_mode _exclusive_
    where( Number = 301
        and Name = "Acme"
        and Telephone = "222-333-4444"
    )
);}
```

Daytona considers a blind append to have been specified when a there_is_a_new using_no_index is an argument to a do Change call that is located lexically outside of any transaction.

Blind appends will typically but not necessarily use a using_source directive since if it is omitted, then the new records generated will go onto the end of the data file as located by the FILE information in the rcd (which could be sensitive to shell expansions). On the other hand, the former strategy may be attractive when a large amount of records are being added because the user will want to run Sizup on the entire file anyway just to get nicely compacted, efficient indices. Other users may use the base data files during this time without interference from the blind append activity. Incidentally, blind appends will create a data file if it doesn’t already exist. Blind appends open their files in UNIX append mode.

There are three possible lock modes for blind appends: _no_lock_, _share_, and _exclusive_ with _no_lock_ being the default. Under _no_lock_, several different processes can be blind appending to the same file simultaneously even though no file locks have been obtained: this is guaranteed by the nature of UNIX file appending and by some careful programming on Daytona’s part to make sure that only whole records are flushed out of buffers. When _share_ is used, then a share file lock is obtained which will prevent any other process needing an exclusive lock from running
concurrently on the same file; such processes would include Sizup and processes running Cymbal

table-modifying transactions. In the case of horizontal partitioning when using the _share_ mode,

Daytona will not resort to its usual lock file acquisition policy when it is forced to close its access to a
bin in order to move on to another bin: blind appends have no notion of transaction including no

concern about serialization; in _share_ mode, they just want to concurrently append their records in

such a way that other blind-appenders don’t time out waiting for a lock file another one has and true

Cymbal transaction updaters are kept from modifying any file that a blind-appender has open for use.

An exclusive file lock is obtained when _exclusive_ is used.

17.4 Using Cymbal To Modify Records

The this_is_a variant of a Change call enables users to update the FIELDS of records satisfying
certain conditions. Here is a Cymbal request for increasing the ORDERS of every blue PART by 10%
(blubump10.IQU):

```cymbal
when( Orders_Of_Blue_Parts_Not_Increased )
do Exclaim_Line( "no blue part orders found" );

global_defs:
def PRED transaction task: Orders_Of_Blue_Parts_Increased[]
{
    set .change_made = _false_;
    for_each_time .qty
        is_such_that(
            there_is_a PART where(
                Number = .part_nbr
                and Color = "blue" )
            and there_is_an ORDER where(
                Part_Nbr = .part_nbr
                and Quantity = .qty
            )
        )
    {
        set .change_made = _true_;           
        do Change so_that(
            this_is_an ORDER where(
                Quantity = (INT(_short_))(1.1 * .qty) )
        )
    }
    return( .change_made );
}
```

A feature of Cymbal illustrated by this query is the support for introducing Not (and Does_Not) in a

PREDICATE name and having the system rewrite it as the PREDICATE preceded by the appropriate

negation. Observe the explicit casting necessary to force a FLT to the INT(_short_) value that is stored

in the FIELD Quantity.
It is by using *this_is_a* in a Change assertion that Cymbal users are able to specify that particular records are to be updated. Regarding implementation, Daytona updates a record in place in the data file whenever possible. So, if the new version of a record has a length less than or equal to the length of the original (including any subsequent comment characters), then the original record is overwritten with the new, using comment characters to fill out any discrepancy. However, if the new record is bigger than the old record’s place in the data file, then the old record is deleted and the new record is added in the same way that any Cymbal add is processed. Consequently, updating-in-place can be promoted by initially loading the data in such a way that each record is followed by a convenient number of comment characters. As described in Chapter 3, the *Pad_To_Len rcd* specification is used to cause the system to maintain a minimum (padded) length for records.

Here is a request which modifies the first ORDER (in order of appearance in the file) for SUPPLIER 411 so that it is resubmitted as of today (this_isa.IQU):

```
do Resubmit_First_Order( with_supp_nbr 411 );
global_defs:
define PROCEDURE transaction task :
    Resubmit_First_Order( with_supp_nbr INT .supp_nbr )
    {
        for_each_time [ .ord_nbr ] Is_The_First_Where(
            there_is_an ORDER where(
                Supp_Nbr = .supp_nbr
                and Number = .ord_nbr )
        ) do {
            do Change so_that(
                this_is_an ORDER where(
                    Date_Recd Is_absent_
                    and Date_Placed = today() )
            );
        }
    }
```

*Is_The_Last_Where* could have been used here instead of *Is_The_First_Where* in order to change the last order encountered. It would have been illegal however to have used *Is_Something_Where* or *Is_The_Next_Where* because both of these involve building boxes which typically contain more than one TUPLE: due to the way Daytona processes boxes, this would result in incorrect operations as detailed in the following argument. As each intensional box assertion is encountered during the processing of a *for_each_time* assertion, the box is constructed in its entirety at the first encounter and then each of its elements are made available in turn for each step in the incremental-satisfaction/backtracking logic process of generating all possible satisfactions of the entire *for_each_time* assertion. Consequently, in the example above, if *Is_Something_Where* were used, for any given assertion satisfaction, when the Change procedure in the *for_each_time* body is invoked, the record cursor for ORDER is positioned on the last ORDER record yielding an entry for the box and is therefore not properly positioned for the current satisfaction unless it is the last one. There is no problem of course if there is only one TUPLE in the box, as occurs with *Is_The_First_Where* and *Is_The_Last_Where.*
Users who feel inclined to want to use *Is_Something_Where* or *Is_The_Next_Where* may find that they can get what they want by not using a box at all. This is illustrated in the following variant of the above request which will resubmit all of a given SUPPLIER’s ORDERS (this_isa2.IQU):

```c
    do Resubmit_All_Orders( with_supp_nbr 417 );

    global_defs:
    define PROCEDURE transaction task :
        Resubmit_All_Orders( with_supp_nbr INT .supp_nbr )
        {
            for_each_time
                there_is_an ORDER where(
                    Supp_Nbr = .supp_nbr
                    and Number = .ord_nbr )
            do {
                do Change so_that(
                    this_is_an ORDER where(
                        Date_Recd Is _absent_
                        and Date_Placed = today() )
                    );
            }
        }

    Notice how the *for_each_time* above does not specify any *for_each_time* variables.

    The following transaction definition shows how *when* can be used to specify an update to some record, probably one known in advance to be unique, else there would be ambiguity as to what record would be modified.
global_defs:
define PROCEDURE transaction task :
    Resubmit_All_Orders( with_ord_nbr INT(_long_) .ord_nbr )
{
/** this stops at just one order **/
    when(
        there_is_an ORDER where(
            Number = .ord_nbr )
    ) do {
        do Change so_that(
            this_is_an ORDER where(
                Date_Recd Is _absent_ 
                and Date_Placed = today() )
        );
    } else {
        do Change so_that(
            there_is_an ORDER where(
                Number = .ord_nbr 
                and Supp_Nbr = 100 
                and Part_Nbr = 400 
                and Date_Recd Is _absent_ 
                and Quantity = 0 
                and Date_Placed = today() )
        );
    }
}

17.5 Adding And Updating LIST- or SET-Valued FIELDS

FIELDS can have LISTS and SETS of scalars as values. These BOXES can be further specified, if desired, by the use of the keywords with no duplicates and one of with_lexico_order and with_reverse_lexico_order. The updating process will cause the specified order to appear in the data file record.

In order to add a record with LIST/SET values for given FIELDS, just proceed as before and provide such values using any available syntax:
do Change so_that(
    there_is_a MANIFOLD where(
        Number = 5
        and Counts = .my_list
        and Birthdays = { .y, .z, `3-4-89`, `5-3-98` }
        and Sizes = [ .x : .x Is_In [ 101.5 -> 103.0 by .5 ] ]
        and Timestamps = [ `1-1-89@2` : : with_lexico_order ]
    ));

Note that the values given for the FIELDS must have types that are exactly the same as those of the FIELDS, hence the use of with_lexico_order. (cf., multival.B.IQA).

The paradigm for updating LIST/SET-valued FIELDS is a simple one: first, get the value of the FIELD into a BOX of the same type, second, modify the box in whatever way desired, and third, put it back in the FIELD in the RECORD. Here is a transaction that sums up the INT elements of a LIST-valued FIELD and then appends that sum to the end of the LIST (multival.A.IQU).

define PROC txn task: Add_A_Count
{
    for_each_time [ .cnts ] is_such_that(
        there_is_a MANIFOLD where ( Counts = .cnts )
    )
    {
        set .new_cnt = sum( over .x each_time( .x Is_In .cnts ) );
        do Change so_that( .new_cnt Is_In_Again .cnts );
        do Change so_that( this_isa MANIFOLD where( Counts = .cnts ) );
    }
}

In general, the this_isa that does the updating can assert that the LIST/SET-valued FIELD is to get a LIST/SET-valued term, including constants and even empty LISTS like []. Once again, the new values given for the updated FIELDS must have types that are exactly the same as those of the FIELDS.

Here from multival.E.IQU is the essence of a transaction body that will change the 3rd element (if any) of the Flags LIST FIELD of the fourth RECORD in MANIFOLD2:

fet [ .flist, .eltvv ] ist(
    there_is_a MANIFOLD2 where(
        Number = 4
        and Flags = .flist
    )
    and ? Is_In .flist with_elt_vbl_vbl eltvv with_selection_index 3
)
{
    set ..eltvv#1 = `1111111`B;
    do Change so_that( this_isa MANIFOLD2 where( Flags = .flist ) );
}
17.6 Adding And Updating TUPLE Valued FIELDS

When updating TUPLE-valued FIELDS, feel free to work with them as a unit, i.e., as TUPLES (dynord_hparti_bin.1.IQU):

define PROC txn task:
  Change_Dynord_Hparti_Bin
  { local: TUPLE[ OBJ, OBJ, OBJ ] .new_hparti_key
// Updates
  fet [ .hparti_key ]
  ist(
    there_isa DYNORD_HPARTI_BIN where(
      File = "ORDER.11" and
      Hparti_Key = .hparti_key
    )
  ){ set .new_hparti_key = [ 6, .hparti_key#2, .hparti_key#3 ] ;
  do Change so_that( this_isa DYNORD_HPARTI_BIN where(
    Hparti_Key = .new_hparti_key
  ));
  }
  fet [ .hparti_key ]
  ist(
    there_isa DYNORD_HPARTI_BIN where(
      File = "ORDER.18" and
      Hparti_Key = .hparti_key
    )
  ){ do Change so_that( this_isa DYNORD_HPARTI_BIN where(
    Hparti_Key = [ 8, .hparti_key#2, .hparti_key#3 ] )
  );
  }
  fet [ .hparti_key ]
  ist(
    there_isa DYNORD_HPARTI_BIN where(
      Hparti_Key = .hparti_key where( .hparti_key#3 = 4 )
    )
  ){ do Change so_that( this_isa DYNORD_HPARTI_BIN where(
    Hparti_Key = [ 8*.hparti_key#1, 4*.hparti_key#2, 444 ] )
  );
  }
// Delete using TUPLE constraint
do Change so_that( there_is_no DYNORD_HPARTI_BIN where(
  Hparti_Key = .hp_key where( .hp_key#2 = "A" )
));
do Change so_that( there_is_no DYNORD_HPARTI_BIN where(
  Hparti_Key#2 = "G"
));
// Add
do Change so_that( there_is_a DYNORD_HPARTI_BIN where(

Number = 2 and
Fif_File = "$\{ORDERS_DATA:~\$ORDERS_HOME}/dynord_flis.2" and
File = "ORDER.22" and
Source = "$\{ORDERS_DATA\}" and
Indices_Source = "" and
Hparti_Key = [ 11, "D", 2 ]
);

Note in particular the use of this construction where a component of a TUPLE FIELD is taken: Hparti_Key#2. Upon examination, rcd.DYNORD_HPARTI_BIN (and its brethren) will show that the Source FIELD is defined to be SAFE_STR. This is to support syntactically unconstrained pipe Sources.

17.7 Batching Updates To Enhance Performance

Daytona’s per transaction overhead is sufficiently large that it can be substantially inefficient to write transactions that update just a single record per invocation. Instead, it is much better to update thousands of records per invocation if possible. In one real-world example where the batch update strategy updated 10000 records per invocation, the batch update strategy was 18.5 times faster than the single update strategy. Part of the speedup has to do with Daytona optimizing access to the disk by grouping together the work for similarly located records.

Here is some actual user Cymbal code that shows how easy it is to batch updates by using a box. This code is reading its work list, one record at a time from .in_chan. When 10000 quadruples have accumulated in .to_do_box, the transaction PROC Make_Changes is called to do the associated updates.
export: LIST[ TUPLE[ STR, STR, STR, INT ] ] .to_do_box
// define .in_chan here
loop {
    set .to_do_box = []; set .cnt = 0;
    for_each_time .idx Is_In [ 1 -> 10000 ]
    {
        set [ .wtn, .btn, .name, .month ] = 
        read( from .in_chan upto "\n" but_if_absent ["0","0","NA",9001])
        otherwise_switch {
            case( = _instant_eoc_ ){ do Exclaim_Line( "At_Eoc" ); break; }
            case( = _missing_value_ ){ with_msg "surprise missing value" do Exit( 2 ); }
            else { do Exit( 3 ); }
        }
        set .cnt++;
        when( .cnt % 1000 = 0 )
            do Exclaim_Words( ".cnt=" , .cnt, "at", date_clock_now());
            do Change so_that( [ .wtn, .btn, .name, .month ] Is_In_Again .to_do_box );
    }
    do Make_Changes;
    when( At_Eoc ) break;
}

Of course, the PROC Make_Changes contains within it a for_each_time loop that iterates over the elements of .to_do_box and performs the appropriate do Changes for each work list record. Important: instead of using a BOX to hold the tuples to process, it would be more efficient to use a TUPLE-valued conventional ARRAY.

17.7.1 Batching Sequential Updates To Enhance Performance

As a special case of batching updates, consider the situation where it is desired to update so many records in a table that one or more of the following concerns arise:

1. accessing each updatable record by an index will be tremendously inefficient because there are so many records and indexed retrieval at this scale will swamp the disks with I/O requests, thus leading to unacceptably long elapsed times with correspondingly miniscule total CPU (=user+sys) times.

2. whereas sequential access is indicated, there are so many records to update that if they were all updated in one transaction, the resulting Do_Queue would exhaust memory.

3. a single sequential transaction updating in stages would appear to be tedious to write because of the need to remember where to resume the sequential scan after each stage.

4. it is desired to let other perhaps read-only transactions periodically access the table while this massive update is being undertaken.
Fortunately, there is a single simple solution to all these concerns. As illustrated first in massequp.1.IQU, it revolves around the fact that subsequent visits to \texttt{there\_is\_a\_next} will resume the sequential search where the last visit ended. The goal of this query is to increase the Quantity field value of every ORDER by 15% but to do so in increments of 22 records per transaction invocation.

\begin{verbatim}
while( bump\_next\_batch\_of\_orders() = 22 ){ }

global\_defs:
define INT FUN txn task: bump\_next\_batch\_of\_orders
{ local: INT .cnt = 0;

  for_each_time [ .qty ]
    is\_such\_that( there\_is\_a\_next ORDER where( Quantity = .qty ) ){
      set .cnt++;
      do Change so\_that(
        this\_is\_an ORDER where( Quantity = (INT(_short_))(.qty * 1.15) ) );
      when( .cnt = 22 ) break;
    } else {
      set .cnt = 0;
    }

  return( .cnt );
}
\end{verbatim}

Unfortunately, there is a serious problem with this query. The fact is that some of the updated ORDER records will be longer than the originals and thus the original record location will be put on the free tree and the new record will be located elsewhere either in some other reclaimed space or at the end of the file. If the new record should be located in advance of the record cursor maintained by \texttt{there\_is\_a\_next}, then that record will be visited again and unless either the update is idempotent (i.e., applying it twice is identical to applying it once (massequp.2.IQU)) or else there is some test applied whereby a previously updated record can be identified and left alone, then the update will unfortunately happen again.

In this case where the goal is simply to increase the Quantity by 15% once for each ORDER, there is a simple solution: use \texttt{using\_reverse\_siz} with a \texttt{there\_is\_a\_next} (massequp.5.IQU). This will be effective because the ORDER record class has an empty free list at the start of this query and so, any updated records that are moved will either go behind the cursor in previously opened slots or else they will go at the end of the file, which amounts to the same thing, i.e., all recycling activity goes on behind the advancing cursor. This can also be guaranteed to be the case if \texttt{<Reuse\_Freed\_Space, no>} holds for all FILES in the record class. Also, please note that a RECORD\_CLASS can be made amenable to this strategy of updating by processing it with a Sizup\_packing to squeeze out all freed space.

17.8 Two Sample Transactions

The examples presented in this section make use of the procedural part of Cymbal to accomplish several modifications in the same transaction.
In the first example, every PART_ that weighs more than 5.0 units is updated so that its Weight is increased by 50% and, in addition, in order to compensate, every ORDER involving that PART_ has its Quantity decreased by 50% so that total weight ordered remains constant (wtadjust.1.IQU). (Of course, this does not make a lot of sense in the real world but it does yield a pedagogically useful example for the purposes of this manual.)

do Adjust_Weights_And_Quantities;

global_defs:
define PROCEDURE transaction task: Adjust_Weights_And_Quantities() {
    set .part_cnt = 0;
    for_each_time [ .part_nbr, .wt ] is_such_that(
        there_is_a PART_ where(
            Number = .part_nbr
            and Weight = .wt which_is > 5.0 )
    ) do {
        set .part_cnt++;
        do Change so_that( this_is_a PART_ where( Weight = 1.5 * .wt ) );
        for_each_time [ .qty ] is_such_that(
            there_is_an ORDER where(
                Part_Nbr = .part_nbr
                and Quantity = .qty )
        ) do {
            do Change so_that( 
                this_is_an ORDER where( Quantity = (INT(_short_))( .5 *.qty ) )
            )
        }
        do Exclaim_Words( .part_cnt, "parts have been updated" );
    }

Note the nesting of the for_each_time loops and the interspersion of code to count up the number of parts updated.

For comparison purposes, consider how this would be done in SQL (wtadjust.1.ISU):
Certainly, the SQL is terser. It is also less efficient because it visits the PART table three times instead of the single time specified by the Cymbal query. This is typical: Cymbal tends to get users exactly what they want in the most efficient fashion. SQL tends to be pleasantly terse within the realm of the questions it can answer. In Daytona, both can be used as convenient.

The next query is considerably more complicated. Here the goal is to consolidate the Quantities of all unfilled ORDERS of a given PART from a given SUPPLIER into a single ORDER placed at the most recent date of any ORDER in that group (consord.1.IQU). In order to handle this request efficiently, the associated Cymbal begins by constructing a box containing the relevant ORDER information for only those SUPPLIER-PART combinations that have more than one unfilled order. This box will serve as an in-memory RECORD_CLASS that is accessed four separate times in this query. This will save time since the expectation is that most of the ORDER file can be excluded from subsequent processing because most SUPPLIER-PART combinations are associated with a single filled or unfilled ORDER. Once the ORDER to be updated and its new total Quantity are determined, an update transaction is called to both do the update and remove the other ORDER records in the group.
set .ord_box = [ [ .supp_nbr, .part_nbr, .ord_nbr, .date_placed, .qty ] :
    there_is_an ORDER where(
        Number = .ord_nbr
        and Supp_Nbr = .supp_nbr
        and Part_Nbr = .part_nbr
        and Date_Placed = .date_placed
        and Date_Recd Is _absent_
        and Quantity = .qty )
    and
    there_is_an ORDER where(
        Number != .ord_nbr
        and Supp_Nbr = .supp_nbr
        and Part_Nbr = .part_nbr
        and Date_Recd Is _absent_
    ) ];
for_each_time [ .ord_nbr, .total_qty ]
is_such_that(
    [ .supp_nbr, .part_nbr ] Is_Something_Where(
        [ .supp_nbr, .part_nbr, ?, ?, ? ] Is_In .ord_box )
    and .max_date =
        max( over .date each_time(
            [ .supp_nbr, .part_nbr, ?, .date, ? ] Is_In .ord_box ))
    and [ .supp_nbr, .part_nbr, .ord_nbr, .max_date, ? ] Is_In .ord_box
    and .total_qty = (INT)(
        sum( over .qty each_time(
            [ .supp_nbr, .part_nbr, ?, ?, .qty ] Is_In .ord_box )))
) do {
    do Consolidate_Orders( to_order .ord_nbr with_new_qty .total_qty );
}
global_defs:
define PROC transaction task:
Consolidate_Orders(
    to_order INT .ord_nbr, with_new_qty INT .qty ) {

    for_each_time [ .supp_nbr, .part_nbr ]
is_such_that(
        there_is_an ORDER where(
            Number = .ord_nbr
            and Supp_Nbr = .supp_nbr
            and Part_Nbr = .part_nbr )
    )
    do Change so_that(
        this_is_an ORDER where(}
Notice that an update has been coupled with a delete in the same Change call. In general, \texttt{so\_that} assertions can be conjunctions of \texttt{there\_is\_a}, \texttt{this\_is\_a}, and \texttt{there\_is\_no} assertions.

### 17.9 Final Synchronous Writes

As a compromise between running a transaction with or without logging, Daytona also offers the \texttt{with\_final\_fsync} transaction task option. With this option in effect, before the transaction task returns, all modified data will be written to disk. Were this option not to be used, then the user’s changes would be flushed to kernel space where at some unknown time in the future they would be written to disk. So, the use of \texttt{with\_final\_fsync} significantly narrows the exposure of the user’s updates to machine crashes. On the other hand, the use of logging reduces this window to zero. Furthermore, widespread use of \texttt{fsync}, since it usurps the OS’ policy for flushing to disk, will result in an inefficient use of the platform that will likely be visible to other processes. Here is an example of how to use this keyword (partedrnd2.IQU):

\begin{verbatim}
define PROC transaction with_final_fsync task :
    Update_Or_Add_Or_Delete_Parted( INT .nbr )
{ ... }
\end{verbatim}

### 17.10 Transactions In General

This section discusses the implementation of Cymbal transactions in general as well as a number of miscellaneous issues and features.

Daytona executes the body of a transaction by processing it the same way it would the body of any fpp with the exception that, instead of doing any of the indicated data changes at the time it encounters them, it constructs a queue of all of the basic reads and writes to data and indices that the transaction would like to make. After the body of the transaction has been processed in this way, the corresponding so-called Do Queue is executed so as to truly accomplish the changes specified in the transaction. The commit step of a Daytona transaction is the execution of this Do Queue.

Regarding storage efficiency, Daytona’s B-tree indices become increasingly inefficient at using their space if they are being constantly modified by transactional \texttt{do Changes}. To remedy bloated indices, just run Sizup on the associated data files. The improvement in space efficiency can be dramatic.

### 17.10.1 Logging

The durability part of the transaction paradigm is achieved by transaction logging. Transaction
logging occurs when the UNIX process that is running one or more transactions synchronously writes to its own disk log file(s) a record of those database changes it intends to make in order to fulfill each transaction before it actually does so. This record of database changes is just a reformatted, disk-oriented form of the in-memory Do Queue. This disk record of changes is called a REDO log because, when used, it makes changes to the database in a going-forward-in-time order and in the process of doing that, it may redo actions that had already been made by transactions earlier. This is OK because REDO logs have a special idempotent property which means that running them twice has the same effect as running them once.

Anyway, if the system crashes during the execution of the Do Queue, then when the system comes back again, Recover, the recovery manager, just redoes all of the relevant REDO logs in the appropriate order. REDO logs for successfully committed transactions are removed by Clean Logs, a daemon cleanup process. Of course, if the system crashes prior to beginning the commit stage, then no changes have been made and so there is nothing to REDO or UNDO: it is as if the transaction had never run. Also, if a signal from the system is caught by a transaction at any time during its run, then all portions of the Do Queue that have been executed (if any) will be undone during the abort of the transaction (without relying on log files, if any) and that portion of the REDO log, if any, will be expunged. Likewise, Do Queue-based UNDO activity may occur if an integrity constraint violation is detected during commit. (In practice, the most common integrity constraint violation encountered during commit is the detection of duplicate keys.)

There are several implications of this architecture. One is that Daytona can support recovery from media failure by rebuilding the database from archive tapes together with a cumulative log of all transactions done since the last backup. Clearly, in this case, it is important not to run Clean Logs after the last backup. It is also important for the log files not to be on the same disk as the data files! Applications greatly concerned about media failure can also use fault-tolerant computers with mirrored-disks. One advantage of running Clean Logs is that the database administrator (DBA) does not have to worry about the logistics of storing an ever-growing, indefinitely large amount of log data. Applications can speed up logging by writing logs to solid-state disks.

See the last section of this chapter for details on how to use logging.

17.10.2 Transaction Statistics Including Change Counts

Each transaction has a serial number which can be accessed immediately after it finishes by referring to the global quantity .txn_status.Serial_Nbr:

C_external STRUCT( UINT .Serial_Nbr,
                   UINT .Cur_T_Deletes,
                   UINT .Cur_T_Inserts,
                   UINT .Cur_T_Updates ) .txn_status

As indicated, the number of deletes, inserts, and updates for the last transaction alone are also available; these numbers are insensitive to the number of do Save calls, if any. These statistics count the number of times that the corresponding data modification code was executed: recall that Cymbal’s do Change calls only make those changes that are actually needed to make the Change assertion true. For example, if, when doing an update, a candidate record already satisfies the Change assertion, then
the update code will not be executed for that record and so .txn_status.Cur_T_Updates will not be incremented for that record. (See count.updates.1.IQU.) Likewise for deletes and adds "that have already happened". Thus these counts reflect changes that are actually made, i.e., modifying actions that actually happen. To highlight this reasoning, suppose one had ten new records to add to a record class with a corresponding transaction, containing, of course, a do Change, that ensures that those 10 records are in the record class. Suppose this exact same transaction was run twice in a row. The first time, .txn_status.Cur_T_Inserts would be 10; the second time, it would be 0.

17.10.3 Transactions Cannot See Their Own Updates Unless do Save Is Used

Another architectural implication is that, unless special steps are taken, transactions cannot see their own updates, meaning that transaction logic cannot change records and then, within that same transaction, act as if it knows what those changes are so as to return to the same records and change them again. For example, a transaction whose logic first changes a record and then later comes back and changes it again is implicitly assuming that it can see its own updates; in this case, the first update can be lost and what’s worse, unwanted records can be created/deleted in the data file and the indices can (erroneously!) contain entries for deleted records. Daytona does not prevent this! So, given the kind of misbehavior that can result, the user does need to be aware of when their transaction logic assumes an ability to see its own updates in the sense of expecting to be able to read or modify records that have been previously modified in that same transaction.

However, since the ability to detect this flaw in transaction logic is actually pretty subtle and therefore something of an art, Daytona offers the user the option of invoking Tracy with the special -COU flag, which can be taken to mean "check own updates" or "see own updates". Use of -COU will cause Daytona to perform additional checking at runtime by analyzing the nature of the Do_Queue during the course of an additional linear scan. The additional overhead is probably not noticeable. This analysis may produce fatal errors, or in some cases just warnings, that together alert the user to transactions that are either guaranteed to cause trouble or else are merely suspect.

Fortunately though, if the user really does want for the transaction to see its own updates at certain points in its execution, this can be arranged by the judicious placement of a do Save call. Cymbal users may issue a do Save call in a transaction body so as to cause the Changes processed so far to be committed to disk and thereby made visible to the transaction. Conversely, a do Rollback causes all Changes saved so far to be erased, i.e., undone. (And of course, any changes Saved are also undone if the transaction aborts for some reason.) Save and Rollback calls can be placed at any point except in the body of a for_each_time loop whose assertion refers to record classes. Placing such calls in the do-group of a for_first_time loop is permitted as is placing them in the else do-group of a for_each_time loop (rndordera.2.IQU). Rollback is done on a logical, not a physical basis: a Rollback of appended records will leave deleted records in the data file and corresponding entries in the free list.

17.10.3.1 Case Study Of A Malformed Transaction

Here is a badly formed transaction whose style should not be imitated! :
define PROCEDURE transaction task:
  Reset_Order( INT .ord_nbr )
{
  for_each_time [ .supp_nbr, .part_nbr, .qty ]
    is_such_that{
      there_is_an ORDER where(
        Number = .ord_nbr
        and Supp_Nbr = .supp_nbr
        and Part_Nbr = .part_nbr
        and Quantity = .qty
      )
      do {
        do Change so_that(
          there_is_no ORDER where(
            Number = .ord_nbr
          )
          and there_is_a ORDER where(
            Number = .ord_nbr
            and Supp_Nbr = .supp_nbr
            and Part_Nbr = .part_nbr
            and Quantity = 0
          )
        );
      }
    }
}

Since Number is a Unique KEY for ORDER, this transaction just makes no sense upon close inspection because, reading Cymbal as English, it is asking Daytona to change the ORDER table so that it is both the case that there is no ORDER record with a given Number and that there is such a record. Both cannot be true! Apparently the goal of the author of the query was to first delete the given ORDER record and then to add it back in with a Quantity of 0. But this very goal makes no sense when expressed as a conjunction assertion to a do Change call because Daytona considers the conjunction with the conjuncts reversed to be semantically the same as the one with the original order in so far as the ultimate effect on the database; in that case the author would have to be thinking instead that the "new" record would be added and then deleted, which is surely not what they wanted. This is an example of where the author is implicitly assuming that a transaction can see its own updates because the transaction is (apparently) assuming that it can delete an ORDER record with Number .ord_nbr before it adds such a record. (Recall that Cymbal record adds always check to see if the intended add already exists, and if so, then they do nothing.)

Of course, the author's apparent intent can be handled by a simple update by replacing the above Change call with this one (resetorder.IQU):
do Change so that(
    this_is_an ORDER where(
        Quantity = 0 and
        Date_Recd Is _absent_ and
        Date_Placed Is _absent_ and
        Last_Flag = 0 )
);

On the other hand, by using a do Save, the author’s original logic can be implemented as a delete followed by a Save followed by an add (resetorder.2.IQU):

define PROCEDURE transaction task:
    Reset_Order( INT .ord_nbr )
    { local: .prev_supp_nbr; .prev_part_nbr // here for scoping
        for_each_time [ .supp_nbr, .part_nbr, .qty ] is_such_that(
            there_is_an ORDER where(
                Number = .ord_nbr
                and Supp_Nbr = .supp_nbr
                and Part_Nbr = .part_nbr
                and Quantity = .qty
            )
        ) do {
            set [ .prev_supp_nbr, .prev_part_nbr ]
                = [ .supp_nbr, .part_nbr ];
            do Change so that(
                there_is_no ORDER where( Number = .ord_nbr )
            );
        }
    do Save;
    do Change so that(
        there_is_an ORDER where(
            Number = .ord_nbr
            and Supp_Nbr = .prev_supp_nbr
            and Part_Nbr = .prev_part_nbr
            and Quantity = 0
        ));
}

17.10.4 Handling Very Large Transactions

Daytona’s Do Queue/REDO log architecture also permits a user to have so-called very large transactions (VLTs). If logging has been turned on, then a Do Queue that exceeds memory limits or needs more memory than a certain user-settable maximum will be written to and executed from disk. In this way, Daytona supports effectively unlimited-size transactions for those users requiring such behemoths.
The query command-line keyword \texttt{+MDQ} can also be used to specify, as an integer, the maximum Do Queue size in kilobytes as in \texttt{+MDQ 409600}; this would override at runtime any Max Do Queue In K value given in a note in the PROJECT description, if any. A C extern variable \texttt{Max Do Queue In K} contains this maximum value. The default value is 100 megabytes. Even though the average transaction step stored in the queue does not vary too much in size, there is no effective way to estimate what Max Do Queue In K should be for an application because there is no effective way to estimate the number of transaction steps in any transaction, let alone over all the possible or probable transactions. So, Max Do Queue In K has to be determined by trial-and-error.

With regards to sizing Max Do Queue In K, recognizing that the Do Queue is an in-memory data structure, the objective becomes one of choosing a size that will accommodate all the Do Queues created in practice while ensuring that these further two objectives are met: 1) Daytona transactions do not take up so much physical memory that they interfere with the performance of other transactions and processes and 2) Daytona transactions do not create Do Queues that are so large in relation to physical memory that significant portions of them get paged out to swap disk, thus incurring a significant performance penalty. Note that transaction logging provides a way to handle transactions with Do Queues that are very large in relation to physical memory. (Since Daytona is a 32 bit application and since the Do Queue lives in the heap, it is clear that the Do Queue cannot exceed 4GB in size at the most but in practice somewhat less than that due to the space needed by other portions of the process image.)

The query command-line keyword \texttt{+IDQ} can also be used to specify, as an integer, the minimum (or starting) Do Queue size in kilobytes as in \texttt{+IDQ 409600}; this would override at runtime any Init Do Queue In K value given in a note in the PROJECT description, if any. A C extern variable \texttt{Init Do Queue In K} contains this maximum value. The default value is 8 kilobytes.

17.10.5 Locking Issues

At this time, Daytona employs locking at the file level to achieve concurrency. In a later release, it will support record locking as well. Within a transaction task body, if all of the accesses (i.e., \texttt{there is a's}) for a given RECORD CLASS are not associated with Change calls, then Share locks and only Share locks will be gotten on the file(s) corresponding to that RECORD CLASS; otherwise, each access associated with a Change call will result in the exclusive locking of the associated RECORD CLASS's files as it visits them. Greater concurrency can be achieved even with file locking in effect by using horizontal partitioning.

\textbf{Important Note:} any data reading specified in Cymbal that does not appear in the body of a \textit{transaction} task is not getting any locks. These so-called \textit{dirty reads} are simply taking their chances if there are any Cymbal/DSQL updating processes running concurrently or if Sizup is running concurrently. In general, all of the usual bad things can happen: the reading of uncommitted updates, the reading of indices out of sync with their data and even (in rare circumstances) reading of data structures that are not yet structurally sound because they are in transition, thus ending in aborts. Consequently, users who wish to avoid such risks may do so in a guaranteed way by enclosing their reads in a transaction task. \textbf{However}, as a special case, Daytona has been architected so as to support dirty reads without fear of program failure if the only updating that is going on consists exactly of appending records at the end of data files, either by Sizup or by transactions.
Caveat: the Important Note above must be read literally. In the following example, no (read) locks will be gotten on SUPPLIER:

```c
    do Doit;
    global_defs:
    define PROC txn task: Doit
    {
      _Show_Exp_To( doit() );
    }
    define INT FUN task: doit
    {
      when( there_isa SUPPLIER ) return( 1 );
      return( 0 );
    }
```

This is because the Cymbal code referring to SUPPLIER does not appear in the body of the transaction task. A read lock would have been acquired if the `doit` task were made a helper fpp of the `Doit` transaction task.

### 17.10.6 Handling Deadlock

On occasion, transactions may abort with a message:

```
    system runtime error = Deadlock situation detected/avoided
```

Here are some ways that transactions can avoid these deadlock aborts. The classical way is for all updaters and readers (using locking) to agree as a matter of application protocol to access the data files they do within any given transaction body according to some agreed upon order, for example, by alphabetical order on the RECORD_CLASS name. In other words, if a transaction only accesses SUPPLIER records before accessing ORDER records and all other transactions behave in the same way, then there can never be a deadlock due to accesses to these two tables. (Note that the policy here is not based on using alphabetical order by RECORD_CLASS name.)

Since Daytona locking takes place at the file level, deadlock can also occur when just accessing bins (files) within the same RECORD_CLASS. For example, if Max_Open_Random_Acc_Bins is 24 and an updater has all 24 open for doing random updates, then a reader transaction can well experience deadlock if it somehow manages to have open more than one file simultaneously. One way for that to happen is just to have a (flush_on_return) transaction that is only looking up records in one bin per invocation. However, over the invocations, if there is more than one bin that is searched, the flush_on_return nature of the transaction will keep open (and locked) the previously opened bins up to a maximum of 24. Thus deadlock is probable. There are two ways in addition to the classical method to avoid this. First, if the transaction was made close_on_return, then each invocation would open and close (losing the lock) exactly one bin -- no deadlock possible here. However, close_on_return transactions take longer to run than their flush_on_return counterparts. So, the second alternative solution begins with the reader keeping the original flush_on_return characteristic for efficiency’s sake (so that if the next invocation is to the same bin, it is already open and locked). Instead, the idea is to define in the rcd for, say, SUPPLIER:
and then running the reader under the environment where

\$SUPPLIER_MAX_OPEN_RANDOM_ACC_BINS = 1 .

This way, the updater gets the default of 24 and the reader gets by with 1 as the value of Max _Open _Random _Acc _Bins: this will cause the reader on each subsequent invocation to close the previous file (if different) (thus releasing the lock) and opening and locking, as needed, a new file. No deadlock can occur in this situation either. (It may be helpful to recall that the Cymbal function put _shell _env can modify an executable’s environment during execution.)

### 17.10.7 Updates To Horizontally Partitioned Tables

Updates for horizontally partitioned RECORD _CLASSES are specified in Cymbal and SQL just like they are for ordinary RECORD _CLASSES. The only exception is that hparti _FIELDS cannot be updated. If there are hparti _FIELD values that must be changed for some record, then it will be necessary to first delete the record from its original BIN and then to add it to its new one.

In order to preserve transaction semantics, if a transaction visits more than Max _Open _Bins for a given horizontally partitioned RECORD _CLASS, then lock files are obtained for the older, visited BINS and are held for the duration of the transaction; this is true even for read-only transactions.

Creating lock files for read-only transactions may be a nuisance but it is the only way to guarantee serializability for these transactions. When more than Max _Open _Bins are opened, lock files are employed in order to preserve serializability by keeping those closed bins locked. Since UNIX file locking can only be used when files are open, when old bins have to be closed in order to open new bins, lock files provide the technology to keep those closed bins locked. Unfortunately, the current lock file implementation can only get Exclusive locks. If the user is willing to tolerate lack of serializability or if they can prove that their particular usage of their database is serializable, then Daytona can be instructed to forgo creating lock files for read-only transactions by using the no _share _lock _files keyword in their transaction definition, as illustrated by:

```plaintext
define PROCEDURE txn no_share_lock_files task:
   Verify_Update_Parted( INT .nbr, STR(25) .color )
```

For those users willing to accept the possibility of even more dire non-serializability consequences, the no _lock _files keyword will prevent the associated Daytona transaction from creating any lock files, regardless of whether its table accesses are Share or Exclusive. The default behavior to use lock files as needed to maintain serializability can be specified by using the optional keyword lock _files _ok.

Incidentally, if a transaction or Sizup should die in a tragic way and leave a few lock files behind, there is no need to be concerned since the next transaction or Sizup run that is done that needs access to the associated data files will detect that the lock files are no longer associated with any living process and will consequently remove them.

### 17.10.8 Exception Handling For Transactions

FUNCTION and PREDICATE transaction tasks may specify a value to be returned on event of
transaction abort (as would occur if a signal was caught or if some data file could not be found). This is accomplished with the `on_abort_return` keyword argument to the declaration of the FUN/PRED as in:

```c
define INT FUN on_abort_return _last_signal_ transaction task:
```

- `_last_signal_` is a Cymbal object whose value is the signal number of the last signal caught or else is `_sigleave_` (which is 99 at the C level) indicating that the transaction aborted due to a problem that Daytona detected itself (that would typically otherwise cause Daytona to exit the program with status 101) or due to the user calling the Abort PROC at the Cymbal level.

In light of the above, `_sigleave_` is not considered to be a signal but rather an `abort_event`. Also, for the record, any SIGTSTP signals (also known as `^Z` and in Cymbal as `_tstop_`) sent during a transaction execution will be held (not caught) until the txn ends; on the other hand, SIGSTOP signals will be caught.

In this regard, the following sequence of actions occurs when handling a transaction abort. First, any database changes in the Do_Queue that had been done are undone. Second, the signal catching functions for the signals SIGQUIT, SIGINT, SIGTERM, SIGPIPE, and SIGHUP are reset to what they were prior to the invocation of the transaction. Then, if `on_abort_return` has been specified, this FUN/PRED transaction returns the value directed. Otherwise, if a signal had caused this abort, then the signal catching function for this signal prior to transaction invocation is called, if it is not the system default action. If it returns, then the process will exit with status equal to the signal number. Consequently, the only way to prevent the process from exiting in the case of a transaction abort is to use `on_abort_return`.

As mentioned above, users may call the no-argument Abort PROC at the Cymbal level. If the transaction fpp is a PROC and Abort is called, then rollback occurs and Exit is called causing the program to end with an exit status of `_sigleave_` (which is 99). On the other hand, if the transaction fpp is a FUN/PRED and Abort is called in its body, then the fpp must have been defined with an `on_abort_return` keyword argument to specify what the FUN/PRED is to return after the rollback has occurred. Notice that Abort not only causes rollback but in effect it causes a non-local goto from wherever the program logic is when Abort is called (e.g., in a nested function call) all the way back to the top-level task function where the rollback occurs and the return value, if any, which could well be a function call or variable dereference, is produced. In this case, the program does not exit but rather the transaction returns and the user may take whatever Cymbal action is deemed necessary to handle the abort.

As illustrated in btclustelt.1.IQU, Cymbal’s try-else blocks can be used to cause execution to carry on after any transaction abort:

```c
try {
    do Delete_Clustelt( "tan", 2);
} else {} // the empty else is needed
```

### 17.10.9 File Descriptors And Transactions

There are two styles of transactions that are indicated by the `flush_on_return` and `close_on_return` keywords as in:
define PRED close_on_return transaction task:

If a transaction is flush_on_return, then when it returns from an invocation, it flushes its results to the buffer cache but it leaves all of its files open, ready to be reused by the next invocation. If it is close_on_return, then it closes all of its files at the end of each invocation and, of course, opens them all again at the start of each invocation. flush_on_return transactions are much faster, which is why they are the default for Cymbal transactions. However, if a Cymbal user is running out of file descriptors, then close_on_return will help to conserve and reuse them. close_on_return is the default for SQL transactions, since this is a notion that is foreign to SQL syntax and since Daytona must therefore take steps to ensure that SQL transactions never run out of file descriptors.

17.10.10 Caveats

Currently, Daytona does not support nested transactions. In particular, this means that, in general, Dayton does not allow a transaction to call another transaction. This may appear to seriously inhibit procedural abstraction whereby certain update functionality is abstracted into fpps for multiple invocation. Fortunately, the helper fpps live up to their name: it is perfectly permissible for do Change calls to appear within the helper fpps for a given transaction task (nextord.IQA).

As remarked in the for_each_time section in Chapter 6, be careful not to call transactions in a for_each_time do-group that change some table that is being looped over in the for_each_time assertion.

17.10.11 Use Of Daytona In Shared-Disk Clusters

Given a POSIX-compliant cluster filesystem, Daytona transactions in general work fine except when used inappropriately in a cluster node failure situation. First though, it is important to keep in mind that there is only one cluster filesystem that has been shown to be POSIX-compliant for Daytona use and that is the Veritas cluster vxfs and it is not cheap.

Now in more detail, when a POSIX-compliant cluster filesystem is being used, then transactions can be run from any node on any and the same tables just as they can when using a single SMP machine. This is because both fcntl file locking and lock files are supported by the combination of Daytona and a POSIX filesystem. Also, Sizups can be run on any nodes on any and the same tables.

As stated before, the only way to get (data) durability in the face of system crashes is to turn on logging. Note however that in the cluster setting, logging can only support durability when the transactions are constrained as follows: either only one node does all of the transactional changes or what amounts to be the same thing really, multiple nodes can be making changes but only to (shared) tables that each node exclusively is allowed to change. In other words, no two nodes can be creating logs that specify changes to the same tables. Since the logs would presumably be on shared disk, any one of the surviving nodes could run Recover. In order to quiesce the system to enable Recover to do its work correctly, every txn can check for the existence of a greenlight file and either exit or start polling for the go-ahead if one is not found.

A word on qfs. As of 2012, the cluster filesystem qfs is not POSIX-compliant because it fails miserably on Daytona’s transaction stress test (which Veritas passes). Nonetheless, people use qfs with
Daytona -- even though it has other problems as well. The point is that none of these problems are Daytona problems and so the Daytona team will not fix them. However, there is a heavy-duty production system using qfs that has not had any concurrency problems but note though that it updates only via file appends.

And of course, since NFS is not even remotely a cluster filesystem, let alone a POSIX-compliant one, Daytona transactions over NFS are absolutely not supported.

17.11 Logging Transactions and Using Recover

To enable transaction logging simply add an appropriate keyword to the Cymbal transaction definition. For example, to convert the first transaction example in this chapter into a logging transaction, one writes:

```
global_defs:
define PROC with_logging transaction task:
    Delete_A_Part( with_nbr INT .part_nbr )
{
    do Change so_that( there_is_no PART where( Number = .part_nbr ) );
}
```

The three possible keywords to invoke logging are:

- `with_logging`,
- `with_logging_optional`, and
- `with_logging_flag BOOLEAN`.

The `with_logging_optional` keyword is used in conjunction with the extern C variable `log_option`, which is set either explicitly to `TRUE` by the user in some user-written (and linked-in) C code or, in the case of query executables generated by Daytona, via command line arguments (+L for `TRUE` and +NL for `FALSE`, the default). For the `with_logging_flag` keyword, the logging status is determined by the value of the supplied BOOLEAN expression. The `with_no_logging` keyword is implied if no other logging keyword is used.

Log files, which have a `RMLOG` prefix, are generated on a per process basis and deposited in the directory indicated by the `Txn_Log_Dir` note to the PROJECT description. There is no default transaction log directory -- the user must specify one by using a `Txn_Log_Dir` note in a pjd. The `Txn_Log_Dir` value may be a shell expression that is expanded at runtime by logging transactions. Transaction logs exceeding `Max_Log_Files_In_K` are continued as needed in new log files of length at most `Max_Log_Files_In_K`. `Max_Log_Files_In_K` can be set by a `Max_Log_Files_In_K` note in the PROJECT description or overridden at runtime by using the query command-line argument +MLF. The default value is 10 megabytes.

After a system crash, the `Recover` command should be run before any new database activity occurs. When run without the `-l` argument, `Recover` will perform database recovery, using the value of `Txn_Log_Dir` in the pjd to locate the logs to process. However, it is good practice to first run `Recover` with the `-l` (mnemonic: list) option. The list option will cause `Recover` to retrieve all transaction log IDs...
from the log files and generate, on standard output, a time-sorted list containing the following tab-delimited information: transaction log ID, transaction commit second, transaction commit micro-second, and a 0-1 rollback indicator. The transaction log ID consists of the full path name of the log file where the logged transaction begins and the byte offset within that file of the transaction separated by an at-sign (@). The rollback flag indicates that the associated transaction needs to be (and will be) rolled back (i.e., undone). The ordering of transaction logs in the listing is the ordering that Recover will use when it is performing recovery. By using the list option, the integrity of the log files can be checked without any logs actually being executed. In the case of a system crash, a few warnings might appear concerning incomplete logs as the list is generated.

When Recover is run for recovery, it ignores or rolls back (if there were some intermediate saves during the transaction) the first incomplete transaction it encounters and then terminates. It is the responsibility of the database administrator to determine the appropriate action to take in the presence of incomplete logs. Generally, any subsequent incomplete logs uncovered using the list option should be run manually by supplying the transaction log IDs as arguments to the Recover command.

The Clean_Logs daemon can be run continuously in the background to periodically remove old log files to keep disk usage from getting out of control. In these circumstances, the protection being offered is to protect against system crashes: after such a crash, Recover is run to redo the various logs of transactions that were interrupted by the crash or whose updates had not yet made it successfully to disk. So, obviously, Clean_Logs should not be allowed to remove any log files for completed transactions whose updates have not successfully made it to disk. This translates into not removing log files that are too new. The safety aging interval here is platform specific; Clean_Logs uses a default of an hour. However, in order to be able to recover from disk media failure, which is an entirely different failure scenario, Clean_Logs should not be run at all, thus promoting the preservation all log files since the last backup.
18. Modes Of Use: Producing Answers, Executables, Code And Packages

There are at least two different taxonomies categorizing how to use Daytona. One has to do with the degree to which the database functionality is used. The other has to do with the uses to which Daytona-generated artifacts are put.

18.1 To The Database Or Not

Traditionally, the storage of data in a DBMS (database management system) entails using a declarative query language like SQL, creating a data dictionary, creating indices, using concurrency methods and transactions, using logging, and reformatting the data into a proprietary format. All of these are certainly supported by Daytona but they are not necessary. In fact, Daytona users typically don’t use the logging capability and of course, were it not for the optional choice of using HEKA types and record-level data compression, Daytona’s data would be stored in the ASCII flat file format that is so universal, readable, and friendly to other tools -- no reformatting needed!

What is of interest is the degree to which the other accoutrements may be used or not. Firstly, Cymbal can always be used just as a programming language with the same general capabilities as Perl, although with the additional benefit of superior speed. It’s easy to tell when Cymbal is being used as a programming language because there are no `there isas` present betraying a dependence on rcds (i.e., a data dictionary). Consequently, in this setting, there are no transactions although nonetheless, `new_channel` would offer support for concurrency by means of explicit locking. Likewise there are no (persistent) indices when Cymbal is being used as a programming language since those are defined using rcds. This means that any processing of data in files will necessarily be sequential; of course, the user has the option of creating associative arrays and boxes for in-memory indexing purposes.

However, Daytona also offers an interesting hybrid mode of use, that is, one where the user does define and use rcds (data dictionaries) but yet there are no indices, there are no required concurrency mechanisms via locking, and there are still updates in the way of blind appends (which are intrinsically concurrent, whether asked for or not). There can even be cases where the information Daytona processes is not even being stored in files that are directly readable by Daytona: this is the case where Daytona receives data in a format that it can understand via rcds via third-party programs piping into Daytona from sources and formats totally beyond Daytona’s influence (and perhaps even the permissions of the Daytona user to access directly). One example of this is where a pipeline uncompresses binary netflow data and pipes into a third-party custom C program which produces ASCII flat file output understandable by Daytona rcds. This lightweight use of rcds offers a lot of advantages, including referring to fields by name instead of number, automatic conversion of field values between their storage representation and their in-memory representation, the use of horizontal partitioning to manage access to potentially thousands of files, and all the other advantages listed in the section describing blind appends -- which are also available.
18.2 Utilizing Answers, Executables, And Code

As a system, Daytona takes queries as input and produces three kinds of output: answers, executables, and code. Depending on the application, some of these products will be of more interest than others. For example, users who are using Cymbal to analyze some special project data are interested almost exclusively in the answers that Daytona produces; they wouldn’t be particularly interested in the executables (unless they wanted to parameterize them for subsequent invocations) and they certainly wouldn’t have any interest in the C code that is produced.

Daytona users in the second and third categories use Daytona for ‘generating methods’ instead of ‘generating answers’. Specifically, these latter two categories consist of application developers whose applications have serious data management needs. Instead of writing the necessary data management code themselves, they are, in effect, hiring Daytona as a programmer to write the relevant methods/algorithms/code for them.

The latter two categories differ according to the degree of closeness of the interaction of the Daytona-generated code with that of the application itself. The second category holds the Daytona-generated methods at arm’s length by interacting solely with Daytona-generated executables. This situation would occur, for example, when constructing an end-user application using GUIs (graphical user interfaces) which interact with their environment by means of a shell interface. In this situation, the GUI dialogs with the end-user in order to first determine what data to retrieve and/or changes to make and then accomplishes these tasks by means of invoking Daytona-generated, parameterized executables through the GUI’s shell interface.

The third category employs what Daytona calls code synthesis, i.e., the creation of a "Daytona sandwich" wherein application C code calls Daytona-generated C code which may itself call other application C code (using the C extensibility feature of Daytona). This amounts to enabling the developer to construct a customized database procedure library (or a data manager library, in the sense that all of the routines for managing the application’s data are in this library).

Instead of code synthesis, the analog with other systems would be to use their low-level, usually record-at-a-time subroutine library interfaces; such interfaces don’t amount to much more than offering the capability to open and close tables and to get, unpack, and modify records one-at-a-time. The same tedious style of programming holds with the more modern cursor-based Java-, Perl- and C-embedded SQL paradigms. Daytona offers instead the opportunity to write much higher level routines which make full use of the power of Cymbal to do entire multi-functional database tasks, indeed, tasks which would encompass dozens of low-level, one-at-a-time-record subroutine calls from many tables. Later in this chapter, there is an extensive example showing how superior Daytona’s code synthesis approach is to solving the kinds of problems commonly addressed by using the cursor-based embeddings of database query languages in procedural languages.

Speed is also a clear advantage of the third category approach due to the tight binding of the data management functionality with the application code itself: after all, the fastest possible coupling of code on a computer is by means of a function call. Note the contrast between this tightly bound single-threaded approach and the two-threaded approach embodied by C-embedded SQL programs that converse with a database server over a socket.

Cymbal packages provide the best way to create C libraries to support code synthesis -- and to pre-compile Cymbal functionality for automatic inclusion in standalone Cymbal programs. A Cymbal
package is a specification of a collection of (related) Cymbal tasks that are compiled as a unit and whose corresponding compiled code is stored in a C library for linking with other C or Cymbal program code. (ADA and Oracle’s PL/SQL also employ the package concept.) Once again, not only can Cymbal packages be linked against user C code that calls the associated C-compiled Cymbal tasks, but the tasks in Cymbal packages can be called from within 100% Cymbal programs just by using an import package statement.

As it turns out, there is a fourth mode of use, which is where a user program generates sophisticated adhoc Cymbal code on the fly and then processes that as in the Daytona-for-answers mode.

The rest of this chapter presents the strategy and tactics involved with using Daytona in each of these four modes.

18.3 Daytona For Answers

Using Daytona to get answers involves the following simple steps:

1. **Make sure the data is in an appropriate format.**
   An appropriate format, of course, is a UNIX flat file format, possibly extended with LIST/SET-valued fields and comments. The #msg)flds) comment is an especially useful one since it is used by DS DC−rcd to identify the names of fields (otherwise, Field_1, Field_2, . . . are used).

2. **Generate an rcd for the data and put it in an aar.**
   The easiest way to get a first approximation to an rcd for a given data file is to use DS DC−rcd. Please note that using an actual data file is not really necessary since DS DC−rcd will work just fine on a data file consisting of the #msg)flds) comment and one line of prototype data. DS Vi can be used to modify this rcd so that appropriate keys are defined, suitable maximum STR field lengths are specified, etc. Other reasonable ways to generate an rcd include using DS Archie to extract an existing one from an aar and then using a text editor to modify that and DS Archie to put it back in again. It is not recommended that the user try to generate an rcd from scratch in a text editor because of the likelihood that pound-braces will not match or that some other syntactic error will occur.

3. **Write a query and run it.**
   The Dice menu interface and DS QQ makes it easy to write and run Cymbal and SQL queries. The data index builder Sizup will be called automatically as needed; if the user wishes to invoke it directly, just use DS Sizup or the appropriate menu choice in Dice.
Ad-Hoc Queries For Getting Answers

In this situation, a developer writes a probably parameterized Cymbal query and uses either *DS Compile* or its *Compile Query* analog from Dice to compile the query into an executable. The paradigm here is that other software will determine appropriate parameters to pass to a Daytona compiled query and, will use either pipes or files to get and process the corresponding query output.

There is a full range of options with regard to parameterizing executables. Chapter 8 discusses many ways to get information into a Cymbal program: these consist of the command line, standard input, files, pipes, and (given sufficient user interest) sockets and shared memory.

Chapter 3 explains how to easily incorporate a *-?* or *-U* usage option into Daytona-generated
Here is a Cymbal program that processes its command line argument to tell it what action to perform and then reads a bunch of records from stdin into a box to serve as the basis for performing the action. The connection between INPUT_EMPLOYEE and stdin is established in the rcd (see Chapter 23).

```cymbal
set [.action] = read( from _cmd_line_ ) otherwise do Exit(2);
set .in_box = [ [ .name, .age ] :
    there_is_an INPUT_EMPLOYEE where(
        Name = .name
        and Age = .age ) ];
switch(.action){
    case( Matches "pr.*" ){
        for_each_time [ .name, .age ] Is_In .in_box
do {
            do Write_Words( "Name =", .name, "Age =", .age );
        }
    }
    case( = "dump" ){
        set .out_chan = new_channel( with_mode _append_ for "/etc/safe" );
        for_each_time [ .name, .age ] Is_In .in_box
do {
            to .out_chan
do Write_Words( "Name =", .name, "Age =", .age );
        }
do Close( .out_chan );
    }
}
```

Copyright © 2013 AT&T
All Rights Reserved.
September 15, 2013
18.5 Code Synthesis: Daytona For Code

Recall that code synthesis occurs when the user’s own application C code calls C code generated by Daytona from a Cymbal specification, which in turn may call other user application C code.

There are essentially two steps involved in using Daytona to generate C code. The first is to write the Cymbal definition of the target data manager library functions and the second is to generate and compile the corresponding .c’s with their .h’s into the .o’s that will make up the data manager library.
18.5.1 Writing Cymbal For A Data Manager Library

As an example, consider this scenario: the user wants to have a C program that

1. is invoked with a sequence of file names as command line arguments,

2. calls a generated-from-Cymbal C function `get_data_from_file` for that sequence of files that repeatedly loads data into arrays for upto 1000 table adds at a time,

3. calls a generated-from-Cymbal C function `load_data` that loads each bunch of adds as a single transaction, thus reducing the per-add transaction overhead.
Note that all this could be handled within Cymbal exclusively but for pedagogical reasons, we wish to illustrate C calling Cymbal (more accurately, C code calling C code generated from Cymbal).

Here is the C program (Main_Bulk_Loader.c):

```
#include "stdio.h"
extern long get_data_from_file(char *);
extern long load_data(
    long count, char name[][30+1], char phone[][25+1]);
char name[1000][30+1];
char phone[1000][25+1];
long total = 0;
int
main( int argc, char **argv )
{
    int idx;

    { extern void Initialize_Constant_Channels_Modulo( int );
      Initialize_Constant_Channels_Modulo( 0 );
    }
    /* no Sizup call here! (but could be in general) */
    for( idx = 1; idx <= argc-1; idx++ )
    {
        while( get_data_from_file( argv[idx] ) != 0 )
        {
            fprintf(stderr, "just loaded %ld records from file %s\n",
                load_data( total, name, phone ), argv[idx]);
        }
    }
}
```

Note the required C declarations of the C function translations of the two Cymbal functions. Also, obviously when doing code synthesis, the user will need to provide the C main() routine, since Daytona will not generate one. One consequence of this is that Sizup will not be called automatically by the system to ensure that any indices are up to date. The user may instead arrange for the application to call Sizup at appropriate times.

Another consequence is that in order to use I/O CHANNELS in the Cymbal code, the user must execute the following C code:

```
{ extern void Initialize_Constant_Channels_Modulo( int );
  Initialize_Constant_Channels_Modulo( 1 );
}
```

prior to executing any C-code generated from Cymbal that uses CHANNELS. Putting such a block at the beginning of the C main() would be a good idea.
Initialize_Constant_Channels_Modulo is 1 if the _popkorn_ CHAN is to be opened and 0 otherwise.

Another scenario occurs when there is a Cymbal main (i.e., Begin) that does not use _popkorn_ that calls user-generated C code that calls Cymbal routines that do use _popkorn_. To get _popkorn_ enabled in the Cymbal main, simply include the call:

```c
    do Initialize_Constant_Channels_Modulo( _true_ );
```

In general, a Cymbal-specified data manager library consists of query files, each of which consists solely of a sequence of Cymbal C_external task definitions. These tasks embody a high-level way of specifying sophisticated data management functionality for use by C applications. Note that it is important that there not be any Begin task in such a query file, either explicitly or implicitly: in other words, there must not be any statement in the file which Daytona is being told to execute: the file must consist solely of task definitions. Following along with the current example, here are the associated Cymbal task definitions (taskdef.2.PQ):
global_defs:

define C_external INT FUN( STR(=) .src_file ) task:
  get_data_from_file
{
  imports C_external: STR(30) .name[ [ 1 -> ] ]
  STR(25) .phone[ [ 1 -> ] ]
  INT .total

  local:
  static STR(*) .prev_file
  static CHAN(_file_) .in_chan

  when( .src_file != .prev_file ){
    if( .in_chan Is_Open ) do Close( .in_chan );
    set .in_chan = new_channel( for .src_file )
    otherwise { do Exit( 1 ); } 
    set .prev_file = .src_file;
  }

  for_each_time .ii Is_In [ 1 -> 1000 ] do {
    set [ .name[.ii], .phone[.ii] ]
    = read_words( from .in_chan )
    otherwise_switch {
      case( = _instant_eoc_ ) { break; }
      else{ do Exit(1); } 
    }
    set .total++;
  }

  return( .total );
}

define C_external INT FUN transaction task:
  load_data( INT .count, STR(30) .name[ [ 1 -> ] ],
            STR(25) .phone[ [ 1 -> 999 ] ] )
{
  local: static INT .nbr_base = 1000

  for_each_time .ii Is_In [ 1 -> .count ] do {
    insert into SUPPLIER ( Number, Name, Telephone )
    values( (:INT(_short_)):)(.nbr_base+.ii),
    .name[.ii], .phone[.ii] );
  }

  set .nbr_base += .count;
  return( .count );
}

There are a number of interesting points to make about this example. First, observe that both of
these tasks are modified by the C_external keyword. A C_external fpp is one whose generated C
name is guaranteed to be the same as the Cymbal name: otherwise, Daytona might well add a prefix or
suffix in creating the C name in order to ensure that there will not be two C functions with the same
name. (Consequently, users of C_external must warrant the uniqueness of their names.)
C_external also has the analogous effect on variable names; note that when it is used as part of an
import statement itself, it modifies all subsequent imports.

The first task, get_data_from_file, communicates with the application code that calls it by importing
3 exported (i.e., global) variables and by taking one argument. Two of the imported variables are
actually C arrays even though they are being treated as Cymbal ones whose indices start at 1, instead
of 0 as with C. Daytona will automatically map Cymbal indices in these situations so that they start
the access at index 0 in C arrays. The argument is a STR(=) variable which is translated into a char
* at the C level. STR(=) as a type should only be used when Cymbal refers to variables that have
been defined externally in C as char * and are being imported or passed as arguments to Cymbal
functions called from C. Any other use of STR(=) is strongly deprecated for a number of reasons
including lack of copy semantics and non-inclusion in Daytona’s string garbage reclamation system.

On the other hand, load_data chooses to obtain information from the outside strictly by means of
parameters, one of which is a one-dimensional STR(30) array of unspecified length (although actually
we do know what it is in this example).

Note the utility of static variables here.

An important caveat is that if you are calling a Cymbal STR FUN from C, then, since the STRING
that the function returns is considered to be reclaimable as garbage later by Daytona, then you must
copy this value into some local C variable should you wish to preserve it. Also, the general rule is that
since Daytona will automatically manage its own string garbage, the user must never free strings that
are allocated by Daytona. In other words, the user may or may not have their own garbage
reclamation system and Daytona certainly has its own -- and never the twain should meet. Basically,
user C code should have nothing to do with the C implementation of Cymbal STR(*) values.

18.5.2 Compiling A Data Manager Library

To compile a Cymbal file, say, mytasks.cy, consisting of global_defs: followed by several task
definitions into a collection of .o’s, one for each task, simply say:

DS Compile mytasks.cy

Since there is no Begin task in such a Cymbal file, Daytona will simply produce .o’s instead of an
executable -- and it will construct one .o per task. These .o’s can be put into some ar(1) data manager
library of the user’s specification. (Note that this is handled automatically when using Cymbal
packages, as described next.)

If these tasks make use of Daytona tables, then the data manager library will also have to include
.o’s for the fio files (i.e., <RECORD_CLASS>.o) and likewise regarding any need for any compiled
Daytona environment files. Here is an example of how to create such a supporting library of .o’s:
$ cd ${HOME}/dict

$ rm *.o

$ Tracy -app myapp -gen_fio_for_app myapp

$ ar rc myapplib.a *.o

### Adding any necessary C-level env files:

$ ar rc myapplib.a .usr.env.o \
    .orders.env.o \
    .daytona.env.o \
    /work/rxga/DS.g/d/daytona.o \
    /work/rxga/DS.g/d/orders_aux.o \
    /work/rxga/DS.g/d/daytona_aux.o

In order to link against this data manager library to create a hybrid executable consisting of compiled user application code and Daytona-generated code, it is necessary to mimic the Daytona make process. Whatever it is, it is guaranteed to be embodied in the R.mk file generated for the compilation of regular queries. (This can be obtained by running Tracy since DS Compile and QQ will erase C and make artifacts.) And in fact, here is the make file for the Bulk_Loader example (Bulk_Loader.mk):

```make
include $(DS_DIR)/DS.mk.rules

QUERY_OBJS = Main_Bulk_Loader.o \
    get_data_from_file.o load_data.o

APP_LIB = myapplib.a

Bulk_Loader: $(QUERY_OBJS)
    $(CC) $(DS_LDFLAGS) $(QUERY_OBJS) $(APP_LIB) \
        $(DS_DIR)/libR_DC.a \
        $(DS_DIR)/libR_RC.a \
        $(DS_BTDIR)/libcbt.a \
        $(DS_DIR)/libDS.a $(DS_DIR)/libRsys.a \
        $(DS_DIR)/libEnv.a $(LDLIBS) \
        -o Bulk_Loader
```

The $(QUERY_OBJS) correspond to the task _name_.o.

Happily, by using the mk script in $DS_DIR, the entire process to build the executable for this example is a two-liner:

```bash
DS Compile taskdef.2.PQ
$DS_DIR/mk Bulk_Loader
```

Note that mk assumes the make file is named arg_1.mk where arg_1 is the argument it is invoked with, which in this case is Bulk_Loader.
18.5.3 Code Synthesis Quality Considerations

Code synthesis has a lot of good things going for it but it is accompanied by one peril: since the application’s C code and the Daytona-generated C code cohabit in the same process, a bug in one can show up in the other. This is particularly true for memory corruption problems. The problem is that it is not necessarily clear who should be responsible for fixing the bug.

18.6 Code Synthesis v. Embedded SQL, Modules, CLI, ODBC, JDBC, RDA, etc.

The SQL community has many different (and complicated) paradigms for interfacing SQL databases to general-purpose programming languages. Daytona has two and they are a lot simpler. On a single machine, the most natural paradigm is code synthesis (for C). In a distributed setting, when the Daytona database is on another machine separate from the client machine, then a program in any language on the client can utilize pdq’s simple ASCII text protocol to converse with a Daytona pdq query daemon on the remote server machine. pdq is discussed in Chapter 22.

On the code synthesis front, a very common situation occurs when the user has an application C program which needs to run a query on the database and then loop over the answer rows one by one in order to process them with application-specific C code. In comparison with Daytona’s competitors, Daytona’s code synthesis provides a remarkably simple and efficient way to do this. The following example consists of a C file (orderstat.c), a Cymbal file (orderstat.PQ), and a make file (orderstat.mk) . The paradigm is that of a C-Cymbal-C sandwich where the main C program calls a C function generated from a Cymbal task which itself loops over all TUPLES satisfying a query. As it does so, it calls the user’s own C function on each answer TUPLE. The primary goal of the following example is to identify all ORDERS for a given SUPPLIER which were processed in fewer than a given number of days. A secondary goal is to compute and report the total weight of all the PARTS for all such ORDERS.

The following C driver routine contains a main() routine which calls the C-from-Cymbal routine Find_Prompter_Orders on each supplier/promptness_bound pair read in from the command line. The only other C function in the file is Process_A_Prompt_Order which is called on each of the answer TUPLES that Find_Prompter_Orders produces. Obviously, this C driver file needs and gets a C extern declaration for Find_Prompter_Orders. The external tot_wt variable will accumulate the total weight of all the PART orders processed in the given number of days for the given SUPPLIER.
#include "stdio.h"

extern void Find_Prompter_Orders( char *supplier, char *promptness );

double tot_wt;

int
main( int argc, char **argv )
{
    int idx;

    { extern void Initialize_Constant_Channels_Modulo( int );
        Initialize_Constant_Channels_Modulo( 0 );
    }

    /* no Sizup call here! (but could be in general) */

    /* command line consists of supplier/promptness_bound pairs
       as in: orderstat "Philip Export" 1000 "Standard AG" 500 */

    for( idx = 1; idx+1 <= argc-1; idx++ )
    {
        tot_wt = 0.0;
        fprintf( stdout,
            "\nOrders processed in less than %s days from %s\n",
            argv[idx+1], argv[idx] );
        fprintf( stdout,
            "Order_Nbr Part Processing_Days\n" );
        Find_Prompter_Orders( argv[idx], argv[idx+1] );
        fprintf(stdout,
            "\nTotal weight over all these orders is %g\n", tot_wt);
        idx++;
    }
}

void
Process_A_Prompt_Order( long onbr, char *part, long delta )
{
    fprintf(stdout, "%5ld %15s %10ld\n", onbr, part, delta);
}

All that the C function Process_A_Prompt_Order does is to print out the answer TUPLE; obviously though, the opportunity is present to apply arbitrary application logic to each TUPLE.

Here are the contents of the Cymbal task definitions file that is used to generate the required C function Find_Prompter_Orders.
global_defs C_external:
define PROC( STR(30) .supplier, STR(=) .pbound ) task: 
    Find_Prompter_Orders
    
    import C_external:
    PROC( INT, STR(=), INT ) Process_A_Prompt_Order
    FLT .tot_wt 

    when( .pbound Is_Not_An_Int_Str 
    }
    with_msg "promptness bound .pbound can’t be converted to INT"ISTR
    do Exit(1);
    }

    for_each_time [ .onbr, .part, .delta, .wt, .qty ] 
    is_such_that( 
        there_is_a SUPPLIER where( Name = .supplier and Number = .sno ) 
        and there_is_a ORDER where( 
            Number = .onbr and 
            Supp_Nbr = .sno and 
            Part_Nbr = .pno and 
            Date_Placed = .dp and 
            Date_Recd = .dr and 
            Quantity = .qty ) 
        and there_is_a PART where( 
            Number =.pno and 
            Name = .part and 
            Weight = .wt ) 
        and .delta = .dr - .dp 
        and .delta < (INT).pbound 
    }
    
    do Process_A_Prompt_Order( .onbr, (STR(=)).part, .delta );
    set .tot_wt += .wt * .qty;
    }

The special Cymbal type STR(=) is used to communicate C char * variables into Cymbal programs. Note that most means of communicating outside C information into a Cymbal program are being used here, i.e., C parameters, imported C variables, imported C functions. C functions could even be passed in as parameters. Of course, all this is contingent on working with types that are shared between Daytona and C. Obviously, these include char *, long, double, float, short, long long. For the more sophisticated Cymbal types like IP, HEKA, DATE_CLOCK, the user is advised to send them between C and Cymbal by means of character strings that can be casted appropriately upon receipt. It’s too risky and too much trouble to try and use Daytona’s C implementations of these types.

This query is a kind of control-break/report-writer query where individual line items are being
produced and then there is a summary statement made about the totality, i.e., their total weight. Note
the freedom one has in pushing logic into C or Cymbal. For example, the entire tot_logic could be
pushed into Cymbal exclusively (not to mention this entire pedagogical exercise).

Here is the simple makefile:

```bash
include $(DS_DIR)/DS.mk.rules
QUERY_OBJS = orderstat.o Find_Prompter_Orders.o

Orderstat: $(QUERY_OBJS)
  $(CC) $(DS_LDFLAGS) $(QUERY_OBJS) \n    ORDER.o SUPPLIER.o PART.o \n    $(DS_DIR)/libR_DC.a \n    $(DS_DIR)/libR_RC.a \n    $(DS_BTDIR)/libcbt.a \n    $(DS_DIR)/libDS.a $(DS_DIR)/libRsys.a \n    $(DS_DIR)/libEnv.a $(LDLIBS) \n    -o orderstat
```

Here are the simple compile/build/run commands.

```bash
DS Compile Q/orderstat.PQ &&
$DS_DIR/mk orderstat &&
orderstat "Philip Export" 1000 "Standard AG" 500
```

### 18.6.1 Other Programming Language Database Interfaces

First, regarding the previous example, observe what is not there.

1. There is no separately running database server process that needs to be connected to and
dialogued with over a slow socket.

2. There are no driver, database, or statement handles.

3. There are no cursors that point to particular answer tuples.

4. There are no Daytona system C header files to include (and use).

5. There are no database error return codes.
   Any error handling is done locally at the C or Cymbal site where the error could occur. Thus
   there is no need in general to communicate a code for an error to some other place for some other
   program logic to handle it.

These and more inconveniences are present to varying degrees in the many different so-called
"binding styles" that the SQL community has developed to couple SQL databases with programming
languages. Here are some of them.
• Module language

Here all SQL statements needed by the main host program are wrapped into SQL procedure definitions and collected all together into one or more homogeneous SQL files along with declarations for cursors and temporary tables. These SQL procedures have scalar parameters that can give values to their SQL statement and return values produced by it. The host program then uses its own procedure call mechanism to call these SQL procedures.

• Embedded SQL

Here SQL statements are embedded in the host programming language using special syntax that a macro-preprocessor can detect so as to expand them into their implementation in the host language; then the host language compiler compiles the ensemble. By the way, this can cause confusion for host language debuggers, the likes of which would not happen when using Daytona code synthesis.

• Dynamic SQL

This goes further and supports the handling of SQL statements, in particular queries, which are not known at the time of the compilation of the host program. Such handling is not supported by Daytona code synthesis but could be addressed by using pdq.

• SQL/CLI (Call Level Interface) (standardizing ODBC)
• JDBC (for JAVA)
• Perl DBI (database interface)
• Python DBAPI
• Oracle Call Interface (OCI)
• MySQL C API
• RDA (Remote Database Access)

Daytona’s code synthesis is structurally similar to the module approach but also differs in a couple important ways. First and foremost, 4GL Cymbal tasks offer far more functionality than the simple SQL DML statements that most if not all module implementations are restricted to, given that you can find one at all. This counters one of the criticisms leveled against the module language namely that it requires people reading the code for a program to constantly look back and forth between two different files, one with the host program and the other with the SQL modules: for Daytona, the tasks can be so comprehensive and expansive in their functionality (i.e., do so much) that there will be fewer of them and in that case, their size alone would suggest placing them lexically away from where they are called anyway, which is what one would commonly do with large subroutines. Secondly, in contrast to code synthesis, there is no provision in SQL for calling host programming language statements from within SQL modules and thus the extra baggage of working with cursors is required.

The rest all pretty much have the same flavor which is good because it enables Daytona’s pdq to be used as the common query server for Daytona’s JDBC, Perl DBI and Python DBAPI interfaces. Here is a skeleton MySQL C API prototype database interaction that amply illustrates the typical complexity of
these interfaces.

```c
#include <stdio.h>
#include <mysql.h>

MYSQL *conn;
/* define and set:
   host, user, passwd, db, port, socket, flags */
MYSQL_RES *result_set;
MYSQL_ROW row;

int
main( int argc, char *argv[])
{
    char *sqltxt = (char *) malloc(1000*sizeof(char));
    char *supp_nbr;
    int i;
    char * order_nbr;
    double order_qty;

    if( argc >= 2 ) supp_nbr = argv[1];
    else { /* handle error */ }

    if( (conn = mysql_init(NULL)) == NULL )
    { /* handle error */ }
    if( mysql_real_connect( conn, host, user, passwd,
                           db, port, socket, flags ) == NULL )
    { /* handle error */ }

    sprintf(sqltxt,
            "select Number, Quantity from ORDER where Supp_Nbr = \'%s\';",
            supp_nbr);
    if( mysql_real_query(conn, sqltxt, strlen(sqltxt)) != -1)
    {
        if( (result_set = mysql_store_result( conn ) == NULL )
            else if (mysql_errno(mysql)) {
                fprintf(stderr,
                        "Error mysql_store_result: %s\n", mysql_error(mysql));
            }
        while( (row = mysql_fetch_row(result_set)) != NULL )
        {
            order_nbr = row[0];
            /* do your own casting for basic types */
            order_qty = (double)row[1];
```
The MySQL C API is discussed in Paul DuBois’ MySQL book in a chapter extending for 53 pages. This is nothing compared to the hundreds of pages that make up typical ODBC or JDBC treatments. Perl DBI provides the simplest interface of this kind; of course, it is of limited value to application code written in C. In any event, it should be apparent that Daytona’s code synthesis is much simpler than these other approaches.

18.7 Cymbal Packages

A Cymbal package is a self-identified UNIX file containing one or more C_external Cymbal task definitions (e.g., aparm.cy) (and no statements to execute) that Daytona will use to create and maintain a compiled version thereof in an ar(1) archive. So, a Cymbal package enables a user to pre-compile related Cymbal functionality into a library that can and will be automatically linked into every subsequent Cymbal query executable by means of a package import statement.

This can be achieved simply by:

1. Create the Cymbal package file as described next.

2. Either in the given Cymbal program file that will use the package, or (for automatic, general use) in one of the Daytona *.env.cy files, simply include a package import statement, as for example:

   import: package `mypkg`

The effect of this is to import all the definitions and declarations given in the package, thus saving potentially quite a bit of typing on the user’s part to do the imports manually one by one. Furthermore, whether or not any of the contents of the package are actually used by the query, one effect of the package's import is that the associated archive of .o's will be linked against in creating the query’s executable.

The file of Cymbal task definitions making up the package is as before in that all of the fpps must be defined as C_external and there must not be any Cymbal statements to execute -- just definitions. In addition, the in_package keyword to task definitions is used to specify the library that gets the .o's that correspond to each defined task. The in_package keyword takes a STR, LIT, or THING argument, say, pkg, which will cause Daytona to put all the .o’s into pkg.a . The name of the Cymbal file containing the package pkg must have the form pkg.cy . An easy way to specify the package name for every task in the file is to modify the global_defs phrase as follows:

   global_defs C_external in_package `mypkg` :

This will be inherited by every succeeding task definition that it scopes, unless that task definition has its own in_package specification. However, this is less freedom here than the syntax suggests. First, any
given Cymbal package file can only refer to one package. Secondly, since the archive file is created anew each time the task definitions in a file are compiled, it is not possible to have task definitions for the same package in different files.

If there are files which this package depends on, in the sense of requiring the package to be remade if that file is changed, then a \texttt{depends_on} TUPLE can be included in the \texttt{global_defs}:

\begin{verbatim}
global_defs C_external in_package ^perquery^
  depends_on[ ^$(ORDERS_HOME)/orders_aux.c^ ] :
\end{verbatim}

See Chapter 3 for more on \texttt{Depends\_On}, which is the apd/pjd-level syntax for this same concept.

Daytona will process packages automatically. However, should the user wish to create the package library manually, that can be achieved by executing:

\begin{verbatim}
DS Compile ourtasks.cy
\end{verbatim}

where the package file is \texttt{ourtasks.cy}.

By the way, behind the scenes, Daytona automatically modifies the MAKE\_GOODS description for either the sole apd (assuming a singleton $\texttt{DS\_APPS}$) or else an otherwise mandatory pjd. Here is an example modification taken from the test suite from \texttt{pjd.daytona}:

\begin{verbatim}
#{ MAKEGoods
  ...
  #{ FILES
    ...
    #{ LIBRARY ( liborders.a )
      <In_Package, liborders>
      <Source, /home/john/d>
      <Project, daytona>
      <__System_Generated, yes>
    #{ FILE_BASE ( SUPPLIER )
      <Source, /home/john/d>
      <Application, orders>
      <Data_Interface, DC> }
    }
  }
}
\end{verbatim}

This annotation is intended to record where to find the package in its several forms and to alert the system to certain dependencies that require the inclusion of supporting files like \texttt{SUPPLIER.o} above.

So, what then is the relationship between packages and code synthesis? Obviously, a package provides a nice, simple way to construct a library of .o’s that can be used to support a code synthesis situation where user application C code would be compiled and linked with Daytona-generated .o’s taken from the package library. However, packages go further in that a package Cymbal import enables the package functionality to be automatically available through Daytona alone to Cymbal programs as well. In this latter case, all the activity is taking place at the Cymbal level exclusively.
18.8 Generating Adhoc Cymbal

In some situations, it is burdensome to write pre-compiled queries which will be able handle all end-user requests just because such queries would just be too complicated. As an alternative, the application developer can write a program which will take end-user specifications and generate a custom Cymbal query that will handle the user request. This can be quite a flexible and powerful mode of use, where the time it takes to generate and compile the query only takes 10 or 20 seconds and that itself can well be more than paid back by the resulting speed of the Daytona executable.
Generated Ad-Hoc Queries For Answers

Web Interface with Text Generator

Cymbal Queries

Cymbal Query

Tracy

C Code

C Compiler

Daytona Executable

Operating System

Answers
19. Parallelization Foundations

To get the fastest query executions possible in a multi-CPU environment, Cymbal provides a full array of parallelization capabilities. These include forking and optionally exec’ing new processes, waiting for the exit statuses of terminated children, synchronizing processes through semaphores and Cymbal funnels, using pipes to enable parent and child processes to communicate with each other, using UNIX selects to determine which of several processes is ready to communicate, and lastly, sending signals to processes. This chapter ends with a discussion of Distribute_Cmds, a Cymbal program which does a good job of parallelizing a set of tasks among a fixed set of workers.

In the next chapter, Parallelization Made Easy, the simplest possible, most powerful parallelization constructs are introduced; behind the scenes, these highest level constructs are implemented using the basic capabilities described in this chapter, which themselves of course will always be available to the user to support maximally flexible and capable custom parallelization.

19.1 BUNDLES OF TENDRILS

A TENDRIL represents the execution of code in an environment. There are two subclasses of TENDRILS: _process_ and _thread_. Currently, Daytona only supports process-level parallelism, i.e., TENDRIL(_process_) and indeed, _process_ is the default subclass specifier for TENDRIL. A BUNDLE is a set of TENDRILS. The only subclass of BUNDLES that is supported at this time is BUNDLE(_process_), corresponding to a set of TENDRIL(_process_); consequently, _process_ is the default subclass specifier for BUNDLE as well. BUNDLES make it easy to work with multiple TENDRILS as a group: for example, while other BUNDLES are busy, a given BUNDLE of TENDRILS can be waited for as a group to finish and report their exit statuses, they can be signalled as a group, and they can be waited for as a group for the next member TENDRIL to be ready for I/O. Here are the imports for the basic BUNDLE-specific functions:

```
BUNDLE(_process_) FUN:
    new_bundle{
    with_name STR = _default_name_,
    ( 0 ) for _3GL_TEXT = _process_ )

PROC( BUNDLE = _default_process_bundle_ ) Describe
PROC Describe_All_Bundles

STRUCT{ STR(=) .Name, INT .Index, INT .Tendril_Cnt } VBL any_bundle

PRED[ BUNDLE, BUNDLE ] : Eq_Bundle
```

It is important to realize that Daytona has a default BUNDLE(_process_) with user name _default_process_bundle_ and known at the Cymal level as the BUNDLE constant _default_process_bundle_. If all the user does is to create TENDRIL(_process_) without explicitly associating them with any BUNDLE, then all those TENDRIL(_process_) will go by default into _default_process_bundle_. In this event, there need not be any explicit reference to any BUNDLE
in the program because the default behavior will ensure that _default_process_bundle_ is quietly employed behind the scenes as necessary. (Nonetheless, it is good practice to explicitly group TENDRILS into BUNDLES to avoid, for example, inadvertently waiting for miscellaneous/extraneous child processes that are not the TENDRILS under consideration at the moment.)

A new BUNDLE can be created by calling the new_bundle FUNCTION. A new_bundle can be optionally named with any STRING that the user feels is informative. The declaration of the VBL any_bundle above is meant to convey that any BUNDLE can be stimulated to produce its name, (ordinal) index (of creation) and TENDRIL count by suitable use of STRUCT member specifiers as in .my_bundle.Tendril_Cnt.

The Describe PROC is overloaded for at least BUNDLE, TENDRIL, and TICKET_BUNCH. (As detailed later, TICKET_BUNCHES provide semaphore-based synchronization to the language.) In each case, Describe prints out information about the associated object that includes the information provided by the STRUCT member specifiers listed in the import for the corresponding any_<whatever> VBL.

There is also an overloaded Free PROC for BUNDLE, TENDRIL, TICKET_BUNCH, and others but since the system automatically calls it whenever an associated VBL goes out of scope, it is simplest and best to let the system handle this object deallocation as it may and only intervene with explicit calls if absolutely necessary. Free can only be called on a VALCALL of appropriate type. (Recall that a VALCALL is an explicit VBL dereference as in .x.)

Since multiple VBLS can each have the same BUNDLE, TENDRIL, or TICKET_BUNCH (resp.) as their values, each call to Free for such a VBL will reduce the reference count for the associated object by one, with the object only being truly freed when the reference count reaches 0. It is important to remember that any TENDRIL that is part of some BUNDLE is considered to be referenced by that BUNDLE. In order to support the user actually removing a TENDRIL from a BUNDLE, when the user calls Free on a TENDRIL in their Cymbal code, if the only references to the TENDRIL are its BUNDLE and the VBL value being Freed, then both references are removed, resulting in the removal (and freeing) of the TENDRIL from the BUNDLE.

The new_tendril FUNCTION has three variant invocations corresponding to creating clone processes, to spawning new processes/pipelines from external programs, and creating threads. At this time, both clone processes and new-program child processes can be created as TENDRILS. While pipelines can not yet be created using TENDRILS, the new_channel( via _pipe_ ) paradigm provides an alternative shell-based mechanism for creating pipelines. Threads remain for the future.

19.2 TENDRIL Clones

The creation of a clone process corresponds to using the UNIX fork(2) command to create a new process whose initial state is an exact copy of the invoking parent process, with the exception that it knows that it is a child and can therefore begin to follow a different execution path (in the same program) if it so chooses. This contrasts with spawning where a fork(2) is followed by an exec(2) that completely overlays the fork(2) process image of the child with that of a new program that begins execution with the first statement of its main() procedure. Threads of course are different still in that they involve independent execution of code in exactly the same process image as the code which created the thread.

Clones enable Daytona to provide what is called SPMD parallelism, i.e., single-program-multiple-
data. In this paradigm, parallelism is accomplished by partitioning a set of objects into groups and then invoking the exact same program multiple times, with each invocation working on its own portion or section of the set of objects. Typically, as they finish, the work of each of the separate invocations is integrated into the whole by another process (usually the parent) so as to provide the final answer.

Here is the import for the version of the FUN `new_tendril` used to create clone TENDRILS:
TENDRIL(_process_) FUN: new_tendril(
  with_name STR = _default_name_,
  for_bundle BUNDLE = _default_process_bundle_,
  (0) with_whole_msgs = ?, // _present_
  with_msg_terminator STR = "\n",
  with_downlink _3GL_TEXT = _none_,
  executing DO
)

STRUCT{ STR(=) .Name, INT .Index, BUNDLE .Bundle,
  STRUCT{ INT .Pid } .Sys_Id,
  STRUCT{ INT .Kind, INT .Value } .Status } VBL any_tendril

(As shown in sys.env.cy, new_tendril is actually overloaded but the above import will encourage and support proper use nonetheless.) It is the executing keyword that tells the system to create a clone TENDRIL(_process_). The argument to executing is a Cymbal do-group, which is just a sequence of procedural statements enclosed by braces. A new_tendril can be optionally named with any STRING that the user feels is informative; it can also be included in any BUNDLE that the user explicitly identifies. As will be seen, with_downlink enables the user to specify whether the parent converses with the child or not, and if so, how, meaning whether it just sends to or just receives information from the child or both.

The any_tendril import above informs the user that information can be gotten about any TENDRIL simply by referring to the desired member of the indicated STRUCTURE, as in .tendril.Sys_Id.Pid being equal to the pid of the associated TENDRIL .tendril.

Incidentally, any Cymbal program can change its execution priority by using the PROC Make_Nicer_By:

PROC( INT .nice_increment = 10 ) Make_Nicer_By

Calling this PROC causes the nice(2) system function to be called with the indicated increment to the priority. Positive increments (counter-intuitively) decrease the process’s priority which will allow more important processes to run instead, in the event that there is any contention for CPU resources. Negative increments, making processes more important, are only allowed to processes running with setuid root. The appropriate UNIX man page provides a window into the excessively complex and confusing world of UNIX process priorities.

19.3 Dividing Up The Work With from_section

The ability to easily partition the workload with the from_section concept is what makes clone processes easy to specify and use. The goal is to divide up a base collection of objects into roughly equal sections and assign each clone its own section to work on: for each object in its section, each clone processes its object using exactly the same logic as any other clone, which may be arbitrarily complex. This implies that there is a good chance that the load will be fairly much balanced and thus that all clones will finish processing at the same time: this is the ideal, of course, because necessarily, the query can’t finish up until the last clone finishes. The next two subsections discuss how from_section can be used to reference portions of BOXes and RECORD_CLASSES.
19.3.1 BOXes Using from_section

The *from_section* keyword can be used to partition work on boxes (section.1.Q):

```c
set [ .sect, .tot_sects ] = read( from _cmd_line_ but_if_absent[1,1] );
set .box_1 = [ 125 -> 175 ];
do Display each [ .nbr, .part ]
each_time(
    .nbr Is_In .box_1 from_section[ .sect, .tot_sects ]
    and there_is_a PART where(
        Number = .nbr
        and Name = .part )
);
```

Here *from_section* is a keyword for *Is_In*; the box is divided up into .tot_sects and the *Display* works with precisely section .sect of the .tot_sects sections available. This particular query may not seem to useful at the moment because there is no apparent mechanism for invoking the query more than once or for integrating the results of multiple integrations -- such mechanisms of course are the main topic of this chapter and will be pursued shortly. Nonetheless, queries of this form can be of value in debugging: to watch the behavior of the program on just a few cases, use Tracy’s -TC, set .sect to be 1, and set .tot_sects to be large.

The *from_section* keyword argument can also be used with the *Is_Something_Where*, *Is_The_Next_Where*, and path PRED boxes. It cannot be used in box assertions which are ground (i.e., which are tests and hence do not generate values). If it turns out that there are more sections than elements of the box, then some sections will contain one element and the others zero.

19.3.2 Descriptions Using from_section

As mentioned above, Daytona’s approach to parallelization depends in part on dividing up the work of a query into (roughly) equal portions and then cloning processes (and eventually, threads), each to do their share of the work. This approach is productive because the nested-loops way that Daytona processes logic assertions is easily partitioned among processes by simply by partitioning the initial sub-assertion generator of values.

As one example, the *from_section* keyword can be used to cause a Cymbal description’s generation of values to be partitioned into one of a specified number of sections, as in (section.5.Q):
set [.sect, .tot_sects] = read( from _cmd_line_ but_if_absent[1,1] );

do Display each [.supplier]
each_time(
    there_is_a SUPPLIER from_section[ .sect, .tot_sects ]
    where(
        Name = .supplier
        and City Matches "^[CP]"
        and Number = .supp_nbr )

    and there_is_an ORDER where(
        Supp_Nbr = .supp_nbr
        and Part_Nbr = .part_nbr )

    and there_is_a PART where(
        Number = .part_nbr
        and Color = "red" )
    );

If .sect = 2 and .tot_sects = 5, then this Display will only work with those SUPPLIERs from the second of 5 roughly equal SUPPLIER sections. Since the SUPPLIER description is the first generator in the above assertion, it drives the processing of the balance of the assertion because for each SUPPLIER that satisfies the conditions, the system will continue to investigate ORDERS and then PARTS in its search for answers. Consequently, by partitioning SUPPLIER with from_section, the workload of the query as a whole has been correspondingly partitioned in what amount to be similar workloads.

If the description is completely ground at its point in the query (i.e., it is just a test and does not generate values), then from_section is not allowed. There is only one other case where from_section is not allowed in Cymbal descriptions, namely, when the description is using a keyed access to a non-horizontally-partitioned table. This would correspond, for example, to wanting to divide up into sections all the EMPLOYEE records whose subjects own blue cars (secondary index search) or all the EMPLOYEE records whose Zip Codes started with 850 (initial substring search) or all the EMPLOYEE records whose subjects live in the State of Virginia (cluster B-tree search). (On the other hand, obviously, there is nothing to be gained from from_section when searching on a Unique KEY.) Anyway, while these are reasonable scenarios, they would place unaccustomed demands on the btree indices and consequently have not yet been implemented. An effective alternative would be to load a BOX up with the Unique KEYS for all the records of interest as identified by a separate (indexed) search, and then partition that with from_section and search the RECORD_CLASS (again) with keys from the appropriate section of the BOX. This will be adequately efficient if the work done processing each element of the BOX is large in comparison to the work needed to make the BOX.

Otherwise, other uses of from_section are permissible -- and robust: for example, if there should happen to be more sections than there are records, then the sections will have exactly one record and others zero.
Horizontal Partitioning And Descriptions Using from_section

There are some subtleties to be aware of when using from_section with horizontally partitioned tables. In this case, the system begins by computing at runtime the total number of bins which it will visit on behalf of the associated there_isa. Since the system will use binary search on the bin keys themselves (if possible) in order to determine the fewest number of bins to visit, the total-number-bins-to-visit can well be much less than the total number of bins that exist. On the other hand, if the query does not provide the information Daytona needs to use binary search on the bin key, then the entire range of bins will be partitioned into the various sections, whether they ultimately contain any information useful to the query or not.

Take the ORDERA table, for example (section.2.Q). It is horizontally partitioned according to the TUPLE [ Region, Category ]. If the there_isa under consideration is:

```
there_is_a ORDERA from_section[ .sect_nbr, 2 ] where(
    Region = 3
    and Category = .x
)
```

where the defining occurrence for .x is in the Category satisfaction claim, then at runtime, the total-number-of-bins will be determined to be 7 (check out ordera_flv in the $DS_DIR/EXAMPLES/usr/orders). This is because [ 3 ] serves as a key into the sorted list of 25 ORDERA bins and the binary search reveals that there are 7 such bins with Region = 3.

Likewise, for the following (with defining .x occurrence)

```
there_is_a ORDERA from_section[ .sect_nbr, 2 ] where(
    Region <= 3
    and Category = .x
)
```

the runtime identification algorithm will determine that there are exactly 11 bins to visit, i.e., the ones with Region <= 3.

On the other hand, when the Category alone is specified, then Daytona cannot use the bin key (because Category does not constitute an initial subsequence of the horizontal partitioning FIELD KEY) and so, the sections will be determined so as to include all of the bins, not just the ones that have Category = "A".

```
there_is_a ORDERA from_section[ .sect_nbr, 2 ] where(
    Region = .x
    and Category = "A"
)
```

(It is assumed here that this there_isa contains the defining occurrence for x.)

Continuing on then, the rule is that if the total number of sections is less than or equal to total-number-bins-to-visit, then each section will consist of a roughly equal number of bins, which may be accessed either sequentially or by btree index.
On the other hand, suppose the total number of sections is strictly greater than total-number-bins-to-visit. Then if sequential access is used, each section will consist of either exactly one bin or a suitably proportionate fraction of one bin. If btree-indexed access is used then the first total-number-bins-to-visit sections each consists of exactly one bin and the remainder of the sections have no bins to work with at all.

When the goal is to do parallel sequential access on a horizontally partitioned RECORD_CLASS, then there is a very useful idiom which ensures well-balanced sections and hence, optimal parallelism. This idiom takes advantage of the fact that if from_section is applied to a single bin in a sequential search situation, then that bin must be divided up into equal sections of records -- sections that are as equal as possible. Contrast

```plaintext
there_is_a ORDERA from_section[ .sect_nbr, 2 ] where
  Region <= 3
  and Category = "A" | = "B"
```

with

```plaintext
.region Is_In [ 1 -> 3 ]
and .cat = "A" | = "B"
and there_is_a ORDERA from_section[ .sect_nbr, 2 ] where
  Region = .region
  and Category = .cat
```

In the second formulation, total-number-bins-to-visit for the there_is_a is at most one (at a time). This is because [ Region, Category ] is a unique key for the ORDERA bins, meaning at most one bin has any given pair of values for the two FIELDS. Note that is OK for there not to be some bin with given values for region and cat.

Consequently, for the second formulation, each of the two sections must consist of half of each of the bins satisfying the Region/Category constraint. The bins can all be of (perhaps radically) different sizes and yet the division of the work will be essentially perfect. (The optimality of this strategy relies on [ Region, Category ] being a unique key.) Contrast this situation with the first alternative where the first section will consist of all of some number of bins and the second section of the rest of the bins. If all the data was in the first section’s bins, then the second clone would have nothing to do! So, clearly, the paradigm of dividing each bin into equal pieces for each section is superior. In short, it is better for each section to have some of all of the bins than all of some of the bins.

See clone.a.Q for some examples illustrating these principles.

### 19.3.3 from_section in general

The parallelization will be most efficient if each section results in an equal amount of work. The idea is to avoid the case where the processing for all but one of the sections finishes and the survivor continues for some appreciable amount of time afterwards. The variance in finishing times can be controlled by specifying an appropriate number of sections relative to the number of CPUs
available. Given a superior process scheduler in the operating system (such as that for Solaris), it has been observed in one case that the total amount of time needed to process some queries can be essentially constant when the number of sections varies from being equal to the number of processors to being equal to twice that number. The reason may be that since the amount of work to be done is constant, if the context switching overhead is low, then the fact that there are more processes is equally compensated for by the fact that each has proportionately less work to do. In the experiment run, there also was no observed rise in I/O contention with the larger number of processes. However, for other queries, an increased contention for a variety of shared resources in the operating system has been observed with an increase in the number of clones. Indeed, a point of diminishing returns will be reached as soon as the disks holding the data are 100% busy. So, there are no hard and fast rules -- variance in clone finishing time (and total elapsed time) can decrease with increasing numbers of clones but after a point, performance can degrade significantly if the clones start driving up system time due to fighting each other for shared resources (like OS mutexes and files). Only experimentation will identify what is best in general.

19.4 Dividing Up Work By Clone Number

Another not necessarily efficient but sometimes effective method of dividing up work is to have each clone sequentially scan exactly the same records except that each clone only does work on those records that fall into its equivalence class. One way to define such equivalence classes would be to take some INT FIELD that assumes a full range of values and partition that range into equivalence classes modulo the clone number. In this case, there is probably no need for the children to send anything back to the parent since each is certainly working on records that the others are not; consequently, each clone could just be able to append its contribution to the output without further ado or integration.

19.5 A Simple Clone Query

One of the simplest kinds of parallel queries is the kind where each clone searches its own portion of the search space and then just writes out its answers to stdout. clone.4b.Q uses new_tendril and next_waited_for_tendril to accomplish this end.
set [ .tot_sects ] = read( from _cmd_line_ but_if_absent[ 5 ] );
_Init_Time_Store
do Configure( with_whole_msgs );
for_each_time .sect Is_In [ 1-> .tot_sects ]
{
    set ? = new_tendril(
        executing{
            for_each_time [ .supplier ]
                is_such_that(
                    there_is_a SUPPLIER from_section[ .sect, .tot_sects ]
                    where(
                        Name = .supplier
                        and City Matches "^[CP]"
                        and Number = .supp_nbr )
                and somehow ( 
                    there_is_an ORDER where(
                        Supp_Nbr = .supp_nbr
                        and Part_Nbr = .part_nbr )
                    and there_is_a PART where(
                        Number = .part_nbr
                        and Color = "red" )
                )
            ){ do Write_Line( .supplier );
                }
        }
    )
    while( next_waited_for_tendril() != _null_tendril_ ){}
_Print_Time_Delta

In this case, each of TENDRILS executes a single for_each_time statement and then quits. This is important: once the TENDRIL’s execution reaches the end of the executing do-group, the TENDRIL exits. Of course, it is perfectly permissible for there to be calls to fpps in this do-group. Those fpps can be helpers or even tasks in the current program text. The clone process has no trouble finding them because it has exactly the same program text as its parent. However, please note that having the same program text is not the same as sharing the same program text. Since each clone has a copy of the parent’s process image, it begins its execution with its Cymbal variables having the same values as the parents did at the instant of the new_tendril call but then since they are copies, when any clone or parent modifies their variables, no other clone or parent sees those modifications because they have independent copies of their own.

19.5.1 Synchronizing Output To _stdout_ With Messages

On the other hand, this doesn’t mean that they don’t share anything. The fork(2) man page describes in detail what the clone inherits from the parent and there are some entities that are shared, if they exist. In particular, a Daytona clone automatically shares the same open files that its parent did:
in particular, the clones and the parent all share the same stdout. As with any shared resource, care must be taken to prevent the sharing entities from colliding in some way while trying to access and use that shared resource. In terms of output, in general, this means preventing the loss or corruption of any clone’s output, in which happy case the output behavior of the clones is considered to be synchronized.

In Daytona, a clone’s output is regarded as being uncorrupted with no losses if, when seen as a sequence of atomic messages, all messages output by clone appear intact (verbatim as they left the clone) in the shared output stream. The atomic messages may appear interspersed with the atomic messages of other clones but any given clone’s atomic messages will appear intact as a subsequence of the messages in the shared output stream, appearing in the order in which they were written.

The simplest Daytona message is just a line. In general, a Daytona message is any sequence of characters terminated by another sequence of characters. The message terminator can be "\n", "\n\n", ",", ",", or just any specified sequence of characters. Every message that a clone produces has to end with the same terminator, of course.

In Cymbal, when clones or spawn are writing output, the user must always define the messages that are being used as a first step towards guaranteeing output synchronization. When writing to stdout, since that CHANNEL is implicitly open by default, the Configure PROC provides the only way to specify the definition of stdout messages and to require that the associated system buffer to be flushed/written in terms of an integral number of these messages.

PROC Configure(
    this CHAN = _stdout_,
    with_whole_msgs = ∼,
    with_msg_terminator STR = "\n",
    with_user_sync = ∼,
    with_bufsize INT = -1 )

So, to configure _stdout_ for whole messages, the user just writes:

    do Configure( with_whole_msgs );

To handle very large messages, the with_bufsize argument is available. Indeed, Daytona will misbehave with no clear error message if a (user) message occurs that is bigger than the buffer size. The with_bufsize argument is truncated to the nearest 1K. For _stdout_, the default bufsize is whatever PIPE_BUF is on the machine, which by the way, is the least it can be. Of course, the size of each message must be less than the buffer size and the buffer will hold as many messages as it can.

As will be seen, there are other ways for clones to write and, in those cases, the new_tendril and new_channel functions provide a largely identical way to define messages in those contexts. For the moment, just observe for this query clone.4b.Q, that the Configure call configures _stdout_ to be flushed at whole message boundaries where messages consist of single lines. Consequently, if the actual buffer flush happens atomically (which is always the case for stdout but must be otherwise properly arranged for other CHANS), then no message can be corrupted by having some other clone’s text interpolated within that message.

Therefore, once stdout has been properly Configured, the user can issue Write calls without further
ADO, confident that Daytona will keep the various clone’s output synchronized. Caveat: however, any bufsize larger than PIPE_BUF and perhaps other parts of that Configuring will be undone if the user uses the shell to pipe the output of the query to another program. So, don’t do that; just redirect the output to a file as in .R >dummi. There is nothing wrong with the shell here -- the hard fact is that the data has been told to go through a pipe and so, there are just necessarily no atomic buffer writes if the buffer size is larger than PIPE_BUF.

By the way, the Configure call must be executed before the clones are created so that they will all inherit the same configuration of stdout.

As an obscure technical note, be aware that once Configure has been called on _stdout_, _stdout_ will no longer be automatically flushed past a line-boundary prior to reading _stdin_. About the only situation where that could have an impact is when writing interactive scripts which issue prompts as follows:

do Write("Enter name: ");

Note the absence of a new-line, the idea being that a subsequent Read from _stdin_ will expect to get input from that same line on the user’s screen. After calling Configure on _stdout_, this will no longer happen: _stdout_ will only flush automatically on whole message boundaries. It is easy to work around this new artifact:

flushing do Write("Enter name: ");

19.6 Waiting For Child Processes To Report Exit Statuses

next_waited_for_tendril is used to wait for (and collect) the exit status information from the next child process available to report.

import gbg_not_here:

overloaded otherwise_ok TENDRIL FUN(
   ( 0->1 ) namely TENDRIL(_process_),
   ( 0->1 ) for_bundle BUNDLE(_process_),
   with_patience INT = _wait_on_block_
) next_waited_for_tendril

The waiting is confined to a specified BUNDLE when the for_bundle argument is used and even to a specified TENDRIL when the namely argument is used. If no such TENDRIL exists, then the function returns _null_tendril_. If there are any downlinks to the child process, those are closed on the theory that communication with the dead is unreliable. Consequently, all parent communication with the child process must be done while the child is still alive (i.e., prior to its exit). By default, if there are still some candidate TENDRILS running, the function blocks waiting for a TENDRIL to finish so as to be able to report on it. As with CHANNELS, this behavior can be modified to be nonblocking by using the _fail_on_block_ argument to the with_patience keyword or else it can be modified to wait for a maximum number of seconds by using an INT argument to the with_patience keyword. Daytona makes the current number of waited-for child processes available in the global (system) INT nbr_kids_waited_for.

In the case of the above query, the programmer is obviously disinterested in the actual exit statuses of the clone processes: the only intent is to gather that information until all clones have
finished. This alone has utility. For example, when using time(1) or the combination of _Init_Time_Store and _Print_Time_Delta to measure the combined execution time of the parent and its clones, it is mandatory that the parent wait(2) for the clones or else their time information will not be gathered. Here is some sample timing output provided by _Print_Time_Delta for a parallelized program:

```
========== timing info @ Thu Apr 15 09:04:13 EDT 2000 =============
elapsed time = 1.718954s

own_user_time = .12s
own_sys_time = .76s
own_user_sys_time = .88s

kids_user_time = 1.01s
kids_sys_time = 3.64s
kids_user_sys_time = 4.65s

cpu fraction of elapsed time = 3.217
```

Note that the child processes are doing most of the processing here and that there is substantial parallelism with the total CPU time exceeding 3.2 times the elapsed or wallclock time.

Another advantage of waiting for one’s clones is that it guarantees that the parent will not exit before the children do. This guarantees that any shared resources that the clones and parent use that the parent owns and manages will not be released/destroyed (with disastrous consequences) until all the clones using them are finished. An example of a shared resource is semaphores or as they are called below, TICKET_BUNCHES.

The next example (clone.4i4.IQ) illustrates, among other things, an appropriately deeper interest in the status of the children.
do Configure( with_whole_msgs with_msg_terminator 10 '"#" + "$n$" );
for_each_time .cidx Is_In [ 1-> 5 ]
{
    set ? = new_tendril( executing{ do Process( .cidx ) } );
}
// recommended minimum way to express interest in clone welfare
loop {
    set ? = next_waited_for_tendril()
    otherwise_switch { case(=_no_such_kid_){ break } else{ do Exit( 4 ) } }
}
define PROC( INT .cidx ) Process
{
    for_each_time there_isa ORDER from_section[ .cidx, .nt ]
    {
        where this_isa ORDER do Describe;
        do Write_Line( 10 '"#" );
    }
}

To understand the waiting strategy here, first note the following possible exit statuses for
next_waited_for_tendril:

    define CLASS NEXT_WAITED_FOR_TENDRIL_CALL_STATUS
        with_symbols { _ok_kid_, _no_such_kid_,
        _failed_kid_, _killed_kid_, _stopped_kid_,
        _would_block_, _timed_out_, _interrupted_ } 

In the program above, if the exit status of next_waited_for_tendril is _ok_kid_, then the loop
continues. Otherwise, if there are no more clones to wait for, then the status is _no_such_kid_ and the
loop is broken out of. (In general, _no_such_kid_ results when the exit status of one or more children is
being sought and for whatever reason, no such exit status is available: e.g., a specific kid might have
already been waited for or was never a child of this process anyway or all kid exit statuses are being
ignored.) The only other possible exit statuses for this use of next_waited_for_tendril are all error
conditions, which are handled in the customary way provided by the use of otherwise, which would
consist of an informative message followed by exiting the program.

FYI, _would_block_ is the status associated when using with_patience _fail_on_block_ and when there
is a clone process of interest which will return an exit status but is not yet ready to do so. _timed_out_
occurs when the with_patience timer in seconds has expired before any appropriate TENDRIL could
finish. _interrupted_ happens when the parent process receives an interrupt signal while waiting.

Here is sample code that shows how to wait zealously for every particle of information about the
exit statuses of one’s clones:
loop { local: TENDRIL .kid
set .kid = next_waited_for_tendril( for_bundle .wkgroup )
otherwise_switch {
    case( = _no_such_kid_ ){ break; }
    case( = _failed_kid_ |= _killed_kid_ |= _stopped_kid_ ){ }
    else{ do Exit( 4 ); }
}
switch_on( .kid.Status.Kind )
{
    case( = _exited_ ){
        when( .kid.Status.Value = 0 )
        {
            do Assimilate_Results_Of_Section( .kid );
        }
        else
        {
            do Exclaim_Words(
                "error: child process failed to complete, returning code",
                .kid.Status.Value );
        }
    }
    case( = _killed_ ){
        do Exclaim_Words(
            "error: a child process has been killed by signal",
            .kid.Status.Value );
    }
    case( = _stopped_ ){
        do Exclaim_Words(
            "error: a child process has been stopped by signal",
            .kid.Status.Value );
    }
}

The above code for the parent process waits for each of its clones to finish. A clone that exits under its own power has .kid.Status.Kind equal to _exited_, in which case, the clone's own chosen exit status is contained in .kid.Status.Value. Here, the parent calls Assimilate_Results_Of_Section to process the results of the current clone to finish -- if that clone exited with a status indicating success or normal completion; otherwise, appropriate action can be taken by the parent on the basis of the clone's announced error status. Note that a .kid.Status.Kind of _exited_ corresponds to two possible values for next_waited_for_tendril call status, namely, _ok_kid_ and _failed_kid. _ok_kid_ means that the child process exited with a 0 exit status, thus indicating successful execution; _failed_kid_ means the opposite. If an exogenous agent killed or stopped the clone with a signal, then that is indicated by the appropriate value of _killed_ or _stopped_ for .kid.Status.Kind at which point the associated signal number is conveyed by .kid.Status.Value.
19.7 Clones Writing To Shared File CHAN

When a parallelization goal is for the clones to contribute their output to a shared CHANNEL that is not _stdout_, then additional strategies become available and/or necessary to ensure that the clones do not interfere with each other when writing to the same output CHANNEL. The circumstances conform to the following taxonomy. Suppose that the smallest atomic unit of output that a clone can produce has several lines and that the buffer size for the output CHAN is larger than the largest atomic unit. Suppose further that the message can be characterized by terminating with a unique string _termin_. Then by using the _with_whole_msgs_ and _with_msg_terminator _termin_ keyword-arguments in the associated _new_channel_ call, the corresponding buffers will be flushed only in terms of whole messages. (This is necessary because otherwise if a buffer flush wrote out a buffer with a partial message at the end of the buffer, then some other clone could follow that with its own buffer flush, thus splitting up the message of the first clone that wrote.) Lastly, if there is no such easy characterization of messages (unlikely), then at a substantial cost to I/O efficiency due to lack of buffering, _Flush_ can be called by the clones to output each atomic (multi-line) unit.

Likewise, as a special case, suppose that all atomic output consists of single lines of size less than the buffer size. Then message integrity can be preserved by using _with_whole_msgs_ while relying on the default _with_msg_terminator "\n".

In any case, to ensure correct synchronization with message integrity and no lost or corrupted data, use the msg-based keywords to _new_channel_ along with to one of the following three strategies.

1. Call _new_channel_ with the _with_whole_msgs_ keyword argument and with _with_mode_ _append_, _append_update_ or _clean_slate_append_ (clone.5a2.Q).

2. Call _new_channel_ with the _with_whole_msgs_ keyword argument and with _via_pipe_ (clone.6p2.Q). (In this case, the messages cannot be larger than the size PIPE_BUF of the pipe, which varies from one UNIX OS to another. For example, it is 5120 bytes for Sun Solaris and 10240 for SGI Irix.)

3. To handle even the most general case where the clones are also seeking around and reading in the common file they are writing to, use the _with_user_sync_ keyword argument to _new_channel_ and use a TICKET_BUNCH/semaphore (cf., clone.5wt4.Q) as described towards the end of this chapter.

19.8 Parents Talking To Children

Cymbal supports parents and children exchanging messages over pipes. In the next two example queries, the parent has a list of work to be done, i.e. a list of part numbers whose colors need to be looked up. These part numbers are sent down to a clone via a pipe, one by one, each to a line. The clone then reports the color (with its part number) back through _stdout_.

Somehow, the clone has to know when the parent is finished sending it work to do. There are two ways to arrange for this to happen when using messages over pipes.

1. The clone is expecting exactly one message. After it knows that it has gotten the message in its entirety by reading the message terminator, it does what it needs to and then exits.
2. The clone is expecting several messages: it reads a message, acts appropriately, and then repeats until it gets _instant_eoc_ on the _parent_ channel.

19.8.1 Child Is Expecting Exactly One Parental Message

In the next example, clone.4c.Q, the clone is expecting exactly one multi-line message terminated by five sharp signs on a line.
local: INT .nbr_clones = 3
TENDRIL ARRAY[ INT ] .ta
do Configure( with_whole_msgs );
for_each_time .cidx Is_In [ 1 -> .nbr_clones ]
{
    set .ta[ .cidx ]
        = new_tendril( with_name (STR) .cidx
            with_downlink _send_
            executing{ do Process; } )
};
set .pno_box = [ 101 -> 109 ];
for_each_time .cidx Is_In [ 1 -> .nbr_clones ]
{
    for_each_time .pno Is_In .pno_box from_section[ .cidx, .nbr_clones ]
    {
        to .ta[ .cidx ] do Write_Line( .pno );
    }
    flushing to .ta[ .cidx ] do Write_Line( "#####" );
}
_Wait_For_Tendrils
define PROC Process
{
    local: STR .line
    loop {
        set[ .line ] = read_line( from _parent_ )
        otherwise{ with_msg "child failed to Read line" do Exit( 2 ); };
        when( .line = "#####" ) break;
        set .pno = (INT) .line;
        for_each_time [ .color ] is_such_that(
            there_isa PART where( Number = .pno and Color = .color )
        )
        {
            do Write_Words( "Part", .pno, "has color", .color );
        }
    }
}

Note that the new_tendril function’s keyword-argument pair with_downlink _send_ is what instructs Daytona to support messages going from the parent to any specified child. In this example, for each clone, the parent sends down its entire worklist for that clone by writing the associated lines plus message terminator and then flushing the communication channel to that TENDRIL. The syntax employed here does constitute something of an abuse of notation, albeit a convenient one: to be precise, what is being written to (or flushed) is the CHANNEL that links the parent to the clone TENDRIL, not the TENDRIL itself. Likewise, the clone reads lines containing part numbers from the TENDRIL’s
parent CHANNEL until the message terminator is detected. \_parent\_ is a symbolic constant denoting the TENDRIL’s parent CHANNEL that is directed to this clone.

This query illustrates using \textit{from\_section} in the parent to divide up the work in the children; a previous query used \textit{from\_section} in the code for the clones to determine the workload for a clone.

Note the use of the \texttt{ds\_m4} macro \texttt{\_Wait\_For\_Tendrils} to do the standard, recommended waiting for child TENDRILS. As the definition in $\texttt{DS\_DIR/sys.macros.m}$ shows, \texttt{\_Wait\_For\_Tendrils} takes an optional argument consisting of the BUNDLE to be waited on:

\begin{verbatim}
\texttt{\_define_(@(\texttt{\_Wait\_For\_Tendrils})@, @<
loop {
  local: TENDRIL .kid
  set .kid = next_waited_for_tendril( for\_bundle @ifelse_($1,\_null\_bundle_,$1) )
  otherwise_switch { case(=\_no\_such\_kid_){ break }
   else{ with\_msg concat(["for bundle = ",.kid.Bundle.Name," for kid = ",.kid.Name,
    ", error code = ",.kid.Status.Value]) do Exit( 91 ) } });
}>@
\end{verbatim}

If there is no first argument, then all child processes are waited on: this includes all TENDRILS in all BUNDLES and it also includes any other child processes such as the shell associated with \texttt{CHAN(popkorn)}, if that is being used. Since \texttt{CHAN(popkorn)} may not be closed down until program exit, an indefinitely long wait will typically ensue in the event that \texttt{CHAN(popkorn)} is being used. That is why the fail-safe thing to do is to always provide a \texttt{for\_bundle} argument to \texttt{next\_waited\_for\_tendril}, even if it is just \texttt{\_default\_process\_bundle}. In fact, the system will exit with an error message if popkorn is running and there is an attempt to do a non-specific wait.

Note that an associative array can be used to hold the TENDRILS (as array element values, not indices): this prevents the accidental array bounds exceptions that can occur with conventional arrays.

Also, take note that it is very important for the parent to flush the end-of-message token ####. Otherwise, the IO buffer in the parent’s memory that is accumulating this short message (shorter than the length of the buffer) will not be prompted to physically write the buffer contents outside of the parent’s process image into the pipe, thus making it available to the clone to read.

Conversely, in general, for a clone writing its results out, it is important to make sure that there is a final \texttt{Flush} on the output CHANNEL. \textit{Fortunately, please note that Daytona takes care of that for the user automatically by quietly calling Close (hence, Flush) on all open CHANNELS when the program exits. So, no explicit Flush on the part of the clone is necessary.}

In general, explicit Flushing is needed in only one situation: when synchronicity is desired, i.e., when the program logic requires that the message be sent immediately, i.e., by the time the next statement by the sender comes up for execution. Otherwise, for efficiency’s sake, it is best to let the system flush buffers automatically when full, subject to the \texttt{with\_whole\_msgs} preservation when in effect.
19.8.2 Child Is Expecting Several Parental Messages

The second strategy for parent-clone communication is for the parent to write a sequence of messages to the clone and then close its end of the pipe when it has sent the last message. Subsequently, when the clone has read the last written byte from the pipe, the next read will result in an end-of-file, i.e., \_\_instant\_eoc\_. The clone may then react accordingly. This strategy is illustrated by clone.4d.Q:

```c
local: INT .nbr_clones = 3
TENDRIL ARRAY[ INT ] .ta
do Configure( with_whole_msgs );
for_each_time .cidx Is_In [ 1 -> .nbr_clones ]
{
    set .ta[ .cidx ] =
    new_tendril( with_downlink _send_ executing{ do Process; } );
}
set .pno_box = [ 101 -> 109 ];
for_each_time .cidx Is_In [ 1-> .nbr_clones ]
{
    for_each_time .pno Is_In .pno_box from_section[ .cidx, .nbr_clones ]
    {
        to .ta[ .cidx ] do Write_Line( .pno );
    }
    do Close( .ta[ .cidx ] ); // mandatory
}
_Wait_For_Tendrils
define PROC Process
{
    local: STR .line
    loop {
        set[ .line ] = read_line( from _parent_ )
        otherwise_switch {
            case( = _instant_eoc_ ){ break; }
            else { with_msg "child failed to Read line" do Exit( 2 ); }
        }
        set .pno = (INT) .line ;
        for_each_time [ .color ] is_such_that(
            there_isa PART where( Number = .pno and Color = .color )
        ){
            do Write_Words( "Part", .pno, "has color", .color );
        }
    }
}
```

Note the explicit `Close` of the channel to the TENDRIL `.ta[ .cidx ]`: this must be done because otherwise the clones will not receive \_\_instant\_eoc\_ on their end of the pipe; in that case, the clones will
sleep indefinitely trying to read line from their parent as their parent is waiting for them to exit. (Note the abuse of notation here in that while the argument to Close is a TENDRIL, what is actually being closed is not the TENDRIL but rather the I/O CHANNEL(S) that belong to the TENDRIL -- since only I/O CHANNELS can be Closed in actuality.)

Since there is no contention, no sharing of resources when parents send messages to children over their several pipes, there may not seem to be a need here to require whole-message-flushing or to define the message terminator for the associated new_tendril calls. However, as will be explained later, while there is no danger of message corruption here, there is a possibility for inefficient execution that can be removed by inserting the appropriate message-oriented keywords in the new_tendril call.

19.8.3 Children Talking Back To Parents Once

When multiple children want to send information back to their parent, this can occur either in one transmission per child or in a time-sharing arrangement where the parent alternately accepts information from any of its children ready to transmit. The first alternative is illustrated in clone.2b.Q:
local:

    TENDRIL ARRAY[ 3 ] .ta
    LIST[ INT ] .pno_list

set .pno_list = [ 101 -> 109 ];
for_each_time .cidx Is_In [ 1->3 ]
{
    set .ta[ .cidx ]
    = new_tendril(
        with_downlink _receive_
        executing{ do Process( .cidx ); }
    );
}
for_each_time .cidx Is_In [ 1->3 ]
{
    for_each_time [ .pno, .color ]
    is_such_that( [ .pno, .color ] = tokens( from .ta[ .cidx ] upto " \n" ) )
    {
        do Write_Words( "Part", .pno, "has color", .color );
    }
}
_wait_for_tendrils

define PROC( INT .cidx ) Process
{
    for_each_time [ .pno, .color ]
    is_such_that( there_isa PART where(
        Number Is_In .pno_list from_section[ .cidx, 3 ]
        and Number = .pno
        and Color = .color )
    ){
        to _parent_ do Write_Words( .pno, .color );
    }
}

Note that the argument to the with_downlink keyword to the new_tendril call is _receive_ indicating that the parent expects to receive data from each of its children.

Each child knows what work to do by virtue of the from_section argument to the PART table being searched in the Process PROC. This is necessarily effective because each clone starts off with a copy of (essentially) the same process image, including program text and stack. Consequently, the pno_list that was defined in the parent is automatically known in the child. This illustrates a convenient and powerful paradigm for transmitting a potentially large amount of information to the child at start-up: simply create a (large) box or associative array in the parent and then create the new tendrils that will automatically be able to access that information in the usual Cymbal fashion illustrated here. Indeed, typically, each clone will work on its section of that initial box.
After having launched its child clones, the parent in this query visits each one of them in turn to read back the result of their work. Note the attractive use of tokens to accomplish this task: using the same terminological abuse as before when calling a TENDRIL’s CHAN(_pipe_) by the TENDRIL’s name, tokens makes use of this already open CHAN just by using a .ta[ .cidx ] argument to the from keyword.

This query is somewhat inefficient because the parent seeks to get the answers from its children in the order in which they were cloned, which most certainly is not necessarily the order in which they finish. The next section shows how to avoid this inefficiency.

19.8.4 Clones Talking Back To Parents Constantly Helped By next_io_ready_tendril

If clones will be living a relatively long time and during that period will be intermittently sending messages back to their parent, then the parent will benefit from the next_io_ready_tendril function which enables it to determine which clone wishes to speak to the parent at any given time -- and thus which clone to read from next. Query clone.7d.Q below has the same flavor as clone.2b.Q with two exceptions. First, it uses next_io_ready_tendril to determine which child to listen to next. Secondly, it is a textbook example of how to do an efficient group-by query using these low-level (but powerful) parallelization primitives. This query begins with each clone scanning its portion of the ORDER record class and computing for each SUPPLIER Number encountered, the total count of orders from that SUPPLIER as well as the total Quantity ordered. Then the parent reads back (as notified by next_io_ready_tendril) the statistics computed by each clone and integrates them into the final summaries required, taking into account that the same supplier can (and does) appear in the data scanned by more than one clone.
local: INT .tot_sects = 5
TUPLE[ INT .tot_ordered, INT .tot_qty ] .supp_stats[ INT(_short_) .snbr ]
 = { ? => [ 0, 0 ] }
set .bndl = new_bundle();
for_each_time .cidx Is_In [ 1-> .tot_sects ]
{
    set ? = new_tendril( for_bundle .bndl
        with_downlink _receive_
        executing{ do Process( .cidx ); } );
}
loop {
    set .io_ready_tendril = next_io_ready_tendril( for_bundle .bndl )
    otherwise do Exit( 3 );
    when( .io_ready_tendril = _null_tendril_ ) break;
    for_each_time [ INT(_short_) .snbr, INT .sub_cnt, INT .subtot_qty ]
    is_such_that( [ .snbr, .sub_cnt, .subtot_qty ]
        = tokens( from .io_ready_tendril upto " \n" )
    )
    {
        set .supp_stats[ .snbr ] = [ $#1+.sub_cnt, $#2+.subtot_qty ]
    }
}_Wait_For_Tendrils(.bndl)
for_each_time [ .snbr, .tot_cnt, .tot_qty ]
is_such_that( [ .snbr, .tot_cnt, .tot_qty ] Is_The_Next_Where(
    .supp_stats[ .snbr ] = [ .tot_cnt, .tot_qty ] ) in_lexico_order
){
    do Write_Words( .snbr, .tot_cnt, .tot_qty, ((FLT).tot_qty)/.tot_cnt );
}
define PROC( INT .cidx ) Process
{
    for_each_time [ .snbr, .qty ]
    is_such_that( there_isa ORDER from_section[ .cidx, .tot_sects ]
        where( Supp_Nbr = .snbr and Quantity = .qty )
    )
    {
        set .supp_stats[ .snbr ] = [ $#1+1, $#2+.qty ];
    }
    for_each_time [ .snbr, .tot_cnt, .tot_qty ]
    is_such_that(.supp_stats[ .snbr ] = [ .tot_cnt, .tot_qty ]
    )
    {
        to _parent_ do Write_Words( .snbr, .tot_cnt, .tot_qty );
    }
}

next_io_ready_tendril is based on the UNIX select(2) system call. At the Cymbal level, the call in the
query above monitors all of the CHANNEL(pipe) associated with the BUNDLE .bndl of TENDRILS used to do the work in this query. When one of them is ready to do I/O, the FUN returns with a value equal to that TENDRIL. When that value is _null_tendril_, then there are no longer any possibilities of any TENDRIL in .bndl ever being ready for I/O, no doubt due to all of them having exited. A subtlety associated with next_io_ready_tendril is that when a TENDRIL process exits, its pipe assumes end-of-channel status. If some read does not discover that first (and get _instant_eoc_), then next_io_ready_tendril will consider that to be an I/O condition worth reporting, even though there is nothing left to read. In this case, the declarative use of tokens causes not only all the data to be read but also the _instant_eoc_ (which is what tells it to stop generating tokens). Consequently, in this case, the end-of-channel status has been discovered and it will not exist for next_io_ready_tendril to react to. On the other hand, that is not the case for Distribute_Cmds as discussed below. As the following import for next_io_ready_tendril shows, it has a number of different call statuses depending on the patience used.

```python
define CLASS NEXT_IO_READY_TENDRIL_CALL_STATUS
   with_symbols { _worked_, _timed_out_, _would_block_, _interrupted_ } import:
   otherwise_ok TENDRIL FUN(
   for_bundle BUNDLE = _default_process_bundle_ ,
   with_mode INT = _receive_ ,
   with_patience manifest INT = _wait_on_block_ ) next_io_ready_tendril

The possible patience values are the same as they are for new_channel or elsewhere in Daytona: some INT number of seconds to wait, _fail_on_block_ and the default _wait_on_block_. Polling is achieved with _fail_on_block_ where, if there is no TENDRIL ready to talk, then the function returns with status _would_block_. If more than INT patience seconds go by before a TENDRIL becomes ready for I/O, then the call status is _timed_out_. If the parent process is receives an INTERRUPT signal, then the call status is _interrupted_.

Observe how the parent and all the clones each make use of the ARRAY .supp_stats: of course, they each have their own (empty) copy once the clones have been created by new_tendril and so, the ARRAY is not in fact a shared resource.

Note the explicit use of the BUNDLE .bndl to capture exactly the TENDRILS that are to be ‘selected’ on (i.e., the subject of a next_io_ready_tendril call) and to be waited on by next_waited_for_tendril. While not necessary for this query, nonetheless this is good programming practice: the user can inadvertently cause some other child process to be created which is not even associated with any BUNDLE, default or not. The presence of an extraneous process in the environment that next_io_ready_tendril and next_waited_for_tendril are trying to work with can well cause such problems as the parent process hanging (clone.6p31Q). As usual, even when using next_io_ready_tendril, it is still good practice to wait for the clones to finish using _Wait_For_Tendrils: this enables the parent to find out about any unusual fates that the clones may have endured as well as supporting the user getting accurate times on the performance of the ensemble of processes. It also prevents the parent from exiting before its children do, which prevents it from freeing shared resources like TICKET_BUNCHES (see below) before the clones are finished with them.

There is a role here for the with_whole_msgs and with_msg_terminator concepts even when dealing with parents and children TENDRILS talking to each other on a dedicated CHAN(pipe).
message integrity is not threatened by the actions of third parties but nonetheless, message integrity can be valuable. Specifically, if the sender’s buffer is flushed with a partial message at the end, then the receiver can well have been coded to go into an indefinite wait state as it blocks on the pipe trying to read the rest of the message. Unfortunately, the rest of the message will only arrive with the next buffer flush, whenever that is. On the other hand, if the message semantics is defined in the appropriate new_tendril call, then only whole messages will come down the pipe at a time and a receiver will thus avoid any unnecessary waiting. This holds for parents using next_io_ready_tendril to wait for (complete) incoming messages from their children and for children who are waiting for (complete) work assignments from their parents. Please note that by default, with_whole_msgs is assumed for any parent-child CHAN(pipe) with a message terminator of "\n". Consequently, the only explicit option that the user has in the new_tendril call is to use with_msg_terminator to override that default with their own.

19.8.5 Parents And Clones Conversing Frequently

The query clone.1.Q shows how the parent and its children can converse with each other, i.e., for each clone, the parent and clone are able to send and receive messages from each other in the same query. Here, the parent sends to each clone in turn a part number to look up and the clone reports back the associated color immediately.
local: INT .nbr_clones = 3
TENDRIL ARRAY[ 3 ] .ta
STR .color
INT ARRAY[ 3, 3 ] .pno_ara
    = [ 101, 102, 103, 104, 105, 106, 107, 108, 109 ]
for_each_time .cidx Is_In [ 1->.nbr_clones ]
{
    set .ta[ .cidx ]
    = new_tendril( with_downlink _converse_ executing{ do Talk; } );
}
for_each_time .iter Is_In [ 1 -> 3 ]
{
    for_each_time .cidx Is_In [ 1-> .nbr_clones ]
    {
        flushing to .ta[ .cidx ] do Write_Line( .pno_ara[ .iter, .cidx ] );
        set [ .color ] = read_line( from .ta[ .cidx ] )
            otherwise with_msg "parent failed to read .color" do Exit( 2 );
        do Write_Words( "Part", .pno_ara[ .iter, .cidx ], "has color", .color );
    }
}
_Wait_For_Tendrils

define PROC Talk
{
    local: INT .pno
    for_each_time .ii Is_In [ 1 -> 3 ]
    {
        set[ .pno ] = read_line( from _parent_ )
            otherwise with_msg "child failed to read .pno" do Exit( 3 );
        for_each_time .color is_such_that( there_isa PART where( Number = .pno and Color = .color )
        )
        flushing to _parent_ do Write_Line( .color );
    }
}

The establishment of bidirectional communication is accomplished by using the _converse_ value for the with_downlink keyword to new_tendril. Note the parent flushing its message to the clone and conversely: this is required to realize the interactive exchange; otherwise the respective buffers would accumulate messages (without sending them) and the two processes would hang waiting for each other.

19.8.6 Merging Sorted Output With Pipes Back To Parent

When output from a parallel query is required to be sorted, each of the clones can be configured to sort their own output and then send that output back to the parent where the final merge of that output can be accomplished in O(n) time by virtue of a priority queue implemented using heaps. This is faster than each clone sending unsorted data back to the parent, which itself does all the
sorting.

The _Merge_Sorted.Clone_Results ds_m4 macro enables Cymbal to easily support this O(n) merging of clone results (heapmerge.2b.IQ).
_define_( _max_clones_, 60 )
local: INT .nbr_clones

TENDRIL ARRAY[ INT ] .tndrl

TUPLE[ INT, INT, DATE(_mmdyyy_) ] ARRAY[ _max_clones_ ] .ta

define PRED[ TUPLE[ INT, INT, DATE(_mmdyyy_) ] .t1,

TUPLE[ INT, INT, DATE(_mmdyyy_) ] .t2 ] Tuple_Lt
{
    when( .t1#1 < .t2#1 ) return( _true_ )
    else return( _false_ );
}
define PRED[ alias TUPLE[ INT, INT, DATE(_mmdyyy_) ] .t] Handle_Tuple
{
    do Write_Words( .t );
    return( _true_ );
}

set[.nbr_clones] = read( from _cmd_line_ bia[ 5 ] );
set .bun = new_bundle();
fet .i Is_In [ 1 -> .nbr_clones ]
{
    set .tndrl[ .i ] =
        new_tendril( with_downlink _receive_
            for_bundle .bun
            executing { do Process( .i ) } );
}

_Merge_Sorted_Clone_Results(
    _max_clones_, nbr_clones, tndrl, ta, Tuple_Lt, Handle_Tuple )

_Wait_For_Tendrils(.bun)

define PROC( INT .cnbr ) Process
{
    fet [ .en, .ut, .doe ] Is_The_Next_Where(
        there_is_a_bin_for HOPPER where( Hopper_Nbr = .hnbr )
        and there_isa HOPPER from_section[ .cnbr, .nbr_clones ]
        where{
            Hopper_Nbr = .hnbr
            and Entry_Nbr = .en
            and Unix_Time = .ut
            and Date_Of_Entry = .doe
        }
    ) sorted_by_spec[1]
    { to _parent_ with_sep "\001" do Write_Line( .en, .ut, .doe ); } }

Note that in order to work with _Merge_Sorted_Clone_Results, the clones must write their sorted TUPLEs back with \001 separating the components. _Merge_Sorted_Clone_Results takes 6 arguments in
order:

1. an INT, e.g., \_max\_clones\_, specifying the largest number of clones that will ever be run for this query.

2. an INT VBL, e.g., nbr\_clones, containing the number of clones being run for the current invocation of the query.

3. a dynamic associative array VBL of type TENDRIL ARRAY[ INT ] that contains all of the TENDRILS for the clones corresponding to INTS in \[ 1 \rightarrow .nbr\_clones \].

4. a VBL with type TUPLE ARRAY[ \_max\_clones ] where the TUPLE type needs to be fully specified as that of the data being sorted. This array is used for temporary storage of the output.

5. a PREDICATE on pairs of data TUPLES which is a less-than PRED if the data is being sorted in increasing order or else is a greater-than PRED if they are being sorted in decreasing order. (In the latter case of course, the query must be written so that the clones produce and send back TUPLES in decreasing order, not increasing.)

6. a PRED which will be called for each data TUPLE in the final sort; this could well be something as simple as Writing the TUPLE to \_stdout\_. The user should make sure that the code ends with returning \_true\_. (While a PROC might seem sufficient, Daytona’s implementation uses a PRED for this capability in other behind-the-scenes circumstances.)

Note that none of the arguments are dereferenced VBLS as illustrated by .xyz. Also, a TUPLE has to be used to hold the data even if there is only one scalar value per answer coming back from the clones. Running DS M4 on the query will expand it to show the user how heaps are being used to achieve the O(n) merging. Finally, please note that while this optimization will most likely result in faster execution times, it can only do so to the extent that the sorting of the output takes up a substantial portion of the execution time of the query.

19.8.7 Parents And Children Talking Via Funnels

Daytona funnels provide an apparently unique feature supporting communication between a parent and its children. A funnel is a special kind of pipe that connects a parent to all of its children at the same time! It’s a miracle. If any child writes a message into a funnel, the parent will get it; the atomic nature of writing into pipes provides automatic synchronization among the children: even as they compete for access to the funnel, no output gets lost, no output gets corrupted. However, this can only happen if the length of the messages is less than the machine’s value for the UNIX constant PIPE\_BUF because it is only then that clone buffer writes are atomic. As seen in clone.8d.Q below, funnels provide quite a nice way for clones to communicate their results back to their parent whenever they have something to say. clone.8d.Q computes the same quantities as clone.7d.Q, only in clone.8d.Q, the clones communicate their information back to their parent through a funnel and it is unaggregated base data that they send back instead of locally computed aggregates.
local: INT .tot_sects = 5
    TUPLE[ INT .tot_ordered, INT .tot_qty ] .supp_stats[ INT(_short_) .snbr ]
        = { ? => [ 0, 0 ] }
set .funl_chan = new_channel( via _funnel_ ) otherwise do Exit( 3 );
set .bndl = new_bundle();
for_each_time .cidx Is_In [ 1-> .tot_sects ]
{
    set ? = new_tendril( for_bundle .bndl
        with_downlink _receive_
        executing{ do Process( .cidx ); } );
}
with_mode _write_ do Close( .funl_chan ); // mandatory
for_each_time [ INT(_short_) .snbr, INT .qty ]
is_such_that(
    [ .snbr, .qty ] = tokens( from .funl_chan upto " \n" )
){
    set .supp_stats[ .snbr ] = [ $#1+1, $#2+.qty ]
}
_HWait_For_Tendrils(.bndl)
for_each_time [ .snbr, .tot_cnt, .tot_qty ]
is_such_that(
    [ .snbr, .tot_cnt, .tot_qty ] Is_The_Next_Where(
        .supp_stats[ .snbr ] = [ .tot_cnt, .tot_qty ] ) in_lexico_order
){
    do Write_Words( .snbr, .tot_cnt, .tot_qty, ((FLT).tot_qty)/tot_cnt );
}
declare PROC( INT .cidx ) Process
{
    for_each_time [ .snbr, .qty ]
is_such_that( there_isa ORDER from_section[ .cidx, .tot_sects ]
        where( Supp_Nbr = .snbr and Quantity = .qty )
    ){ do Write_Words( .snbr, .qty );
    do Close( .funl_chan );
}

The most important thing to remember about a parent reading from a funnel to its children is that the parent must close the funnel CHAN for writing immediately after launching the new TENDRILS. Otherwise, the program could hang. Also, while not mandatory since program exits will close a funnel CHAN, it is nonetheless good practice to explicitly close a funnel CHAN in a child immediately after finishing the last write to it; this too will prevent program hangs in atypical cases.

It is OK to have two or more funnels that the children are writing into BUT there must be an order on the funnels where each child does ALL of its writing to EACH funnel in turn in that order
(AND then immediately, explicitly Closes that funnel) whereas the parent reads everything thing there is to read from a funnel before moving on to the next funnel -- in that shared common order (clone.8e.Q). And of course the parent still has to close ALL the funnel CHAN for writing immediately after launching the new TENDRILS.

By default, CHAN(_ funnel_) are opened with whole msgs. Their new_channel call can be customized with a with_msg_terminator argument as appropriate.

Another fascinating fact about funnels is that they can also work in reverse, i.e., when the parent writes messages into the funnel and the clones read from it. In this situation, it is first-come, first-served for the clones: the clones are all competing to read from the funnel and non-deterministically, they will succeed in some unpredictable order, meaning that each message sent down by the parent will be read by exactly one clone (not more than one) and it is a race to see which of the clones trying to read will get the message. This behavior is perfect for the parent assigning work to whichever clone is ready to do it.

However there is a MAJOR caveat with reverse funnels. They can only work under very special circumstances. In particular, they will not work with variable length messages. What happens is that since there are multiple buffered readers, when each reader succeeds, it will typically read an entire buffer’s worth of data from the funnel. Unfortunately, this fixed size buffer’s worth is usually not an integral number of messages and so, what will occur is that a given clone will wind up with a partial message and like as not, never see the other portion. One way to get around this sad situation is to have fixed size messages and to use the fixed size reading capabilities of Cymbal to have each clone read integral numbers of messages at a time (while remaining less than the buffer size in aggregate with each read). Another possibility is for each variable length message be prefixed with a fixed size message length and to have each reader use a TICKET_BUNCH to read atomically the length and the corresponding message in one action. At any rate, reverse funnels must be used with care.

19.9 Synchronizing Using TICKET_BUNCHES

Obviously, a lot of parallel computation can be done in Cymbal without the need for explicit synchronization using semaphores, (OS) mutexes, condition variables, and so forth. However, there are occasions when an additional explicit synchronization mechanism is needed. For Cymbal, that mechanism is provided by TICKET_BUNCHES. The underlying idea is simple. Any given TICKET_BUNCH has some number of tickets. Any process that knows about the TICKET_BUNCH can request some number of those tickets. As the requests are granted, the number of tickets in the TICKET_BUNCH goes down by the amount of the request, subject to the obvious requirement that the number of tickets cannot be decreased below zero. In fact, if a request is made that, if satisfied, would take the number of tickets below zero, then the requestor blocks waiting for the requisite number of tickets to become available, at which point the request is satisfied. So, the paradigm is a common one: there are only so many tickets available to get into the show -- if they run out, the customer has to wait until enough are returned to cover the amount being requested.

In the simplest case, when the number of tickets in a TICKET_BUNCH is exactly one, then the TICKET_BUNCH operates like a mutex (i.e., a mutual exclusion lock). TICKET_BUNCHES are implemented using UNIX semaphores. Here are the relevant imports:
TICKET_BUNCH FUN( with_name STR = __default_name__, with_total_tickets INT = 1 )
    new_ticket_bunch
otherwise_ok INT FUN( from TICKET_BUNCH,
    with_patience INT = __wait_on_block__, with_count INT .nbr = 1 )
    get_tickets
otherwise_ok PROC( to TICKET_BUNCH, INT .nbr = 1 ) Return_Tickets
STRUCT{ STR(=) .Name, INT .Index, INT .Semid } VBL any_ticket_bunch

These should be self-explanatory, where the Semid STRUCT component is the UNIX semid integer identifier for the semaphore that implements the corresponding TICKET_BUNCH. The with_patience feature has the same semantics as it does for new_tendril and others.

In the next query (clone.4r3.Q), a collection of clones compete to be the last one to write their message into a shared file that is opened by the parent. Access to the shared file DS_TMPFL is governed by the .mutex_tb TICKET_BUNCH by defining a critical region consisting of the code between the acquisition of the mutex through get_tickets and the subsequent release via Return_Tickets. Files shared in this way that have been opened for writing (not appending) need to be opened by new_channel using the with_user_sync keyword.

In general, any file neither a pipe, nor stdout, nor opened for append that is worked with in a TICKET_BUNCH critical region must be opened with_user_sync in its new_channel call. The keyword implies that the Cymbal user will be explicitly providing synchronization, for example, by means of TICKET_BUNCHES. The model is that when the execution of TENDRIL code enters a critical region, the user must assume that other clones may have been reading/writing to any common shared files. Consequently, the user must explicitly seek to wherever they would like to start reading or writing before doing so. Failure to explicitly seek can well result in programs that will randomly misbehave, sometimes as little as 1 percent of the time. That is subtle misbehavior well worth avoiding.
set [ .nbr_clones ] = read( from _cmd_line_ but_if_absent[5] );
set .milestone_chan
    = new_channel( for "DS_TMPFL" with_mode _clean_slate_update_ with_user_sync );
set .mutex_tb = new_ticket_bunch();
set .rally_tb = new_ticket_bunch( with_total_tickets 0 with_name "rally" );
for_each_time .idx Is_In [ 1-> .nbr_clones ]
{
    set ? = new_tendril( executing{ do Lay_Claim(.idx); } );
}
set ? = get_tickets( from .rally_tb with_count .nbr_clones )
    otherwise do Exit( 3 );
from _start_ with_offset 0 do Seek_In( .milestone_chan );
set [ .line ] = read_line( from .milestone-chan ) otherwise do Exit(4);
do Write_Line(.line);

define PROC( INT .idx ) Lay_Claim
{
    set ? = get_tickets( from .mutex_tb with_count 1 )
        otherwise do Exit( 3 );
    from _start_ with_offset 0 do Seek_In( .milestone-chan );
    flushing to .milestone-chan do Write_Words( "Clone", .idx, "was last here" );
to .mutex_tb do Return_Tickets( 1 );
to .rally_tb do Return_Tickets( 1 );
}

The .rally_tb TICKET_BUNCH is used to provide a rally point or what is also called a barrier for
the clones. .rally_tb starts out with zero tickets; as each clone finishes, it adds a ticket to
.rally_tb. Meanwhile, the parent is waiting for .rally_tb to contain .nbr_clones tickets.
Consequently, when the last clone adds a ticket, the parent knows that all clones have finished and it
can proceed with finishing up the job. In this query, this rally strategy could have been implemented
with _Wait_For_Tendrils, which would have been preferable, actually, since then clone exit status
information would have been collected. Clearly though, TICKET_BUNCHES can be used to provide
rally points at intermediate points in the overall computation as well.

shm.stress.wr.5.Q shows how to implement a more useful rally in that in can be used over and
over again. Here is a sketch of how that is done:
_define_(MAX_NBR_CLONES,64)

export:
   INT .nbr_iter = 100, .nbr_clones = 5
   TICKET_BUNCH .rally_pt;
   TICKET_BUNCH ARRAY[MAX_NBR_CLONES] .mutex_tb;

set .rally_pt = new_ticket_bunch( with_total_tickets 0 );
fet .i Is_In [1->.nbr_clones] {
   set .mutex_tb[.i] = new_ticket_bunch( with_total_tickets 0 );
}

fet .i Is_In [1->.nbr_clones] {
   set .kid[.i] = new_tendril( with_name (STR) .i
   with_downlink _receive_ for_bundle .my_kids
   executing { do Insert_And_Read(.i) } );
}

_Write_For_Tendrils(.my_kids)

global_defs:

define PROC( INT .clone_id ) txn task: Insert_And_Read {
}

import:
   INT .nbr_iter, .nbr_clones
   TICKET_BUNCH .rally_pt;
   TICKET_BUNCH ARRAY[MAX_NBR_CLONES] .mutex_tb;

fet .iter Is_In [ 1 -> .nbr_iter ] {
   // do the work for this iteration

to .rally_pt do Return_Tickets(1) otherwise
   with_msg "error: can’t Return_Tickets for .rally_pt" do Exit(33);

when( .clone_id != 1 ) {
   set ? = get_tickets( from .mutex_tb[.clone_id] with_count 1 )
   otherwise do Exit( 3 );
} else

when( .clone_id = 1 ) {
   set ? = get_tickets( from .rally_pt with_count .nbr_clones )
   otherwise do Exit(22);

   // reinitialize data structures for next iteration

fet .ii Is_In [ 2 -> .nbr_clones ] {
   to .mutex_tb[.ii] do Return_Tickets(1) otherwise with_msg
This shows how to iterate over a bunch of work making sure that all the clones start at the same time on each iteration.

At the lowest-level, semaphores can be a nuisance to use because by nature they are happy to survive forever after the lifetime of the process that created them. Fortunately, Daytona is very zealous and successful about removing the semaphores that implement TICKET_BUNCHES after they are no longer needed.

The main caveat with TICKET_BUNCHES is that the parent which created the TICKET_BUNCH cannot exit before the last clone that uses that TICKET_BUNCH has exited -- because the parent is the owner and the TICKET_BUNCH is destroyed on parent exit. Here is the kind of message from a clone that will result if the parent exits prematurely and causes a semaphore still in use to be destroyed:

```
error: cannot return ‘1’ tickets from a TICKET_BUNCH named ‘mutex’
and with semid ‘20643843’
```

THE MOST LIKELY EXPLANATION is that the owner of the TICKET_BUNCH (i.e., the parent process) has already exited, thus causing its TICKET_BUNCHES to be destroyed:

instead the parent should wait and not exit before its children finish

Most likely, the remedy to this malady is to use _Wait_For_Tendrils.

For general use, the UNIX ipcs and ipcrm commands can be used to examine and manage semaphores (and shared memory and message queues). Also, the Cymbal PROC( TICKET_BUNCH ) Describe will print out informative information about TICKET_BUNCHES. If the UNIX command ipcs shows you a semaphore that you own, you can get the complete information about that semaphore (indeed much more than ipcs -sa will deign to show) by executing $DS_DIR/Describe_Semid <semid>. Also at the shell level are the very helpful Show_Sem(1) and Rm_Sem(1) -- see their DS Man pages.

### 19.10 Parents Signalling Children

Signals provide an asynchronous way for a parent to communicate to its children: the clones needn’t be listening to a pipe nor polling a file somewhere -- instead they can be busy computing and the signal from the parent will come in and interrupt them causing their own appropriate signal handling PROC to be called to make sense of the interruption. One occasion when a parent process needs to signal some of its child clone TENDRILS occurs when the various clones are each searching their own portion of the search space and one of them finds the answer and reports that back to the parent. Then, since there is no need for the remaining clones to continue, the parent can signal them to stop (i.e., clean up and exit). That can be easily done using the likes of:
Here are the imports for the signal-related fpps:

define CLASS SIGNAL
    with_symbols { _abort_, _quit_, _hangup_, _kill_, _terminate_, _pipe_broken_,
        _tstop_, _stop_, _continue_, _interrupt_, _alarm_, _usr1_, _usr2_, _child_,
        _fpe_, _bus_, _segv_, _xcpu_, _xfsz_ }
import:
    overloaded PROC( to BUNDLE|TENDRIL, INT .signal ) Signal
    PROC( to BUNDLE, INT .signal ) Signal_Bundle
    PROC( to TENDRIL, INT .signal ) Signal_Tendril
    PROC( for INT .signal, PROC( INT .signal ) .handler ) Install_Signal_Handler
    PROC( for INT .signal ) Install_Default_Signal_Handler
    PROC( for INT .signal ) Ignore_Signal
    PROC( for INT .signal, STR .msg ) Exclaim_Signal_Msg
    PROC( INT .signal ) Print_Current_Sighandler_For
    PROC Push_Hold_Signals, Pop_Hold_Signals
    PROC( PROC( INT .signal ) .handler, UINT .secs ) Install_Handler_And_Set_Alarm
    PROC Restore_Alarm_Signal_Handler
    PROC Turn_Off_Alarm

Note that it is possible to Signal every not-waited-for TENDRIL in a BUNDLE by signalling the
BUNDLE. Given the appropriate PROC definition, it is easy to install a signal handler PROC as seen
in:

    import: sighandler PROC( INT .sig ) task: Handle_Usr1
    for _usr1_ do Install_Signal_Handler( Handle_Usr1 );
global_defs:
    define sighandler PROC( INT .sig ) task: Handle_Usr1
    { 
        do Exclaim_Line( "USR1 signal!!" );
        with_msg "I just got a USR1 signal = .sig" ISTR
            do Exit( 98 );
    }

Important: any PROC that is an argument to Install_Signal_Handler must be defined with the
sighandler keyword as illustrated above: this supports Daytona defining and using a special environment for
handling signals. The import is needed here so that when the parser encounters the import first, it
knows to treat the Handle_Usr1 argument as a PROC in the call to Install_Signal_Handler. sighandler is
considered to be part of the type of the PROC as indicated by its presence in the import. Note that
signal handlers cannot be installed for _kill_ or _stop_. The PROCS Install_Default_Signal_Handler and
Ignore_Signal perform as their names indicate.

While somewhat tricky to work with, the utility of _child_ is that the parent is asynchronously
notified of when the child process exits, and thus, the parent knows as soon as possible that the child
has finished and can react accordingly in its signal handler for _child_, including probably waiting for the child process’ exit status. However, since the handling strategy for _child_ is inherited by any clones, they will probably need to restore the default handling of _child_, which incidentally is to ignore it. However, on Solaris and Linux, if the user calls the Cymbal Ignore Signal for _child_, then children as zombies will never be created, whereas if they call Install_Default_Signal_Handler, then they will. Zombies can only be waited for if they exist! In other words, the behavior of the SIG_IGN and SIG_DFL handlers are different in handling SIGCHLD. For more information on this tricky matter, see "Advanced Programming In The UNIX Environment -- Second Edition" by Rago and Stevens (page 308).

There may be sections of code wherein a process subject to receiving signals does not want to be disturbed. In that case, the PROC Push_Hold_Signals may be called to cause the following signals to be held until after the corresponding PROC Pop_Hold_Signals has been called: _hangup_, _interrupt_, _quit_, _terminate_, _broken_pipe_, _child_, _alarm_, _usr1_, _usr2_. Calls to this pair of PROCS may be nested.

Exclaim_Signal_Msg is used to write an English message identifying a signal number to _stderr_. Print_Current_Sighandler will print out a message identifying the current signal handler for the given signal argument: the possibilities include SIG_DFL, SIG_IGN, and SIG_HOLD as well as the hex address of user-installed signal handlers. (For executables that have not had their symbols stripped away, nm(1) can be used to associate the hex address with the name of the handler.)

Here is an example of a signal handler PROC that the parent of a bunch of clones would use: when that parent receives a signal to die, it first sends SIGTERM signals to its clones, waits for them, and then exits with a message.

```prolog
define sighandler PROC( INT .sig ) Parent_Handle_Signal

{ to .mybun do Signal( _terminate_ );
loop {
    set .kid = next_waited_for_tendril( for_bundle .mybun )
otherwise_switch {
    case{ = _no_such_kid_ | = _interrupted_ ){ break; }
    case( = _failed_kid_ | = _killed_kid_ | _stopped_kid_ ){ } }
else{ do Exit( 4 ); }
}
do Exclaim_Words( "Just terminated Clone_" + (STR).kid.Index );
for .sig do Exclaim_Signal_Msg( "parent exiting due to" );
do Exit( 1 );
}
```

There are occasions when a process wants to send a signal to itself. Here is an example of a txn asking for the OS to send it an alarm signal in 4 seconds. The handler will print a message, restore the previous alarm handler, and cause the txn to abort (alarmtxn.1.IQ).
do Doit;

global_def:
define PROC txn task: Doit
{
    define sighandler PROC( INT .sig ) : Handle_Alarm
    {
        do Write_Line("Just received SIGALRM = .sig; and now aborting ...");
        do Restore_Alarm_Signal_Handler();
        do Abort();
    }
    do Install_Handler_And_Set_Alarm_For( Handle_Alarm, 4 );
    do Sleep( 10 );
}

Of course, the alarm handler could do anything else it wanted if the user did not want to abort the txn.

Turn_Off_Alarm will turn off any alarm that may be active.

19.10.1 Handling Interrupts To stdin

When constructing an interactive user interface where the Cymbal program is reading user input from stdin, it is natural to want to allow the user to hit the interrupt key and have the program catch that signal and reset back to an earlier state. For example, suppose the user is in the midst of typing in multiple lines of input to the program and then thinks the better of it. They would want to hit the interrupt key and be able to start over again from scratch (by having the program discard what they had been typing and starting to read again fresh). Here is a test query doing this for one-line inputs (intr.read.IQ):
define PROC( INT .signal ) Do_Nothing{}

local: STR .cmd

for _interrupt_ do Install_Signal_Handler( Do_Nothing );

loop {
    read_again:  
    do Exclaim( "Enter command: " );
    set [.cmd ] = read_line()
    otherwise_switch {
        case( = _interrupted_ ){
            do Exclaim_Line( "Just got interrupted, now resuming the read" );
            go read_again;
        }
        else{ do Exit(3); }
    }
    do Write_Line(.cmd);
}

When the user hits interrupt while the program is doing the read_line(), what happens internally is that the signal will cause the Do_Nothing handler to be invoked, which since it does nothing but return, will then cause the read call to return and the appropriate case of the switch to be executed.

19.11 TENDRIL Spawn

The spawning keyword to a new_tendril call will create a TENDRIL that corresponds to a new child process that is running either a specified command or shell program.

set .kid = new_tendril(
    spawning `~/home/john/bin/Run_It 2 Charles 3/3/99@12am`CMD );

In this simple example, the indicated command (CMD) is executed in a child process where the .kid TENDRIL object provides the means by which the user can manage this execution. Every CMD consists of a filepath to some executable program followed by a sequence of arguments that are separated by one or more blanks. To run the program, a UNIX fork following by an execvp is executed: essentially, this just means that a child (and clone) of the calling process is created and then overlayed/replaced with an execution of the program identified by the CMD filepath whose argc/argv type arguments are given by the remaining tokens of the CMD string. One implication of this is that there is no shell involved in this process of invocation and so, the arguments are not going to experience any shell expansion prior to being made available to the program.

Here is a simple program illustrating how a spawned child can communicate back to its parent (spawn.1.Q):
Just as with clones, the \texttt{with\_downlink} keyword specifies the nature of the communication between parent and child. Note that again (in the \texttt{tokens} call), the TENDRIL .uten is being confounded with the CHANNEL that links child to parent. And, also once again, since the TENDRIL runs asynchronously in relation to its parent, it is recommended to use \_\texttt{Wait\_For\_Tendrils} to wait for spawn exit statuses.

The next example shows the parent sending several lines of output down to a child sorting process and then reading the reverse sorted output back for printing (spawn.1.Q):

\begin{verbatim}
set .uten = new_tendril( with_downlink _converse_ spawning `sort -r`CMD );
to .uten do Write( "abc\nxyz\ndef\nabc\njkl\nghi\nuvx\n" );
// do Flush( .uten );
with_mode _write_ do Close( .uten );
loop {
  set [.line] = read_line( from .uten )
  otherwise_switch {
    case( = _instant_eoc_ ){
      break;
    }
    else { do Exit( 3 ); }
  }
  do Write_Line( .line );
}
do Close( .uten ); // not necessary since done in next_waited_for_tendril
\_\texttt{Wait\_For\_Tendrils}
\end{verbatim}

As is customary when dealing with pipes, it is necessary to explicitly cause the parent’s buffer to be flushed in order to deliver the sum total of the parent’s message to the child. Typically, this is done with a \texttt{Flush} call or with a \texttt{flushing} \texttt{Write} keyword, which is not necessary here though because it is part of what happens when the \texttt{with\_mode\_write\_do\_Close} occurs. The latter statement is absolutely necessary because not only does the parent’s buffer need to be flushed, but so also does the \texttt{write} end of the pipe need to be closed so that the child can get the end-of-file (i.e., end-of-channel) that it needs in order to know that it has received all there is to receive on that pipe.

Alternatively to CMD, Korn shell programs can be spawned. All that is needed is to pass a SHELLP (shell program) argument to the \texttt{spawning} keyword. SHELLP constants follow the same pattern as CMD constants do, i.e., they are strings enclosed by hats followed by the class name as in \texttt{`date; top -b 10`SHELLP}. The casting command (SHELLP) enables the use of ISTR in forming the shell program to be executed as illustrated in the following \texttt{new\_tendril} call where information on the top 10 processes is logged every 10 minutes to a special file named according to the \texttt{.user} parameter read in from the command line.
set [.user] = read( from _cmd_line_ );
set .uten = new_tendril(
  spawning (SHELLP)
  "if [ ! -d ~/.user/logdir ]
  then { mkdir -p ~/.user/logdir ; } 
  fi
  rm -f ~/.user/logdir/log
while true 
do
  top -b 10 >> ~/.user/logdir/log
  sleep 60
  done"ISTR );
_Wait_For_Tendrils

This program could be nohupped and run in the background as a daemon. (This is just a pedagogical
example: in real life, one would use crontab(1).)

The use of SHELLP for a new_tendril is superior to the Shell_Exec command described elsewhere.
Shell_Exec is a synchronous command that executes its command via a ksh -c. As such, the argument to
Shell_Exec must be of a form that ksh -c can execute. On the other hand, executing a SHELLP with a
new_tendril call is done asynchronously and can employ any valid Korn shell program for the SHELLP.
When Daytona processes a SHELLP spawn, it needs to create temporary files of the form
$([DS_TMPDIR:-.]/DS_SPWN-<child_pid>); clearly, if DS_TMPDIR is not set, these files are created in
(and removed from) the current directory.

Here is the complete import for new_tendril as it is used when spawning child CMD or SHELLP:

TENDRIL(_process_) FUN: new_tendril(
  with_name STR = _default_name_,
  for_bundle BUNDLE(_process_) = _default_process_bundle_,
  with_downlink _3GL_TEXT = _none_,
  (0) with_whole_msgs = ?, // _present_
  with_msg_terminator STR = "\n",
  spawning CMD|SHELLP
)

A spawn shares its parent's stdout if with_downlink is neither _receive_ nor _converse_, in which
cases, the spawn's stdout is redirected into a pipe writing up to the parent.

In the event that the parent process will be spawning and communicating with an indefinite
number of children without waiting for them until the very end, it may be necessary to call Close on the
TENDRIL spawn in order to prevent exhaustion of open file descriptors due to unClosed TENDRIL
I/O CHAN.
set .hi_cnt = 0;
let .ii Is_In [ 1-> 1000 ]
{
    set .ten = new_tendril( with_downlink _receive_ spawning ^echo hi`SHELLP );
    set [.x ] = read_line( from .ten );
    when( .x = "hi" ) set .hi_cnt++;
    else do Exclaim_Line( "error in spawn.7.1.Q" );
    // Must Close the downlink CHAN(_tendril_pipe_) or else get too many open files
    do Close( .ten );
}
_Wait_For_Tendrils

But an explicit Close is usually not necessary because the user will probably _Wait for the spawn before spawning another (to make sure that all went well with the former): fortunately, Daytona assumes that a parent that has waited for one of its children is no longer interested in keeping open any associated I/O CHAN and Daytona will Close them automatically.

19.12 _fifo_ CHAN For The Unrelated

Pipes can be only be used between processes when the two processes in question have a parent-child relationship. How then would two clones or two spawn or two Cymbal processes owned by different userids converse with each other? Cymbal offers the CHAN(_fifo_) for this purpose as described in Chapter 8. UNIX also offers shared memory and message queues for this purpose but Cymbal does not yet support those directly.

19.13 UNIX Tools

A not infrequent misbehavior of parallel programs occurs when the processes hang and no longer consume any CPU resources. In order to debug this and other misbehaviors, the user can make good use of a variety of software/performance tools that vary by name and function across all the different varieties of UNIX. ps displays the status of various processes; top provides a display updated every few seconds of the top few processes by CPU utilization. These tools can verify whether or not the suspect processes are hung or not. To identify which system call might be blocking in a hung process, Solaris uses truss, SGI Irix uses par, and Linux uses strace. In /usr/proc/bin on Solaris, there are a variety of useful commands including ptree which takes a pid argument and prints out a tree of all ancestor and descendant processes.

19.14 Distribute_Cmds: An Illustrative, Useful Tool

The ideas presented in this chapter are nicely illustrated by Distribute_Cmds.cy, a Cymbal program whose executable Distribute_Cmds comes with Daytona in $DS_DIR. This program takes a file of one-line commands, possibly from stdin, and parallelizes them by sending them to the next available clone from a fixed set of clones, where they are executed by the shell. It also has a daemon mode which is described in the DS Man page and so is not discussed here nor in fact, illustrated by the source code given here, which is from the first version of this command. The Cymbal source code for the complete
and current program is included in $DS_DIR.

The net effect is that for $k$ clones, there are always $k$ commands running until there are less than $k$ that haven’t been run yet. This is fairly well balanced parallelism whose virtues are highlighted when compared with other simpler schemes such as the one that just assigns the first $n/k$ of $n$ commands to the first clone, the second $n/k$ to the second and so on. In that latter case, clones can well finish early if the batch of commands they received takes significantly less time to complete than the longest time-consuming batch. On the other hand, Distribute_Cmds works by having the parent put the set of commands to do in a box and then launching the clones, each of which asks the parent for a command to do when it becomes idle. Unfortunately, it’s easy to see that this is not necessarily the optimal schedule. Consider the case where there are two clones and three jobs taking $x$, $x$, and $2x$ time, respectively. If the two smaller jobs run first, then the job is over in $3x$ time; if a big one and small one are run first, then the job is over in $2x$ time. So, it matters which job is assigned to which clone first. If the jobs can be sorted by size, then a better strategy is to give the larger jobs to the clones first simply by having them appear sorted by size as input to Distribute_Cmds.

```c
/*
 * Take a file of one-line commands, possibly from stdin, and
 * parallelizes them by sending them to the next available clone
 * from a fixed set of clones, where they are executed by the shell.
 *
 * Putting ./Stop_Distribute_Cmds in the directory where Distribute_Cmds is running
 * will cause it to cease issuing work
 *
 * Also, you can kill the lot by using psme to find the process group id pgid
 * and then issuing kill -TERM -pgid
 */
local:
    STR .usage = "usage: Distribute_Cmds <nbr_clones> ( - | <file_of_cmds> ) "
    INT .nbr_clones
    LIST[ STR ] .cmd_box
    STR .cmd_file, .response
    BUNDLE .bundle = new_bundle();

set [ .nbr_clones ] = read( from _cmd_line_ )
    otherwise { with_msg "need nbr of clones\n\n.usage"ISTR do Exit(1); }
set [ .cmd_file ] = read( from _cmd_line_ )
    otherwise { with_msg "need a file of commands\n\n.usage"ISTR do Exit(1); }

set .cmd_box = [ .cmd : [ .cmd ] = ptokens( for "cat .cmd_file"ISTR upto "\n" ) ];
when( .nbr_clones > .cmd_box.Elt_Count ) set .nbr_clones = .cmd_box.Elt_Count;
set .clone_nbr Is_In [ 1 -> .nbr_clones ]
{
    set ? = new_tendril( for_bundle .bundle with_downlink _converse_
        executing { do Exec_Cmd } );
}
```

Copyright © 2013 AT&T
All Rights Reserved.
September 15, 2013
fet .cmd Is_In .cmd_box
{
    when( "./Stop_Distribute_Cmds" File_Exists ) break;
    set .worker = next_io_ready_tendril( for_bundle .bundle )
    otherwise do Exit(3);
    set [ .response ] = read_line( from .worker )
    otherwise {
        do Exclaim_Words(
            "error: Distribute_Cmds: failed to read_line from clone with pid",
            .worker.Sys_Id.Pid );
        continue; // want to continue with all the commands are that are OK!
    }
    when( .response != "Please send command." )
    {
        do Exclaim_Words(
            "error: Distribute_Cmds: failed to get meaningful response from clone with pid",
            .worker.Sys_Id.Pid );
        do Exclaim_Words( " instead just got `.response'" );
        continue; // want to continue with all the commands are that are OK!
    }
    when( .response != "Please send command." )
    {
        with_msg "Can’t understand clone response, namely `.response'"ISTR
        do Exit( 6 );
    }
    flushing to .worker do Write_Line( .cmd );
}
// let the clones finish and then tell them all to die
loop{
    set .worker = next_io_ready_tendril( for_bundle .bundle )
    otherwise do Exit(5);
    when( .worker = _null_tendril_ ) break; // no more kids to wait for
    set [ .response ] = read_line( from .worker )
    otherwise_switch {
        case( = _instant_eoc_ ){ continue; } else { do Exit( 8 ); } }
    when( .response != "Please send command." )
    {
        with_msg "Can’t understand clone response, namely `.response'"ISTR
        do Exit( 9 );
    }
    flushing to .worker do Write_Line( "#Exit Please" );
}
_wait_For_Tendrils
define PROC Exec_Cmd
{ local: STR .cmd
  loop {
    flushing to _parent_ do Write_Line("Please send command.");
    from _parent_ do Read_Line( cmd );
    when( .cmd = ">#Exit Please" ) break;
    set .uten = new_tendril( spawning (SHELLP)( .cmd+"\n" ) );
    set ? = next_waited_for_tendril( namely .uten ) otherwise do Exit(4);
  }
}

Each of the clones does no more than execute the loop in the PROC Exec_Cmd, where it requests a command from the parent and reads it. If the command is #Exit Please, then the clone breaks out of the loop and exits; otherwise, the clone spawns a SHELLP child process to execute the current command.

On the parent’s end, after launching the clones, the parent loops over the commands and sends each one to the next clone that says that it is ready to work on one. After all the commands have been distributed, it is now time to tell the clones to exit. The logic is a little subtle here because of the way next_io_ready_tendril works. Recall that when a clone dies, his (now empty) pipe to the parent enters end-of-channel status and that this is considered to be an I/O event that next_io_ready_tendril will report if it gets to see it -- and noticeably without distinguishing this end-of-channel status from actually having something to read. Consequently, the program logic must handle instant_eoc on the .worker_CHAN not as an error but as an opportunity to continue on with handling the clones. So, in this close-down-processing loop, every clone is dealt with twice: first, to tell the clone to exit and second, to handle its subsequent end-of-channel status. Once all of the clones have died, the parent waits on their exit statuses. Another way to think of this is that one way or another, by a read or by a next_io_ready_tendril, an end-of-channel status must be detected and dealt with.

Thus it is seen that Distribute_Cmds illustrates most of the ideas in this chapter and provides quite a nice general-purpose tool as well.
20. Parallelization Made Easy

The preceding chapter discusses parallelization primitives for Cymbal, which, while higher-level than the ones UNIX offers, nonetheless usually require some intricate programming to create parallel Cymbal programs. What would be the ideal way to make parallelization easier and more powerful for Cymbal? The first decision to make would be on what to parallelize, i.e., in which of Cymbal’s dialects should the parallelization directives appear? Certainly, the greatest impact can be achieved by parallelizing the most powerful, most expressive and concise dialect and clearly, that would be the declarative portion as expressed by symbolic logic assertions. Contrast that to the situation for procedural languages where parallelization is often achieved much as in the same style as it was in the preceding chapter: explicit directives to create and synchronize new threads (Cilk); or perhaps by parallelizing C for-loops (UPC: Unified Parallel C).

Having chosen the dialect to parallelize, the next two requirements are, first, how to specify what assertions are to be parallelized and second, how to specify how many workers (such as processes/threads/CPUs/cores) should be utilized. And of course, as always, it is important for the user to be able to understand what a program written using parallel programming syntax will do, i.e., what it means, what its semantics is: ideally, the semantics of a parallel program would be those of the single-threaded program that results by removing all parallel-related keywords/directives. Lastly and facing up to the fact that there are often many ways to parallelize problems that may in fact welcome different solutions in different environments, the ideal parallelization strategy should support the simple specification of a variety of alternatives.

Not surprisingly then, Cymbal does achieve this ideal parallelization and it does so by introducing exactly two new required keywords: parallelizing and parallel_for. parallelizing is used to identify which value-generating sub-assertion is to be parallelized and second, how to specify how many workers (such as processes/threads/CPUs/cores) should be utilized. And of course, as always, it is important for the user to be able to understand what a program written using parallel programming syntax will do, i.e., what it means, what its semantics is: ideally, the semantics of a parallel program would be those of the single-threaded program that results by removing all parallel-related keywords/directives. Lastly and facing up to the fact that there are often many ways to parallelize problems that may in fact welcome different solutions in different environments, the ideal parallelization strategy should support the simple specification of a variety of alternatives.

Architecturally, this parallelization framework derives its flexibility and execution speed from the job distribution paradigm. Job distribution parallelization occurs when the work to be done can be partitioned into a set of jobs which the parent clone assigns one at a time to child clones as they finish and request additional work. The dynamic runtime nature of job distribution helps solve the central problem facing all parallelization schemes, i.e., the fact that, since all the work is not done until the last worker is done, to be efficient and as fast as possible, all the workers have to finish all their several work assignments at roughly the same time. Static work assignments (done at the start of processing) often fail to achieve this ideal because it can simply be infeasible to compute in advance how much work will actually be involved in the (static) work assignments made to each worker; furthermore, static work assignments by construction cannot adapt to changing conditions at runtime which may slow down or speed up individual workers.

Note that static work allocation schemes can be handled, if desired, by the job distribution scheme by the simple strategy of dividing the work into $k > 1$ pieces and handing them out to $k$ workers,
with each worker getting one and only one piece of work.

So, in general, how does all this get put together for Cymbal? The answer is to (at least implicitly) write an OPCOND whose assertion is a conjunction whose last conjunct is a parallelizing assertion of the form \texttt{parallelizing(assertion)}; the free variables in the parallelizing assertion that are generated by the preceding remainder of the OPCOND assertion are the ones that define the jobs to be done. In other words, the jobs are defined by first identifying the largest TUPLE of VBLs that are each local to the OPCOND (i.e., not "outside" (being defined elsewhere)) and that are each free in \textit{both} the parallelizing assertion and its preceding conjuncts, each of the two groups of assertions considered separately. Then the LIST of jobs is the LIST of values for this TUPLE of VBLs that is generated for the OPCOND excluding the parallelizing assertion. The Cymbal OPCONDS that are currently parallelizable are those that correspond to BOXes, Displays, and \texttt{for_each_times}. Furthermore, a clone’s response to getting a job, i.e., values for the job VBLs, is to process the parallelizing assertion by generating all possible values for the VBLs that are free in the assertion but not defined elsewhere.

The most apparent need for parallelization arises when querying large databases where, obviously, such queries can take quite a long time when using only one CPU. However, to illustrate the general principles of Cymbal’s parallelization paradigm, consider the following artificial example (paraXjobdist.3.Q).

```plaintext
set [ .nbr_clones ] = read( from _cmd_line_ bia[ 5 ] );

with_format _table_
  parallel_for .nbr_clones
    sorted_by_spec[1]
do Display each [ .cohort, .sum_over_cohort ]
each_time{
  .cohort Is_In [ 1000 -> 10000 by 1000 ]
  and parallelizing(
    .sum_over_cohort = sum( over .x
      each_time( .x Is_In [ .cohort -> .cohort + 999 ] ))
  )
}
```

The output incidentally is:
The Cymbal in this example instructs Daytona to parallelize the computation of sums of INTS in given ranges. The idea is that each such sum is a job to be given to one of .nbr_clones clones upon that clone’s request. The number of jobs is in fact equal to the number of values that the cohort variable takes; in other words, a job is defined in this query to be a multiple of 1000 between 1000 and 10000. Thinking more formally, recall that there is an OPCOND underlying the Display call and its assertion is the each_time assertion. Since the cohort VBL is the only VBL that appears free in both the parallelizing assertion and prior to it and is not an outside VBL relative to the each_time assertion, it is the cohort VBL that defines the jobs.

The sorted_by_spec[1] keyword-argument is used to provide a nice order to the output and probably not coincidentally, to remove any dependence of the order of appearance of the answer tuples on when the various clones finish. Now modify this query so that any reference to parallel execution is removed:

```plaintext
set [ .nbr_clones ] = read( from _cmd_line_ bia[ 5 ] );

with_format _table_
// parallel_for .nbr_clones
sorted_by_spec[1]
do Display each [ .cohort, .sum_over_cohort ]
each_time(
   .cohort Is_In [ 1000 -> 10000 by 1000 ]
   and /*parallelizing*/ ( 
      .sum_over_cohort = sum( over .x 
         each_time( .x Is_In [ .cohort -> .cohort + 999 ] )))
)
```

The answer remains unchanged. So, the claim is that, modulo possible reorderings due to the random production of the answer tuples, the semantics of a Cymbal parallel query is precisely that of the single-threaded query that results when the parallel-specific keywords are commented out. (That being

---

Copyright © 2013 AT&T
All Rights Reserved.
September 15, 2013
said, a few more optional special-case parallel-specific keywords will be introduced subsequently.)

The next query will serve to pin these ideas down further (paraXjobdist.3.Q). It is an artificial query that computes almost the same answers as its predecessors.

```plaintext
set .half_base = 500;
with_format _table_
parallel_for 5
display each [.cohort, .sum_over_cohort]
each_time(
    .base = 2*.half_base and
    .cohort Is_In [.base, .base -> 10*.base by .base]
    and parallelizing(
        .base2 = 2*.half_base and
        .sum_over_cohort = sum( over .x
each_time(.x Is_In [.cohort -> .cohort + .base2-1])
    ))
);
```

The TUPLE of job VBLs is [.cohort] because cohort is the only non-outside VBL that appears free both in the parallelizing assertion and in its preceding fellow conjuncts: observe that the VBL base is local to the first two conjuncts only and that base2 is local to the parallelizing assertion only. The latter is true because a parallelizing assertion implicitly begins with a somehow. (So technically and more accurately, the semantics of a parallelized assertion would be obtained by replacing parallelizing with somehow as well as eliminating all other parallel-specific keywords.) Furthermore, each clone is in effect generating values for its own OPCOND formed from the parallelizing assertion with OPCOND VBLs consisting of all free VBLs of the parallelizing assertion which are scoped declaratively outside of the parallelizing assertion but have their defining/generating occurrences within the parallelizing assertion. In this example, the clone OPCOND VBL TUPLE is [.sum_over_cohort]. And in this example, half_base is just an outside VBL and a procedural one at that.

In addition, note that cohort assumes the value of base twice in this example: this will result in two identical jobs; multiple identical jobs can always be eliminated by taking the extra step of forming a SET (of jobs) to eliminate duplicates, perhaps by using Is_Something_Where. The point is that this is not done automatically and therefore the semantics of job VBLs is that, unless otherwise specified, their values are produced as a LIST{TUPLE}, not SET{TUPLE}. Also, since the sorting specification has been eliminated, the order that the answer TUPLES will be output will now typically vary from one invocation to the next, according to the random order in which the OS runtime environment arranges for the clones finish.

Last but not least is this subtle but important point. All the clones inherit knowledge of the variables in the Cymbal query that would have been known at the point of Daytona processing the parallelizing keyword, had that been a somehow. In other words, those variables are there and accessible in the clones and start out in each clone with the values that they had when the assertion containing parallelizing is asked to generate values for its free variables. This is how it is provable that half_base has value 500 in each clone. Furthermore, since the clones are just being forked with no UNIX exec, the clones have all the files open and usable that their parent did at the time of their creation. Note how
simply these parallel queries arrange for all this information and state to be transferred to the clone child processes.

These high-level parallelization queries are implemented by Daytona rewriting them into lower-level Cymbal using the basic parallelization primitives described in the preceding chapter. Generally speaking, this lower-level Cymbal creates a BUNDLE of clone_TENDRIL(process) of the specified number. The parent/initial process and its clone children engage in conversations whereby each child asks for a job when it has nothing to do and the parent sends the next TUPLE of job VBL values down to the child in response. When the parent runs out of jobs to distribute, it tells the children to terminate. Then the parent gathers and integrates the work of the children. There are variations of this architecture but this is the general idea.

There are currently three contexts in which high-level parallelization directives can be used: Display PROCs, BOXes and for_each_times can all have their OPCOND assertions parallelized. Since parallelized Display PROCs just reuse the BOX parallelization mechanism behind the scenes, these two behave in the same way; in particular, each clone will collect all of its own OPCOND VBL TUPLE values in its own local box which will then be sent back to its parent as a group once it has finished. The parent, of course, just accumulates each clone’s box of answers into its own. Details on the differing implementations for parallel for_each_times are provided later in this chapter.

All three contexts for parallelization support the optional use of a builtin heap-sort merge operation on clone result BOXes; this is invoked by using one of these keywords alongside the use of parallel_for:

( 0->1 ) merging_with_lexico_order,
( 0->1 ) merging_with_reverse_lexico_order,
( 0->1 ) merging_with_sort_spec manifest TUPLE[ ( 1-> ) manifest INT ],

Note that the Daytona implementation causes the results of each clone’s work to be sorted in a BOX using the same sort_spec as specified by the merging_with keyword. Consequently, it would be unnecessary and inefficient for the user to include their own sorting box for this purpose. The advantage of this is that the within-clone \( \Theta(n, \log n) \) sorting of each clone’s result BOX happens in parallel with the others, thus enabling the final \( \Theta(n) \) merge to proceed quickly by the parent. Here’s an example (paraXjobdist.3.Q):

```plaintext
with_format _table_
parallel_for 5
merging_with_sort_spec[1]
do Display each [ .cohort, .sum_over_cohort ]
each_time{
    .cohort Is_In [ 1000 -> 10000 by 1000 ]
    and parallelizing{
        .sum_over_cohort = sum( over .x
            each_time( .x Is_In [ .cohort -> .cohort + 999 ] ))
    }
};
```

This query has the same guaranteed sorted output as that first query, the one that used with_sort_spec[1] instead. The point is that this sorting occurs in a different way. In the first
query, the implicit box used by the parent’s Display had the with_sort_spec[1] sorting criterion that it used to sort the randomly generated tuples of its children, whose result boxes were not sorted. On the other hand, this query using merging_with_sort_spec[1] is implemented so that while the kids’ boxes are sorted, the parent’s box is not sorted by the usual skip-list mechanism, but rather the parent loads its box by doing heap-sort merge and simply puts the output of the merge in the box with no further sorting. Assuming $k$ clones operating on a total of $n$ output records, the with_sort_spec[1] query clearly has to do $\Theta(n \log n)$ sorting work; on the other hand, the merging_with_sort_spec[1] query is faster because, for an ideally distributed work load of integer $m$ equal to $n/k$, the clones are each doing $\Theta(m \log m)$ sorting of smaller problems in parallel whereas the parent is doing $\Theta(n \log k)$ work: $\Theta(n \log k)$ is faster than $\Theta(n \log n)$. And the heap data structure that the parent is using for its merge-sorting is much smaller and faster than the skip-list sorting mechanism used by boxes.

The rest of this chapter will discuss more realistic examples that apply these ideas to record classes. In many cases, these subsequent examples will be seen to be concise, high-level expressions of queries from the preceding chapter, which relied on Cymbal’s low-level parallelization primitives.

20.1 Parallelizing Displays

20.1.1 Parallelizing Displays: Sequential Access

This next example shows how to parallelize sequential access to a presumably large table (paraXjobdist.seq.1.Q).

```sql
set .tot_sects = 15;
parallel_for 5
with_format _table_
merging_with_sort_spec[ -2 ]
do Display each [ .order_nbr, .qty ]
each_time(
  .sect_nbr Is_In [ 1 -> .tot_sects ] and
  parallelizing(
    there_is_an ORDER from_section[ .sect_nbr, .tot_sects ] where(
      Number = .order_nbr and
      Quantity = .qty where( .qty < 100 | > 4900 )
    )
  )
);
```

Since B-tree keyed access is not possible for this query as written, all access to ORDER will be sequential. With this use of from_section, the ORDER table is split into 15 jobs, each consisting of a consecutive sequence of records in the table. The 5 workers or clones get one of these 15 jobs to do each time they ask the parent for work, until the jobs have all been assigned.

If DYNORD, a dynamically horizontally-partitioned table, replaced ORDER above, then, as the
previous chapter discusses, groups of whole DYNORD BINs themselves would form the jobs; this is
known not to be optimal because of the potentially large variation in the sizes of the BINs. Instead, the
next query shows the optimal "some-of-all" way to parallelize sequential access to a horizontally
partitioned record class (paraXjobdist.join.1.Q). This query produces DYNORD information on all
orders of blue or turquoise PARTS.

```sql
set [.nbr_clones] = read( from _cmd_line_ but_if_absent[ 5 ] );
set .tot_sects = .nbr_clones;
parallel_for .nbr_clones
  merging_with_sort_spec[-4, 3]
  with_format _table_
do Display each [ .pno, .ono, .supp, .qty ]
each_time(
  .hparti_box = { [ .region, .category, .uniq ] :
    there_is_a_bin_for DYNORD where(
      Region = .region and Category = .category and Uniquifier = .uniq )}
  and .sect_nbr Is_In [ 1 -> .tot_sects ]
  and parallelizing(
    [ .region, .category, .uniq ] Is_In .hparti_box and
    // sequential access
    there_is_a DYNORD from_section[ .sect_nbr, .tot_sects ] where(
      Region = .region and
      Category = .category and
      Uniquifier = .uniq and
      Number = .ono and
      Supp_Nbr = .sno and
      Part_Nbr = .pno and
      Quantity = .qty
    )
    // Unique_Btree_Key_Fam
    and there_is_a PARTC where(
      Number = .pno and Color = "turquoise" |= "blue"
    )
    // Unique_Btree_Key_Fam
    and there_is_a SUPPLIER where( Number = .sno and Name = .supp )
  );
```

In this case, the jobs are the .sect_nbr or informally, a job is which section of all of the BINs that a
clon is supposed to visit. Technically, .hparti_box is also part of the job specification but since it
has just one value, it functions basically as a constant, as it would also have been, had it been defined
as an outside, procedural variable, that also being acceptable. The point of keeping the definition of
.hparti_box outside of the parallelizing assertion is to ensure that it is computed once in the parent
for all of the clones’ inherited use, not computed redundantly for each child.

Anyway, each clone here sequentially scans a portion of each bin. Hypothetically, if the
from_section phrase were moved to the Is_In on hparti_box, then each clone would process all of a subset of the DYNORD bins, which will not be as well-balanced a division of work in general. This is because for sequential access, it is better for each clone to visit some of all of the bins than it is for it to visit all of some of the bins, the former being the best guarantor that all of the clones finish at basically the same time. Note that there is no need for .tot_sects to be larger than .nbr_clones. Indeed, if it is larger, then the overhead of managing the clones is needlessly increased and furthermore, to the extent that .tot_sects is not an integer multiple of .nbr_clones, then it is likely that a significant number of clones will finish significantly before they all do, thus resulting in inefficient parallelization.

Also, caveat lector, note that if this query is modified in such a way that Daytona decides to use a B-tree index instead of sequential access but the query is still using the some-of-all (sequential) parallelization paradigm as written above, then the query will still produce the correct answers and use the index but there will be NO parallelization: all the work will be done by one clone. In this regard, keep in mind that the using_siz keyword can always be used on a there_isa to force the use of sequential access, which can be faster than B-tree access if the fraction of records producing answers is large enough.

This query also shows one way to parallelize index nested loops joins because indeed the joins with PARTC and SUPPLIER use B-tree indices and all these joins are being done in parallel.

### 20.1.2 Parallelizing Displays: Indexed Access

The next query computes the same answers that its predecessor did but shows yet another way to parallelize index nested loops joins; however instead of using sequential access on DYNORD, it uses a B-tree for indexed/random/direct access to DYNORD (paraXjobdist.join.2.Q).
set [.nbr_clones, .tot_sects] = read( from _cmd_line_ but_if_absent[ 5, 10 ] );
parallel_for .nbr_clones
merging_with_sort_spec[-4, 3]
with_format _table_
do Display each [.pno, .ono, .supp, .qty ]
each_time(
  .hparti_box = [ [.region, .category, .uniq ] :
    there_is_a_bin_for DYNORD where(
      Region = .region and Category = .category and Uniquifier = .uniq ) ]
  and .sect_nbr Is_In [ 1 -> .tot_sects ]
  and parallelizing(
    [.region, .category, .uniq] Is_In
    .hparti_box from_section[ .sect_nbr, .tot_sects ]
  // Unique_Btree_Key_Fam
    and there_is_a PARTC where(
      Number = .pno and Color Is_In [ "turquoise", "blue" ] )
  // NonUnique_Btree_Key_Fam on Part_Nbr
    and there_is_a DYNORD where(
      Region = .region and
      Category = .category and
      Uniquifier = .uniq and
      Number = .ono and
      Supp_Nbr = .sno and
      Part_Nbr = .pno and
      Quantity = .qty
    )
  // Unique_Btree_Key_Fam
    and there_is_a SUPPLIER where( Number = .sno and Name = .supp )
  )
);

See how easy it is to express a radically different parallelization strategy: here, the parallelization is
done by dividing the BINs of DYNORD into groups where each group is a job; then when each clone
gets one of these jobs/groups, it does indexed access on each of the whole BINs in that group. So, one
way to parallelize indexed access is to arrange for each clone to work on all of some of the BINs as
opposed to using the some-of-all strategy recommended for sequential access.

Fortunately, while there is no point in having more groups (.tot_sects) than BINs, the query
will work anyway in that case with some of the groups, hence jobs, not containing any BINs at all.

How to determine the best ratio of jobs to clones as it relates to variability in BIN size and
variability in index selectivity across the BINs is a good question. An attractive answer relies on a
minimax criterion. The idea is to minimize the maximum job cost by creating a sufficiently large
number of jobs. That way, if a clone gets a job involving more work than average, then at least it will
tend to be a smaller amount of work than it would have been if there were fewer total jobs. There are
forces working in the other direction, of course, no doubt including the overhead of accessing/opening
the same BINs more than once, the overhead of doing more job distribution, and probably some effects having to do with I/O block access patterns. Of course, one can always run experiments on particular datasets and queries in order to determine an empirical strategy.

When it comes to computing at runtime the number of clones appropriate for the number of CPUs present and the current load average, Cymbal offers the \texttt{nbr\_online\_cpus()}\ FUN and the \texttt{Get\_Load\_Avg()} PROC. In this regard, the number of idle CPUs is roughly the number of online CPUs minus a current load average. If this number is negative, then the machine is completely busy!

Observe that each of the clones in the preceding query will be accessing PARTC records. This can be made more efficient by caching the information we need from PARTC, assuming that there is enough memory to hold the cache (paraXjobdist.join.3.Q):

\begin{verbatim}
set [.nbr_clones] = read( from _cmd_line_ but_if_absent[ 5 ] );
set .tot_sects = 10;
parallel_for .nbr_clones
merging_with_sort_spec[-4, 3]
with_format _table_
do Display each [.pno, .ono, .supp, .qty ]
each_time(
  .hparti_box = { [.region, .category, .uniq ] :
    there_is_a_bin_for DYNORD where(
      Region = .region and Category = .category and Uniquifier = .uniq ) }
and .pno_box = { .pno: there_is_a PARTC where(
  Number = .pno and Color Is_In [ "turquoise", "blue" ]
  : with_lexico_order }
and .sect_nbr Is_In [ 1 -> .tot_sects ]
and parallelizing(
  [.region, .category, .uniq ] Is_In
  .hparti_box from_section[ .sect_nbr, .tot_sects ]
// NonUnique_Btree_Key_Fam on Part_Nbr
  and there_is_a DYNORD where(Region = .region and
  Category = .category and
  Uniquifier = .uniq and
  Number = .ono and
  Supp_Nbr = .sno and
  Part_Nbr = .pno which Is_In .pno_box in_lexico_order and
  Quantity = .qty
  )
// Unique_Btree_Key_Fam
  and there_is_a SUPPLIER where( Number = .sno and Name = .supp )
  )
);}
\end{verbatim}

The point is that it is the parent alone that processes the PARTC table to create the cache (i.e.,
so that each of the clones does not have to do so (redundantly): remember the clones inherit access to the full state of the parent process at the moment they are created, which occurs when the program flow goes into the parallelizing assertion. Note that each clone uses \texttt{.pno\_box} to access DYNORD BINs using the box-of-key-field-values strategy. The point of sorting the PARTC Numbers is to order the access to the B-tree index: in general, this minimizes B-tree page fetches by in-effect searching the tree from left-to-right, although this only works in general when the ASCII string ordering of B-tree keys is the same as the Cymbal ordering of the corresponding elements of the Cymbal type.

20.1.3 Parallelizing Displays: Partitioning Keys

There is another natural dimension available to parallelize a query using B-tree indexed access to an hparti table. In addition to or instead of parallelizing according to all-of-some of the BINs, a query can also parallelize according to sections of box(es) of keys. In other words, if there is a sufficiently large number of keys to look up to make the effort worthwhile, then for any given BIN, one clone handling one job can be looking up, say, half of them whereas another clone handling another job can be looking up the other half. Here is a query that illustrates jobs doing \textit{both} all-of-some of the BINs and splitting the keys into work units (para\texttt{Xjobdist.join.4.Q}).
set [ .nbr_clones ] = read( from _cmd_line_ but_if_absent[ 5 ] );
set .bin_tot_sects = 10;
set .pno_tot_sects = 7;

parallel_for .nbr_clones
merging_with_sort_spec[-4, 3]
with_format _table_
do Display each [ .pno, .ono, .supp, .qty ]
each_time(
    .hparti_box = { [ .region, .category, .uniq ] :
        there_is_a_bin_for DYNORD where(
            Region = .region and Category = .category and Uniquifier = .uniq ) }
    and .pno_box
        = { .pno: .pno Is_In [ 100 -> 200 ] and .pno % 3 = 1 : with_sort_indices_stored } 
    and .bin_sect_nbr Is_In [ 1 -> .bin_tot_sects ]
    and .pno_sect_nbr Is_In [ 1 -> .pno_tot_sects ]
    and parallelizing(
        [ .region, .category, .uniq ] Is_In
        .hparti_box from_section[ .bin_sect_nbr, .bin_tot_sects ]
    )
    // NonUnique_Btree_Key_Fam on Part_Nbr
    and there_is_a DYNORD where(
        Region = .region and
        Category = .category and
        Uniquifier = .uniq and
        Number = .ono and
        Supp_Nbr = .sno and
        Part_Nbr = .pno which
        Is_In .pno_box from_section[ .pno_sect_nbr, .pno_tot_sects ] in_lexico_order and
        Quantity = .qty
    )
    // Unique_Btree_Key_Fam
    and there_is_a SUPPLIER where( Number = .sno and Name = .supp )
)
);

Of course, one does not have to be so bold as to employ both parallelization strategies at the same time. Consider instead (paraXjobdist.join.5.Q):

Copyright © 2013 AT&T
All Rights Reserved.
September 15, 2013
set [ .nbr_clones ] = read( from _cmd_line_ but_if_absent[ 5 ] );
set .pno_tot_sects = 7;

parallel_for .nbr_clones
merging_with_sort_spec[-4, 3]
with_format _table_
do Display each [ .pno, .ono, .supp, .qty ]
each_time(
    .pno_box = { .pno: .pno Is_In [ 100 -> 200 ] and .pno % 3 = 1 }
    and .pno_sect_nbr Is_In [ 1 -> .pno_tot_sects ]
    and parallelizing( // NonUnique_Btree_Key_Fam on Part_Nbr
        there_is_a DYNORD where( 
            Region = .region and
            Category = .category and
            Uniquifier = .uniq and
            Number = .ono and
            Supp_Nbr = .sno and
            Part_Nbr = .pno
            which Is_In .pno_box from_section [ .pno_sect_nbr, .pno_tot_sects ] and
            Quantity = .qty
        )
        // Unique_Btree_Key_Fam
        and there_is_a SUPPLIER where( Number = .sno and Name = .supp )
    )
);

Each job in this query consists of visiting all of the BINs and looking up only its own portion of the .pno_box of keys. Clearly, one wants the number of .pno_box sections to be larger than the number of clones else some clones will have nothing to do.

20.1.4 Parallelizing Displays: Using Hash Joins

But suppose that there is no Part_Nbr KEY/INDEX for DYNORD. What can be done? Recalling hash joins from Chapter 13, the answer is to use a hash-join (para_hashjoin.1xM.1.Q).
set [.nbr_clones] = read(from _cmd_line_ but_if_absent[5]);
set .tot_sects = 10;
parallel_for .nbr_clones
merging_with_sort_spec[-4, 3]
with_format _table_
do Display each [.pno, .ono, .supp, .qty]
each_time(
   .hparti_box = { [.region, .category, .uniq] :
      there_is_a_bin_for DYNORD where(
         Region = .region and Category = .category and Uniquifier = .uniq )
   } // NonUnique_Btree_Key_Fam on Color
   and .pno_ara = {.pno => _true_ : there_is_a PARTC where(
      Number = .pno and Color Is_In ["turquoise", "blue"])
   )
   and .sect_nbr Is_In [1 -> .tot_sects]
   and parallelizing(
      [.region, .category, .uniq] Is_In .hparti_box and
      // sequential access
      there_is_a DYNORD from_section[.sect_nbr, .tot_sects]
      where(
         Region = .region and
         Category = .category and
         Uniquifier = .uniq and
         Number = .ono and
         Supp_Nbr = .sno and
         Part_Nbr = .pno where(.pno_ara[.pno] = ?) and
         Quantity = .qty
      )
      // Unique_Btree_Key_Fam
      and there_is_a SUPPLIER where(Number = .sno and Name = .supp)
   )
)

The reason why this is a hash-join between PARTC and DYNORD is because the PARTC information is read into a hash-table (dynara) and then the DYNORD records are read sequentially and joined against the in-memory PARTC information. In general, a hash-join has to decompose the work into multiple hash buckets in order to adapt to not being able to put all of the first join table information into memory -- so, this is just a special case but nonetheless, useful when feasible (see Chapter 13). Note the parent's caching of .pno_ara as was done previously with .pno_box.

The previous hash-join example was able to use a dynara for PARTC because the relationship between DYNORD and PARTC on Part_Nbr is many-to-one. In one possible many-to-many situation, as occurs when joining DYNORD with itself on Part_Nbr, it is necessary to use a BOX to hold the results of the scan of the first/outer relation so that any particular join attribute(s) value(s) from the second/inner table can be associated with multiple TUPLES of values stored in the BOX for the first table. The next query performs such a many-to-many hash-join (para_hashjoin.MxM.1.Q); notice that
there are two parallelizations that occur: one for the creation of the in-memory SET of TUPLES that contains information extracted from the outer table and the other for doing the join itself.

```sql
set .tot_sects = 10;
merging_with_sort_spec[ 1, 2, 4 ]
parallel_for 5
do Display each [ .pno, .sno1, .qty1, .sno2, .qty2 ]
each_time(
  .supplies_part = { [ .sno, .pno, .qty ] :
    .sect_nbr Is_In [ 1 -> .tot_sects ]
    and parallelizing(
      // sequential access here
      there_is_a DYNORD from_section[ .sect_nbr, .tot_sects ]
      where( Supp_Nbr = .sno and Part_Nbr = .pno and Quantity = .qty )
    ) : with_sort_spec[2,1,3] with_init_max_nbr_elts 2000 parallel_for 5 }
and
  .sect_nbr2 Is_In [ 1 -> .tot_sects ]
and
  parallelizing(
    // sequential access here
    there_is_a DYNORD from_section[ .sect_nbr2, .tot_sects ]
    where( Supp_Nbr = .sno2 and Part_Nbr = .pno and Quantity = .qty2 )
    and [ .sno1, .pno, .qty1 ] Is_In .supplies_part sorted_by_spec[2,1,3]
    and .sno1 < .sno2
  )
);```

(Strictly speaking, this is not a hash join because BOXes are not implemented by using hashing but rather by using skip lists. Nonetheless, this is similar in spirit to a hash join, differing only in the algorithm used to find the matching TUPLES in the first table and in the fact that more than one TUPLE for the first table is typically found for each matching record in the second table. Indeed, hash tables usually don’t support mapping a value to many values.)

It should be apparent now that Cymbal’s high-level parallelization feature is remarkably flexible with very little effort, which is good because there are in fact many, many ways to parallelize computation.

### 20.1.5 Parallelizing Displays: Path Preds

Path PREDs can also be computed in parallel as shown here (paraXjobdist.trans.1.Q):

Copyright © 2013 AT&T
All Rights Reserved.
September 15, 2013
with_format _table_
parallel_for 4
merging_with_lexico_order
with_no_duplicates
do Display each [ .desc, .yob ]
each_time{
    "William I" Is_A_Royal_Parent_Of .x
    and parallelizing(
        .desc Is_A_Royal_Descendant_Of .x
        and if( there_isa ROYAL where( Name = .desc ) )
        then( this_isa ROYAL where(
            Year_Born = .tyob where( .yob = (STR) .tyob )
            but_if_absent( .yob = "not-recorded" )))
        else( .yob = "not-a-royal" )
    )
};

The idea is to compute an initial set of nodes, in this case the children of William I, and let those be the jobs.

As a general rule, it’s wise not to use too fine a grain in defining the jobs, i.e., to have too many as in thousands of jobs. That’s because there is substantial overhead in doing job distribution over pipes. To avoid that, multitudinous would-be jobs can be accumulated in sections of a box, which are then cleverly handed over to the clones as units by using from_section on the box: so, this requires that the box of (little) jobs be defined outside of the parallelizing assertion so that all that needs to be passed to each clone is what section of that box should be considered its next (big) job.

### 20.2 Parallelizing Boxes

The strategies used in parallelizing Displays can also be used in parallelizing BOXes. Here is an example of building a BOX in parallel (paraXjobdist.box.3.d.Q), where the jobs are TUPLEs of values [ .region, .category ], implying one BIN per job. The access to each BIN is indexed via box-of-key-field-values: notice that .pno_box is a procedural VBL in this example.
In light of the preceding parallelized Display examples, this example should be readily understandable because it uses the same principles. Indeed, as remarked earlier, parallelized Displays are quietly implemented by using parallelized boxes. However, this example goes further and shows that it is perfectly acceptable for the BOX being constructed in parallel to have multiple sort specifications. In this case, since it is defined with merging_with_reverse_lexico_order, the selection order for the (parent) box is clearly reverse_lexico_order since that’s the order that the heap-sort merge presents the box.
elements for inclusion in the box. Note that merging with reverse lexico order as well as the other parallel-related specifications are not considered to be part of the type of the BOX although they are placed in along with the other box keywords in defining/declaring boxes; consequently, they are not included in local/import/export specifications nor in fpp parameter BOX VBL definitions.

The example above parallelizes the construction of the value of a procedural BOX VBL. paraXjobdist.box.3.b.Q shows that the other forms of BOX construction can also be parallelized, as is illustrated by this parallelized Is The Next Where, obviously a declarative BOX:

```plaintext
set .pno_box = { .pno : .pno Is_In [ 100 -> 200 by 10 ] );

fet [.pno, .ono, . supp, . qty ] Is_The_Next_Where( there_is_a_bin_for ORDERA where( Region = .region and Category = .category )
and
parallelizing{
    there_is_a ORDERA where( Region = .region and
    Category = .category and
    Number = .ono and
    Supp_Nbr = .sno and
    Part_Nbr Is_In .pno_box and
    Part_Nbr = .pno and
    Quantity = .qty
}
    and there_is_a SUPPLIER where( Number = .sno and Name = . supp )
}

merging_with_lexico_order  parallel_for 11
{ do Write_Words(.pno, .ono, . supp, . qty); }
```

### 20.3 Parallelizing for_each_time

For starters, for_each_times can be parallelized in the same two basic ways that boxes and Displays can, i.e., with and without using the heap-sort merge. However, they offer additional functionality over and above that.

The first such capability is just the ability to specify exactly what is to be done with each of the answer tuples (paraXjobdist.fet.1.e.Q):
set .pno_box = { .pno : there_is_a PARTC
  where( Number = .pno and Color Matches "e|i" ) };
set .hparti_box = { [ .region, .category ] :
  there_is_a_bin_for ORDERA
  where( Region = .region and Category = .category )
  : with_random_indices_stored };
set .tot_pno_sects = 2;
set .tot_hparti_sects = 4;
parallel_for .nbr_clones
merging_with_sort_spec[1, 4]
fet [ .part, .ono, .supp, .qty ] ist(
  .pno_sect_nbr Is_In [ 1 -> .tot_pno_sects ] and
  .hparti_sect_nbr Is_In [ 1 -> .tot_hparti_sects ] and
  parallelizing(
    .pno Is_In .pno_box from_section[ .pno_sect_nbr, .tot_pno_sects ]
    and [ .region, .category ] Is_In .hparti_box in_random_order
    from_section[ .hparti_sect_nbr, .tot_hparti_sects ]
    and there_is_a ORDERA where(
      Region = .region and
      Category = .category and
      Number = .ono and
      Supp_Nbr = .sno and
      Part_Nbr Is_In .pno_box and
      Part_Nbr = .pno and
      Quantity = .qty
    )
    and there_is_a SUPPLIYER where( Number = .sno and Name = .supp )
    and there_is_a PARTC where( Number = .pno and Name = .part )
  )
)
before_doing_the_first{ set .tup_cnt = 0; }
after_doing_the_last{ _Show_Exp_To(.tup_cnt) }
renewing_with{ set .tup_cnt++; }
do { // arbitrary answer-handling logic here.
  do Write_Words(
    ".qty quantity of `.part' ordered from ".supp" on PO# .ono"ISTR ");
}
else { with_msg "where’s the data?" do Exit(1); }

Happily, this query illustrates other ideas as well. Notice that this query defines the jobs to be a cross-
product between sections of a set of part numbers to look up (via index) and sections of the list of
ORDERA BINs. The point is that Daytona offers great flexibility in defining jobs. Secondly, the
.hparti_box has been randomly ordered so as to probabilistically remove any difference in the
sections of the BINs due to their rcd-defined order. (For example, this is helpful in randomizing BINs ordered by TIME when the older BINs tend to be smaller (or larger).) Finally, note that it is the parent that not only does the do but also does all related do-actions, i.e., any before_doing_the_first, after_doing_the_last, renewing_with and else.

Consider the next query, largely the same as before, where there is no merging specification (paraXjobdist.fet.1.f.Q):

```plaintext
set .pno_box = { .pno : there_isa PARTC
     where( Number = .pno and Color Matches "e|i" ) };
set .hparti_box = { [ .region, .category ] :
     there_is_a_bin_for ORDERA
     where( Region = .region and Category = .category )
     : with_random_indices_stored };
set .tot_pno_sects = 2;
set .tot_hparti_sects = 4;
parallel_for 5
fet [.part, .ono, .supp, .qty ] ist(
   .pno_sect_nbr Is_In [ 1 -> .tot_pno_sects ] and
   .hparti_sect_nbr Is_In [ 1 -> .tot_hparti_sects ] and
   parallelizing(
      .pno Is_In .pno_box from_section[ .pno_sect_nbr, .tot_pno_sects ]
      and [ .region, .category ] Is_In .hparti_box in_random_order
      from_section[ .hparti_sect_nbr, .tot_hparti_sects ]
      and there_is_a ORDERA where(
         Region = .region and
         Category = .category and
         Number = .ono and
         Supp_Nbr = .sno and
         Part_Nbr Is_In .pno_box and
         Part_Nbr = .pno and
         Quantity = .qty
      )
      and there_is_a SUPPLIER where( Number = .sno and Name = .supp )
      and there_is_a PARTC where( Number = .pno and Name = .part )
   )
) do {
   do Write_Words( .part, .ono, .supp, .qty );
}
```

This query is not going to sort its results and if the user tried to get it to sort them by specifying sorted_by_spec or the like, Daytona would not accept it. The reason is because when there is no merge-sort specification, then the parallelized for_each_time will not be building boxes, and therefore it cannot accept box keywords. This implies something very important about the non-merge-sort parallelized
for each times, i.e., that their do portion is executed as soon as the system is able to come up with an answer tuple from one of the clones. In other words, there is no accumulating of answers in a box before any one answer can be used (which is the nature of boxes). This means that the parent can be processing the results of the clones as soon as they have enough to flush their buffers and send them back over the pipes. It also suggests (correctly) that some other process must be feeding jobs to the clones. And since all the clones can be writing back answers to the parent at the same time, it also suggests correctly that Daytona is being clever about keeping them from trashing each other’s results.

The downside of this architecture, which may not be an issue for any given query, is that the volume of answers can grow to the point where the parent is 100% busy and becomes the bottleneck, although once again, the parent has been working on the answers all the time, i.e., not waiting for each clone to finish and dump a results box on the parent in one fell swoop. Also in fairness, in extremis, there is always a bottleneck somewhere. On the other hand, being aware of the architecture enables the user to write queries that delay the appearance of bottlenecks; in this case, suppose that all the parent was doing was accumulating counts sent across by the clones: clearly, efficiency would be improved if each clone could summarize its results periodically and send those summaries over to the parent instead of all the detail data. The parent can then summarize the clone summaries.

But suppose the clones didn’t have to write back all their results to the parent. Then this issue would not arise at all. Of course, the parent would no longer integrate clone results but it may be sufficient to just have the clones write all their output to a common file (which could be processed subsequently) or to a FIFO for input into another process -- or each clone could write to its own file/FIFO. Anyway, this is possible by using the with clones doing the do keyword, which lives up to its name, as this next artificial but instructive example shows (paraXjobdist.tup_cnt.1.Q).
local: INT .clone_tup_cnt
set .overall_tup_cnt = 0;
with_clones_doing_the_do
parallel_for 2
fet .i
ist( .a =1 | = 2
and parallelizing(
   .i Is_In [ 1 -> 5 ] from_section[ .a, 2 ]
))
before_doing_the_first{ set .tup_cnt = 0; }
after_doing_the_last{ _Show_Exp_To(.tup_cnt)
   flushing to _parent_ do Write_Line( .tup_cnt );
} // each clone does this!
renewing_with{ set .tup_cnt++; }
do {
   _Show_Exp_To(.i)
}
wrapping_up_each_clone_with {
   set [ .clone_tup_cnt ] = read( from _child_ upto "\n"
       otherwise { with_msg "failed_to_read_tup_cnt" do Exit(32); }
   set .overall_tup_cnt += .clone_tup_cnt;
}
else {
   with_msg "where’s the data?" do Exit(3)
}
// REMEMBER: it’s a race: one clone could get both jobs!
_Show_Exp_To(.overall_tup_cnt)

Here is the output:

.i = 1
.i = 2
.i = 3
.tup_cnt = 3
.i = 4
.i = 5
.tup_cnt = 2
.overall_tup_cnt = 5

This architecture highlights the fact that for all of the parallelized for_each_times, the for_each_time parallelizing assertion belongs to the clones, i.e., their job is to find all ways to satisfy it. However, if not using with_clones_doing_the_do, all the rest of the for_each_time, i.e., the procedural part, is done by the parent.

On the other hand, when using with_clones_doing_the_do, all of the procedural specifications are
handled by the clones with the exception of the optional wrapping up each clone with do-group and the optional else do-group, those being done by the parent. Therefore, when with clones doing the do is specified, the clones will do any before doing the first, after doing the last, and renewing with specifications along with of course, the do itself. By using wrapping up each clone with, the user can arrange for the parent to engage in a final conversation with each clone (by making use of the special _child_ symbolic constant). The reason why the else do-group is done by the parent is because it contains the logic to handle the situation where the clones as a group have nothing at all to do.

Here is a with clones doing the do query where all the clones simultaneously write their output into the same file (paraXjobdist.fet.3.c.Q). There is no corruption because that file is opened in append mode.
local: INT .clone_tup_cnt

set [ .nbr_clones ] = read( from _cmd_line_ but_if_absent[ 5 ] );
set .ochan = new_channel( for "./DS_TMPFL" with_mode _clean_slate_append_ )
otherwise { do Exit(1); }
set .tot_sects = 10;
set .pno_box = { [ .pno ] : there_isa PART where( Number = .pno )
: with_lexico_order );
set .hparti_box = { [ .region, .category ] : there_is_a_bin_for ORDERA
where( Region = .region and Category = .category ) );

parallel_for .nbr_clones
with_clones_doing_the_do
fet [ .pno, .ono, .supp, .qty ] ist(
.sect_nbr Is_In [ 1 -> .tot_sects ] and
parallelizing( [ .region, .category ] Is_In .hparti_box
from_section[ .sect_nbr, .tot_sects ] and
there_is_a ORDERA where(
Region = .region and
Category = .category and
Number = .ono and
Supp_Nbr = .sno and
Part_Nbr Is_In .pno_box and
Part_Nbr = .pno and
Quantity = .qty
)
and there_is_a SUPPLIER where( Number = .sno and Name = .supp )
))
before_doing_the_first{ set .tup_cnt = 0; }
after_doing_the_last{ flushing to _parent_ do Write_Line( .tup_cnt ); }
renewing_with{ set .tup_cnt++; }
do { to .ochan with_sep "|" do Write_Line( .pno, .ono, .supp, .qty ); }
wrapping_up_each_clone_with{
set [ .clone_tup_cnt ] = read( from _child_ upto "\n" )
otherwise { with_msg "failed_to_read_tup_cnt" do Exit(32); }
set .tup_cnt += .clone_tup_cnt;
}
else { with_msg "where’s the data?" do Exit( 3 ); }
do Write_Line( "Total tuples over all .nbr_clones clones = .tup_cnt"ISTR );
do Write_Line( shell_eval("wc -l ./DS_TMPFL" ) );
set ? = shell_exec( "$DS_DIR/Check_DC_Lines -v ./DS_TMPFL" );

Note once again that the clones not only automatically know about .ochan in some abstract sense but
they are all able to write to it. There is no confusion or collision in writing to the file because Daytona
does the Writes by appending "whole messages" to the file.

20.4 Early termination for parallelized queries

There are situations where the goal of a parallelized query is accomplished before all the jobs have been distributed. For example, the goal may be to find at most 1000 answer tuples. In that case, as soon as that occurs, it is pointless to continue with distributing jobs to the clones to do, rather instead the parallelization should just stop and the query should continue to process those 1000 answer tuples. Fortunately, Cymbal provides the means to terminate queries early when using any of its easy parallelization paradigms.

When parallelizing BOXes, early termination is accomplished simply by using a stopping_when assertion on the selection index, just as is done with unparallelized BOXes (paraXjobdist.stop.1.Q):

```plaintext
fet [ .reg, .cat, .number, .supp_nbr, .part_nbr, .date_placed, .quantity]
    ist{
        [ .reg, .cat, .number, .supp_nbr, .part_nbr, .date_placed, .quantity]
        Is_The_Next_Where{
            .reg Is_In [1, 2, 3, 11, 13 ] and .cat Is_In [ "A", "B", "C" ] and
            parallelizing{
                there_isa ORDERA where{
                    Region = .reg and
                    Category = .cat and
                    Number = .number and
                    Supp_Nbr = .supp_nbr and
                    Part_Nbr = .part_nbr and
                    Date_Placed = .date_placed and
                    Quantity = .quantity
                }}}
    parallel_for 4
    in_lexico_order
    with_selection_index_vbl si
    stopping_when (.si >= 50 )
    }
    do Write_Words{
        .reg, .cat, .number, .supp_nbr, .part_nbr, .date_placed, .quantity);}
```

This same construction with stopping_when can be used with the likes of merging_with_lexico_order but the time savings will be minimal because with the merging paradigm, the clones first get all possible jobs they can get so that they can sort all possible answer tuples before sending them to the parent, which will then stop accepting them when and if the limit is reached.

As for terminating any of the three kinds of for_each_time loops early, the idea is for "someone" to tell the process generating jobs to stop doing so, in which event, the clones will then run out of work to do and terminate normally. In fact, that someone determining that it is time to terminate early is either
the parent or one of the clones according to some user-supplied criterion in the query.

So, once again, when for whatever reason, there are no more jobs to distribute, the clones will finish the jobs they are working on and the processing of the for each time will proceed to completion in a completely normal fashion. Daytona considers this to be preferrable to sending (asynchronous) signals to the clones for them to catch, stop whatever job they are working on at whatever point they are in it, and try to exit without making a mess.

So, how does a determining party tell the job distributor to stop? First, to enable this action, all jobs must be put into a BOX (i.e., a SET or LIST) as stored in a procedural BOX VBL. Then when the decision is made to terminate, the determining party simply gets a share fcntl lock on that job BOX. And then when the presence of that lock is detected by the process doing the job distribution (at the point when it is about to send down the next job), it simply stops generating jobs, while leaving the BOX itself unchanged. And then the clones finish up whatever job they are working on and stop their work. Here from paraXjobdist.stop.2.Q is a contrived example:

```plaintext
local: LIST[ INT : act_empty_when_locked_out ] .job_box

set .job_box = [ 1 -> 50 :: act_empty_when_locked_out ];

parallel_for 3
fet [ .x, .y ]

ist(
   .x Is_In .job_box
   and parallelizing(
      .y = .x * .x )
)

when( .x > 12 ) do Lock_Box( .job_box );
else {
   do Write_Words( .x, .y );
}
```

Since the parent is doing the do of the for each time here, when the parent observes that it has collected 12 answers, it issues the Lock_Box call, the clones finish any job they are currently working on, and the processing of the for each time loop comes to a close -- as there are no more jobs being generated. Note that a Lock_Box call will not be effective unless the job BOX has been defined with an act_empty_when_locked_out keyword. If the query would like to continue after the given for each time and use the job box again, it is necessary to unlock it by calling the likes of Unlock_Box( .job_box ). Note that in addition to the efficiency of this approach, i.e., basically the same number of fcntl calls as jobs, when the processes exit, the lock(s) disappear automatically as well, whereas if semaphores were used, for example, then they would need to be explicitly removed at some point.

Here is a much more realistic example that makes a number of useful observations; also from paraXjobdist.stop.2.Q:
In this query, the jobs are SUPPLIER Numbers, each of which is associated with circa 10 ORDERs. When this query is run multiple times, it has been observed to produce 41, 73, 79, 87 answers. Why so much larger than 12? The reason is that when a clone sends answers back to the parent, it does it by means of buffer flushes to a CHAN(funnel). The parent on the other hand is reading the contents of the funnel_CHAN, one answer at a time. When it reaches 12, it calls Lock_Box. It should be clear that the entire contents of the funnel_CHAN is going to be read and processed before the query is over and that in addition to that, so also all of the answers that correspond to the jobs that the clones are currently working on. Obviously, if the query typically overruns the limit by 10%, then reducing the limit by 10% would tend to produce the desired behavior. The variability is due to the races among the clones. The use of box_locked is intended to ensure that Lock_Box is only called once; it’s actually unnecessary in that it is OK to call Lock_Box more than once.

However, it’s not necessary for the parent to print out the answers beyond the limit:
What is important is that the parent just let the clones run out of gas (while creating unwanted output) and then terminate quietly and normally.

Here is an example with clones doing the do:
{ local: LIST[ INT : act_empty_when_locked_out ] .job_box 
set .job_box = [ 1 -> 50 :: act_empty_when_locked_out ];
parallel_for 3 
with_clones_doing_the_do 
fet [ .x, .y ]
  ist(
    .x Is_In .job_box 
    and parallelizing(
      .y = .x * .x )
  )
  when( .y > 144 ) do Lock_Box( .job_box );
  else {
    do Write_Words( .x, .y );
  }
}
}

The effect of this is for each of the clones to be on the look out for when its own criterion for stopping is reached, at which point it locks the job BOX causing job distribution to cease.

There is nothing wrong with using the Lock_Box paradigm with the merging keywords but the same efficiency proviso for the answer BOX case applies here as well.

This paradigm appears to be fully general since by using Cymbal TICKET_BUNCHes (semaphores), any subset of the clones can coordinate amongst themselves to request early termination. For example, if for some reason, the goal of the parallelized for each time is for at least five of the clones to encounter an event, then by using a "counting semaphore", the clones can keep track of the number of those events found so far and when that limit is reached, the clone that notices that last event can tell the job distributor to stop. In other words, any number of clones from one to all of them can determine when it is time for early termination to occur. And since the parent also has the option to request early termination, that makes this capability fully general.
21. Shared Memory In Cymbal

21.1 Using Shared Memory In Cymbal

Objects placed in shared memory are shareable by other processes (subject to a locking protocol) and are persistent (as long as the machine stays up). The two main reasons for using shared memory are application speed-up due to working with in-memory data instead of reading data in from disk and reducing an application’s footprint on memory by enabling cooperating processes to share data instead of maintaining their own private copies. In regards to the former, in one particular instance, reading dynara elements from shared memory was 273 times faster than reading them from disk.

This chapter presents two quite different paradigms for using shared memory. The first is based on a transaction model where all locks are (typically) gotten at the start of a transaction and released at the end. The second is based on the properties of individual statements and do-groups (i.e., not big transactions) and so, as a result of new and elaborate algorithms, is much more concurrent. In fact, readers never get locks of any kind and writers almost never conflict in getting their locks (unless electing to use an optional construct that ensures a higher level of consistency).

Any Cymbal program making use of shared memory has to be written with one paramount thought in mind: the VBLs in shared memory are something else again since they are "over there" in another space, that space being shared memory; consequently, they have to be reached by VBLs that are local to the process running the program, i.e., process-local VBLs. That is why the process-local VBLs used to access shared memory are all VBL VBLs whose values are shared memory VBLs. For example, suppose INT .x is a Cymbal shared-memory-pointing VBL, shmem VBL VBL for short. This means that x is a process-local VBL (belonging to the Cymbal program’s UNIX process), so that .x, which has type INT VBL, is an (unnamed) VBL over in shared memory, whose value .x is an INT. So, the shmem VBL VBL here (local to the process) is x and the shmem VBL (over there in shared memory) is .x.

Just one new keyword, from shmem, is needed to introduce shared memory capabilities into Cymbal, as the following pedagogical example illustrates:
define PROC task: Do_Something
{
  import: INT .a
  local: INT .b = 1
  export from_shmem "bobs_shm_region" : INT ARRAY [ 10 ] ..xx
  export from_shmem "my_shm_region" : FLT ARRAY [ INT ] ..ara
  import from_shmem "anns_shm_region" : TUPLE [ INT, FLT ] ..tup
     BOOL .now_is_a_good_time
  set .b = ..tup#1 + ..xx[5];
  when( .now_is_a_good_time ) set .a = ..xx[3] + .b;
  set ..ara[ .a ] = ..ara[ .b ] + ..tup#2;
  // OK if txn task:
  // do Change so_that( there_is_a WIDGET where( Number = ..ara[ .a ] ) );
}

The constructs used in this example are now discussed.

All shared memory access must be initiated from within fpp tasks or transaction tasks. If the former, then the user has the responsibility to provide explicit synchronization code, perhaps by using TICKET_BUNCHes, because in the absence of inter-process synchronization, shared memory use will simply fail catastrophically when writers are present.

Note that the first half of this chapter deals with the default mode of Cymbal shared memory programming whereas the second half discusses the "concurrent" one, which is a variant of the first. Much of the discussion about the first applies to the second; the presentation of the second will make clear the differences. So, now for more on the first, the default.

If using a txn task, then Daytona will automatically provide read/write locking of individual shmem VBLs, without ruling out the unlikely, careful, and informed use of additional user-provided explicit synchronization mechanisms for other purposes. Such a txn task can of course also be used for regular read/write table accesses, with the understanding that currently only changes to tables can be rolled back, there being no Do_Queue for shmem VBL changes. The nature of txn task synchronization is that all the share/exclusive locks are gotten at the start of the txn and released on exit. All helper fpps in a txn task have free and safe access to the txn task's shmem VBLs.

One useful but risky tactic involves writing a txn task that calls a non-txn task that imports and accesses at most the same shmem VBLs that its caller does -- and no others. This called, non-txn task can avoid its obligation to do its own synchronization if it behaves as follows. Since the caller has already gotten at least share/read locks on its shmem VBLs, the called routine is free to read from these shmem VBLs without (redundantly) doing its own synchronization. It can also safely modify any one of those VBLs but only if its caller is also modifying it (which would have then caused Daytona to get an exclusive/write lock on it at the start of the caller): otherwise the called non-txn task is a naked threat to any other processes using the shmem VBLs in question (shm.4.e.IQ). However, in any case, for certain technical reasons serving to preserve the safety of txns, Daytona’s implementation forbids using any shmem dynara with an @-default in non-txn tasks in any setting.

Shmem VBL VBLs are procedural VBLs that are defined or declared by corresponding export or import statements using the from_shmem keyword. The argument to from_shmem is just a user-specified
THING name for a shared memory region, that region corresponding either to one UNIX shared memory segment or to more, if Daytona happens to link them together in a chain behind the scenes, a mechanism which is not yet implemented. Recall that by default, a UNIX shared memory segment is created tabula rasa before any process can use it and it persists even after all processes using it have exited. Consequently, for a given process, a shared memory segment may contain shmem VBLs and their values prior to that process attaching to it to use those VBLs and values.

In this context then, when a shmem VBL VBL is exported, Daytona will create the corresponding shmem VBL on entry to the associated task, if it does not already exist, whereas if it does exist, the task will continue to execute and provide access to that shmem VBL. Consequently, there is no problem with multiple concurrent processes exporting the same shmem VBL. And when the enclosing task returns and hence when the process exits, the shmem VBL will be left alone to persist.

On the other hand, when a shmem VBL VBL is imported, then on entry to the associated task, the task will use the indicated shmem VBL if it is present in the shared memory region and if it does not exist, then the process will exit with an error message. Not surprisingly, on return from the enclosing task, imported shmem VBLs will be left alone to persist.

In any case, given the persistence of shared memory, the user must take care not to somehow use the same name to refer to shmem VBL VBLs that have different types as that will surely lead to severe problems including segmentation violations.

Note that computational statements can freely mix read/write access to both process-local and shmem VBL VBLs. However, since as a rule, little good can come from referring to the shmem VBL itself alone, e.g., .x for shmem VBL VBL x, such references are considered error conditions, so as to eliminate the possibility of writing .x when meaning to write ..x . Note that this stricture does not apply though to expressions involving .x such as .x[4] where the intent is to refer to a SHMEM_VBL that is part of another SHMEM_VBL.

Recall that txn tasks using shmem VBL VBLs get locks, which if they need to be exclusive, will cause other processes seeking to use those shmem VBLs to block waiting for access. Consequently, in the interests of efficient cooperation, users should probably write their shmem txn tasks so that after attaching to shared memory, they do their shmem-based computations expeditiously, then copy what they need for longer-term cogitation to process-local but exported VBLs, and then quickly return from the txn task, so as to release the locks and allow other tasks to use the gathered process-local information.

By default, Cymbal shmem transactions detach from their shmem regions when they return. The operating system detachment operation can be unacceptably time-consuming for large shmem regions. However, as an option, shmem transactions can remain attached to their segments after returning by using the stay_attached txn keyword as in:

\[
\text{define PROC stay\_attached txn task: Update\_Shmem\_Region}
\]

While it is possible to share dynara, it is not currently possible to share BOXes or for that matter, VBLs of any other type. This is not as restrictive as it may seem because, for example, in order to store an INT or FLT or other scalar type (or even a TUPLE) in shmem, it suffices to share a dynara that maps STR names to objects of that type.
Also, keep in mind that when defined with the @-default option, a dynara that syntactically looks to Tracy like it is only being used for reading may in fact write to shared memory at runtime when asked to retrieve an element that is not already there: in that case, its share lock on the dynara will be promoted to exclusive. This introduces the possibility of deadlock but that is considered to be preferrable to the downside of doing unprotected writes. (Deadlock would occur if two txns each share locking dynara abc and def decide to exclusively lock abc and def, respectively.) However, in order to reduce the possibility of deadlock and to enable this txn to continue on as a reader, as soon as the new element is added, the exclusive lock on the dynara is downgraded to a share lock and all sharing txns after that will be able to see the new element. Of course, to get the exclusive lock on the dynara, the requesting process must (possibly wait to) lock everyone else out, which may take awhile but it will only hold the exclusive lock for a negligible amount of time before it downgrades it to a share lock.

As always, shm mem txns are used to keep reading and writing from colliding. In fact, regarding the ACID properties characterizing well-behaved transactions in the database literature, by virtue of its use of (two-phase) locking of entire dynara, Daytona supports Consistency and Isolation. In particular, shm mem txns are serializable and so, there is no possibility of the canonical pathologies of lost updates, unrepeatable reads, reading uncommitted data, incorrect aggregation, and phantom records. On the other hand, Atomicity is not supported since Daytona is not creating Do_Queues for shared memory operations and so rollback/undo is not supported by the system. Durability is clearly not supported since a system crash (or data corruption) will cause shared memory segments to be lost and since Daytona is not creating any Do_Queues, there will be no logging of Do_Queues to disk, etc. In theory, nothing rules out implementing and using Do_Queues, understanding that their mere existence implies compromising the speed desideratum while on the plus side, providing Atomicity when the shared memory region itself persists; however, if Durability is also required, then the speed objective takes an even greater hit when disks get involved.

### 21.2 Creating And Administering Shared Memory In Cymbal

The Cymbal transaction task Create_Shmem_As_Needed, which is defined in $DS_DIR/shmem_pkg.cy, is used to create shared memory regions.

```cy
define PROC(
    THING .shmem,
    with_size_in_k INT(_off_t_) .size_in_k,
    with_user_perm INT .u_perm = _update_, /* _read_, _update_, _none_* */
    with_group_perm INT .g_perm = _read_,
    with_others_perm INT .o_perm = _none_,
    with_owner_uid INT .uid = _no_change_,
    with_owner_gid INT .gid = _no_change_,
    with_cushion_in_MB INT .cushion_in_MB = 0,
    locked_in_core BOOLEAN .lic_flg = _false_
) txn task: Create_Shmem_As_Needed
```

The required options are the name of the shm em as captured by a THING and the size in KB. The permissions should be self-explanatory. Shared memory has a creator, whose uid and gid are immutable, as well as an “owner”: the permissions apply to both. Obviously, the idea is to create the
shmemb region once and use it repeatedly by other processes in addition to the creating one. To that end, once created, calling this routine again for the same shmemb region has no effect.

And once created, a shmemb region can be described or destroyed by using:

```
PROC( THING .shmemb, BOOL .vmstats ) task: Describe_Shmemb
```

```
PROC( THING .shmemb ) txn task: Destroy_Shmemb
```

Note that Describe_Shmemb will print out much more information if its shmemb region specified by its argument is currently attached when the routine is called. If .vmstats is true, then Describe_Shmemb will print out vmalloc statistics for the region (if it is attached when Describe_Shmemb is called). To provide this additional output, Daytona must lock the vmalloc portion of the region and therefore, lock out all other vmalloc activity by other processes for as long as it takes to traverse vmalloc data structures and compute the requested statistics. This can take minutes in the worst case and during that time, all other processing of the region that requires the use of malloc/realloc/free will simply stop. That’s why this is an option.

There is also an Attach_Shmemb but the system automatically calls it on entry to a from_shmemb task and there is also a matching Detach_Shmemb, which the system automatically calls on exit from a from_shmemb task, unless the stay_attached keyword is used. So, as a rule, the user should not call either one.

Permissions and owner uid/gid can be changed dynamically by:

```
define PROC(  
    THING .shmemb,  
    with_user_perm INT .u_perm = _no_change_, // _read_, _update_, _none_  
    with_group_perm INT .g_perm = _no_change_,  
    with_others_perm INT .o_perm = _no_change_,  
    with_owner_uid INT .uid = _no_change_,  
    with_owner_gid INT .gid = _no_change_  
) task: Change_UGO_Perm_Owner_Uid_Gid_For_Shmemb
```

### 21.3 Using Shared Memory In Cymbal

Here is a simple program creating a shmemb region for others to use by reading information out of /etc/passwd (shm.4.Q):
import: package ¨shmem_pkg¨
    with_size_in_k 256
    do Create_Shmem_As_Needed( ¨pinfo¨ );
do Describe_Shmem( ¨pinfo¨ ); // some output may be cryptic
do Publish_Passwd_Info;

global_defs:
define PROC txn task: Publish_Passwd_Info
{
export from_shmem ¨pinfo¨ :
    TUPLE[ INT .uid, INT .gid, STR .comment ] ARRAY[ STR .login
        : with_default @ => [ -1, -1, "Nonesuch" ] ] ..passwd_info
fet [ STR .login, INT .uid, INT .gid, STR .comment ]
ist( [ .login, ?, .uid, .gid, .comment, 2? ]
    = ftokens( for "/etc/passwd" upto ":\n" )
})
    set ..passwd_info[ .login ] = [ .uid, .gid, .comment ];
}
}

Note the required import of a certain ¨shmem_pkg¨ which in fact contains the PROCs Create_Shmem_As_Needed, etc. mentioned above.

And here is a simple program that gets passwd information out the shared memory region created by the preceding task (shm.4.b3.IQ).

import: package ¨shmem_pkg¨

set [ .login ] = read( from _cmd_line_ bia[ "root" ] );
do Print_Passwd_Info_For( .login );

global_defs:
define PROC( STR .login ) txn task: Print_Passwd_Info_For
{
import from_shmem ¨pinfo¨ :
    TUPLE[ INT .uid, INT .gid, STR .comment ] ARRAY[ STR .login
        : with_default @ => [ -1, -1, "Nonesuch" ] ] ..passwd_info
    do Write_Words( .login, ..passwd_info[ .login ] );
}

The presence of the @-default guarantees that even when there is no entry for .login, there will be useful output.

In order for queries to use multiple shmem segments, some query must have created all associated segments at one time, whether or not it also populated all of those segments with dynara.
21.4 Installing The Shared Memory Infrastructure

In order to use the shared memory feature, it is necessary to follow a simple procedure for installing the necessary materials into the user’s project or application.

First though, by way of introduction, know that Daytona maintains a special system table called SHMEM_SEG that contains details of the Daytona shared memory operations for a given Daytona project. Any user of shared memory must be able to write in SHMEM_SEG, perhaps, if necessary, by means of group write permissions or ACLs (setfacl(1)). Also, any user who wishes to create a shared memory region must be able to write 0-length files SHMEM_<region_name> in the same directory as SHMEM_SEG. So, where is SHMEM_SEG? The answer is specified by the Source note in its rcd.SHMEM_SEG:

```
<Sourc, $(DS_SHMEMDIR:-$(ds_whence par.$(DS_PROJ)))>
```

This means that by default, it is in the same directory as the par for the project, both of which must exist. Otherwise, its location can be overridden by exporting a value for DS_SHMEMDIR.

Since a project supports queries integrating multiple applications and specifically with regards to using shared memory, there is no prima facie aar to put rcd.SHMEM_SEG into and especially so when there may only be a par and no aar (because there are no database tables). Or to put it differently, rcd.SHMEM_SEG belongs in the par because it is a common resource for the entire project (and all of its users) -- and so do not put it in any aar, put it in a par (identified of course by $DS_PROJ); so, a project/par must be created if one does not already exist and is available for use. Assuming then the existence of, say, par.myproj, here is how to install rcd.SHMEM_SEG:

```
Archie -x rcd.SHMEM_SEG -ar $DS_DIR/aar.sys
Archie -r rcd.SHMEM_SEG -ar par.myproj
rm rcd.SHMEM_SEG
```

(The Archie -x can be executed anywhere; however, it would be bad practice and possibly not allowed by permissions to do it in $DS_DIR.)

While it is not necessary to know anything about rcd.SHMEM_SEG, curiosity may prevail and the user may examine it -- but the user should not and must not modify either the rcd or the corresponding table, as that is the system’s sole prerogative. Otherwise, SHMEM_SEG is just a table like all other tables. If it does not exist, it will be created by the first call to Create_Shmem_As_Needed, if that executable is called with the +trustme/+T flag. Otherwise, it can always be created empty and reinitialized by:

```
Sizup -clean_slate -rec SHMEM_SEG
```

Daytona’s Cymbal code for shared memory work comprises the package $DS_DIR/shmem_pkg.cy . Just copy this file into $DS_PATH, preferably in the same directory as the par. (Or alternatively, in order to ensure keeping it up-to-date, one could create a symbolic link somewhere in $DS_PATH that points back to $DS_DIR/shmem_pkg.cy .)
21.5 Special Utilities

Recall of course the shell-level UNIX utilities for managing shared memory: *ipcs* and *ipcrm*. However, Daytona provides much nicer shell-level facilities for managing shared memory. These are *Show_Shmem* and *Rm_Shmem*. The first prints out information about existing shared memory segments for all users by default and the second removes specified shared memory segments:

```
% Show_Shmem +?
Usage: Show_Shmem { -nt <DATE_CLOCK> | -ot <DATE_CLOCK|now> | -u <user_name> }*
Show_Shmem { <name> | <hex_key> | <int_shmid> }*

Rm_Shmem { -nt <DATE_CLOCK> | -ot <DATE_CLOCK|now> | -u <user_name> }+
Rm_Shmem { <name> | <hex_key> | <int_shmid> }+
```

(They are implemented by one executable that is callable by either name.) The *<name>* is the Cymbal SHMEM_SEG name (if any) for the segment (as specified in a call to *Create_Shmem_As_Needed*); *-nt* means *newer than* and *-ot* means *older than*. One of several enjoyable advantages of these commands is that they can work with multiple shared memory segments in one invocation. Also, *Rm_Shmem* has the advantage that when it is invoked with the name of a shmem region, then it not only removes the shmem region from the OS and the SHMEM_SEG table but also, if any of the dynara therein are concurrent, then it will also remove an associated semaphore as well as terminate the Legate process. See their DS Man pages for more details.

Remember also *Describe_Shmem* previously described.

If all processing on a shmem region comes to a halt because some process has a lock on the region and is not releasing it, the culprit can be found by calling *Lock_Blockers_For_Shmem*:

```
PROC( THING .shmem ) Lock_Blockers_For_Shmem
```

For gurus, malloc statistics on a shared memory region can be printed out using *base_malloc_stats_for_shmem*:

```
STR(=) FUN( THING .shmem ) task: base_malloc_stats_for_shmem
```

While one might think of a shmem segment as being large, in any event, its size is fixed. If a program repeatedly frees and recreates dynaras, letting them grow from small to large before freeing them again, then due to the nature of malloc, the shmem region can grow to be full. This is due to fragmentation in malloc block allocations that causes inefficient space usage with active blocks sitting in the middle of unallocated areas of memory. Once this happens, the game is over and the program will fail, saying that it is out of memory even though large portions may not be allocated. One way to avoid this is to create the dynara at somewhat more than their predicted maximum size ab initio by using *with_init_max_nbr_elts* (and *with_growth_factor* = 1.0), although since the concurrent version is never resized, *with_growth_factor* should not be specified in that case: see Chapter 11 for details. Nota bene: for the best performance when using the concurrent version, *with_init_max_nbr_elts* should be specified with the best estimate of the eventual long-term number of elements in the dynara. This can make a big difference: in one test case with 100M elements in the dynara, by adding a *with_init_max_nbr_elts* specification, the time to load them with 128-way parallelism went from 10m to...
4m22s. What’s more, the single process creation time went from 1h34m10s to 14m55s! Clearly, using this option makes a big difference.

21.6 Producer/Consumer Using Non- Concurrent Shmem Dynara

A common way for two processes to communicate data from one to the other is via pipes or when doing so over a network, via sockets. There are several drawbacks to this which are avoidable for pipes by using shared memory. First, the information must be serialized (in the Java sense) by encoding it into a stream of bytes that can be transmitted safely according to some protocol, perhaps by using an end-of-message delimiter (that cannot otherwise appear in the stream). This stream of bytes may contain other delimiters too to enable the receiver to parse it into constituent objects, which would thus discourage sending, for example, the in-memory form of INTs and FLTs which could contain any bit pattern; likewise, obviously it cannot contain any in-memory pointers in this scenario. Also, pipe creation involves a `popen` where one process is fork’d from the other. This is fine but what if two independent processes would like to communicate? They are not going to do it with pipes. In theory, they could do it with FIFOs, i.e., named pipes, but in practice, those are often found not to be reliable.

Fortunately with shared memory, these drawbacks on pipes can be eliminated. In the following example (shm.8.Q), a producer child process produces information to be consumed by an otherwise unrelated child process by means of the synchronized sharing of a dynara element in shared memory. Note that these two processes need not be created as clones of a parent -- they could be created separately at the shell level.
import: package `shmempkg`

with_size_in_k 10*1024 do Create_Shmem_As_Needed( `arenaPC` );
do It;
do Destroy_Shmem( `arenaPC` );

  global_def:
define PROC task: It  // does not have to be a txn task since doing own synch
{  
  // no concurrent since don’t want to copy changed range TUPLEs
  export: from_shmem `arenaPC` TUPLE[ INT, DATE ] ARRAY[ INT ] ..myara
  local: INT .counter;  INT .maxi = 500;

  set ..myara[ 36 ] = [ 44, "2010-02-02" ];

do Kill_Popkorn_As_Needed;

  set ? = new_tendril( executing { do Produce; } );
  set ? = new_tendril( executing { do Consume; } );

  Wait_For_Tendrils
  do Write_Line( "Parent: .counter = .counter"ISTR );
  _Show_Exp_To(..myara[ 36 ])

define PROC Produce
{  
do Lock_Shmem_Obj_Vbl( .myara[36] );
do Sleep(1);  // so that Produce gets the lock first
  fet .i Is_In [ 1 -> .maxi ]
  {
    set ..myara[ 36 ] = [ $#1+1, $#2+7 ];
    // do not separate Unlock and Lock with other code.
    do Unlock_Shmem_Obj_Vbl( .myara[36] );
    do Lock_Shmem_Obj_Vbl( .myara[36] );
  }
  do Unlock_Shmem_Obj_Vbl( .myara[36] );
}
define PROC Consume
{  
fet .i Is_In [ 1 -> .maxi ]

  do Lock_Shmem_Obj_Vbl( .myara[36] );
  when( ..myara[ 36 ]#1 = .prev_seq_nbr +1 )
  {
    set .counter += ..myara[ 36 ]#1;
    set .prev_seq_nbr = ..myara[ 36 ]#1;
  }
  do Unlock_Shmem_Obj_Vbl( .myara[36] );
The Produce clone begins by using Lock_Shmem_Obj_Vbl to get a (default) exclusive lock on a specific dynara element whose range TUPLE will carry the information from Produce to Consume. Note that the lock is gotten on the VBL whose value is a Shmem_Obj, in this case, a range TUPLE. Consequently, .myara[36] is the Shmem_Obj_Vbl to lock, not ..myara[36], which is a TUPLE and which Tracy will not accept. Then Produce enters into a loop where it updates the designated range TUPLE while it has the lock and then releases the lock and immediately asks to have it again so that it can continue to make changes to the shared range TUPLE. Meanwhile, Consume is executing a corresponding loop where it gets the lock as soon as Produce releases it so that Consume can use/read the shared range TUPLE, after which it releases the lock to allow Produce to continue with the next transmission. Note that the It PROC task does not have to be atxn since the query is doing all necessary locking/synchronization explicitly.

This program assumes that locks are granted in FIFO queue order so that when, for example, Produce releases the lock, it has already been arranged for Consume to be waiting for the lock and it so it will get it for sure even though Produce immediately asks to get the lock back after releasing it. There is a race though that must be handled which is that due to scheduling on the machine it could theoretically happen that when Produce releases the lock, Consume goes from getting the lock, doing its work, and releasing the lock and getting it again all before the request from Produce to get the lock is seen by the OS. This is handled by means of keeping track of sequence numbers for the messages.

Here are the prototypes for the locking PROCs (as defined in shmem_pkg.cy), which reuse the locking concepts used elsewhere as with new_channel:

```c
PROC( with_mode manifest /*LOCK_MODE*/ INT .mode = _exclusive_,
     with_patience INT .patience = _wait_on_block_,
     OK_OBJ(void) VBL .some_objv ) task: Lock_Shmem_Obj_Vbl
PROC( OK_OBJ(void) VBL .some_objv ) task: Unlock_Shmem_Obj_Vbl
PROC( OK_OBJ(void) VBL .some_objv ) task:
    Print_Info_On_Blocking_Lock_If_Any_For_Shmem_Obj_Vbl
```

This locking is implementing using fcntl(2) which has the advantage that all locks disappear when the processes holding them exit, thus avoiding the cleanup necessary when using semaphores. In addition, these fcntl(2)-based locks can be _share_ as well as _exclusive_.

Note that this paradigm is actually much more general than just passing a sequence of TUPLEs from one process to the next. In fact, while a process has one of these locks, a process can do whatever it wants in shared memory to the benefit of its partners.

As will become apparent, there is not much of a role here for concurrent shmem dynara because whenever a change occurs in a dynara element, a copy is made, which is not going to be visible to
others unless they pay for doing a lookup.

The locking technology here could also be used for having one or more processes producing messages to be read from a queue by one or more reader processes. This could be accomplish by having a dynara of messages mapping message sequence numbers to messages and by having lock-protected quantities for "last-msg-seq-nbr-written" and "last-msg-seq-nbr-read". Details left to the reader.

21.7 Shared Memory And Parallelization

The following query shows how easily Cymbal parallelization clones can work with a shared memory object (para_hashjoin.1xM.3.d.Q). It also shows how to do a parallelized shared memory hash join.
import: package `shmep_pkg`
export: INT .nbr_clones

set [ .destroy_on_exit, .nbr_clones ] = read( from _cmd_line_ bia[ _true_, 5 ] );

with_size_in_k 1024 do Create_Shmem_As_Needed( `partor` );
do Join_Query;
when( .destroy_on_exit ) do Destroy_Shmem( `partor` );

global_defs:
define PROC txn task: Join_Query // no need for txn if no writers
{
export from_shmem `partor`: FLT ARRAY[ INT .pno ] ..pno_ara
import: INT .nbr_clones

set ..pno_ara = { .pno => .wt : there_is_a PARTC where(
  Number = .pno and Color Is_In [ "turquoise", "blue" ] and Weight = .wt) }

parallel_for .nbr_clones
merging_with_sort_spec[-5, 2]
with_format _table_
do Display each [ .pno, .wt .ono, .supp, .qty ]
each_time(
  .hparti_box = { [ .region, .category ] :
    there_is_a_bin_for DYNORD where(
      Region = .region and Category = .category ) }

and .sect_nbr Is_In [ 1 -> .nbr_clones ]

and parallelizing{
  [ .region, .category ] Is_In .hparti_box and
  there_is_a DYNORD from_section[ .sect_nbr, .nbr_clones ]
  where(
    Region = .region and
    Category = .category and
    Number = .ono and
    Supp_Nbr = .sno and
    Part_Nbr = .pno where( ..pno_ara[ .pno ] = .wt ) and // hash join here!
    Date_Placed Is_In [ '1983-09-07' -> ] and
    Quantity = .qty
  )
  // Unique_Btree_Key_Fam for indexed nested loops join here
  and there_is_a SUPPLIER where( Number = .sno and Name = .supp )
});
}

Note that the creation of the ..pno_ara shmem dynara is accomplished easily at a high-level by a
declarative specification. It is used in a *parallelized* hash join to DYNORD. Every clone is accessing this same `.pno_ara` shm mem dynara: if shared memory were not used, then the operating system would copy the dynara into each clone when it forks the clone. This would be bad if that resulted in running out of memory or causing painful swapping. Also, due to the persistence of shared memory, such a shm mem dynara is available for multiple queries to *import* and use. All that these clones need to do to gain access to the shared memory array is simply to use it; no other syntax is needed (or even available).

### 21.8 Shared Memory, Parallelization, Hash Joins, Views, And SQL Group-By

This next example shows the use of views, parallelization, and SQL group-bys with shared memory. First, note that the following view definition is in `orders.env.cy`:

```c

```
define RECORD_CLASS SQL_SHMEM_HASH_JOIN_VU
as a view where(
    for_each [ INT .pno, FLT .wt, INT .ono, STR(30) .supp, INT .qty ]
    conclude(
        there_isa SQL_SHMEM_HASH_JOIN_VU where(
            Part_Nbr = .pno and
            Part_Weight = .wt and
            Order_Nbr = .ono and
            Supplier = .supp and
            Quantity = .qty )
        iff(
            [ .pno, .wt, .ono, .supp, .qty ] Is_The_Next_Where(
                .hparti_box = [ [ .region, .category ] :
                    there_is_a_bin_for DYNORD where(
                        Region = .region and Category = .category ) }
                and .sect_nbr Is_In [ 1 -> .max_clone_nbr ] // best for seq access
                and parallelizing(
                    [ .region, .category ] Is_In .hparti_box and
                    there_is_a DYNORD from_section[ .sect_nbr, .max_clone_nbr ]
                where(
                    Region = .region and
                    Category = .category and
                    Number = .ono and
                    Supp_Nbr = .sno and
                    Part_Nbr = .pno where( ..pno_ara[ .pno ] = .wt ) and
                    Date_Placed Is_In [ `1983-09-07` -> ] and
                    Quantity = .qty )
                and there_is_a SUPPLIER where( Number = .sno and Name = .supp )
            )
        )
    )
) using( outside [ INT .max_clone_nbr,
                 from_shmem `partor` FLT ARRAY[ INT .pno ] ..pno_ara ]
    );

Note how the view identifies the shmem dynara as being outside, implying of course that some other process created it beforehand. Then it is very easy to use this view (para_hashjoin.1xM.3.h.IQ):
import: package "shmem_pkg"

select Part_Nbr, avg(quantity)
from SQL_SHMEM_HASHJOIN_VU
where Order_Nbr > 0
group by Part_Nbr
order by 2
parallel for 5;

Likewise, for_each_times working with shmem dynaras can be parallelized just by exporting/importing
the dynara into a suitable txn task and parallelizing the for_each_time (see the shm.conc.paraload.2.IQ
discussion in the concurrent section below).

21.9 Concurrent Access To Shared Memory Dynara

Suppose the paramount goal for shared memory processing is speed -- and one is willing to
sacrifice for it. To achieve that end, it would be good to avoid getting any locks at all because they
impede concurrency (and thus speed) by causing processes to wait to act. Fortunately, the data
structures used by Daytona can be configured so that readers of dynaras do not have to get (share)
locks in order to read dynaras safely, even while one or more writers are busy changing them.
Furthermore, while multiple writers on dynaras have to get locks on the dynara data structure, they are
actually just latches, i.e., (exclusive) locks that are so fine-grained and short-lived that the chances of any
two processes colliding on a latch are miniscule -- and certainly far, far less than if each writer had to
exclusively lock entire dynaras in order for it to proceed. (Writers also have to get locks on malloc data
structures which are concurrent to a certain imperfect extent.) Dynaras configured to be used in this
fashion are called concurrent (and they necessarily reside in shared memory).

Also, in the interests of speed, there is no mechanism when using concurrent shmem dynaras that
supports rollback by maintaining a Do_Queue of intended actions-to-do, thus eliminating that
overhead. (This also rules out timestamp-based concurrency control.) Consequently, it becomes clear
that there is no possibility of supporting any of the ACID properties of transactions. Specifically, the
lack of a Do_Queue rules out Atomicity and Durability: to see this, just think of what can happen when
a txn gets a TERM signal or if the machine crashes. Of course, in the latter case, shared memory
disappears as well. Consistency is not supported because, as a rule, any dynara changes can be seen
immediately by other processes and so states inconsistent with application integrity rules can occur.
Unless the Cymbal do_critical statement group is used, Isolation is not supported either because without
(sufficient) locking or timestamping to help out, there is no serializability as can be further proved by
examples demonstrating the existence of the canonical pathologies of lost updates, unrepeatable reads
during a txn, incorrect aggregation, and phantom records. (There cannot be any "reads of uncommitted
data" simply because rollback is not possible.) Fortunately though, using do_critical groups rules out
lost updates but there is nothing that can be done about the read-related pathologies in this setting.

However, the algorithms that Daytona uses for concurrent dynara do at least guarantee basic
safety in the sense of not allowing the actions of multiple processes to corrupt either the shared data
structures or the processes themselves -- which is no small benefit!

So, in short, in the interests of handling concurrent dynara as fast as possible, the system does not
use locks of any kind to read the dynara, except when it scans a dynara in a do_critical statement group; any exclusive lock (on dynara data structures) is a latch -- when not gotten as part of a do_critical; when in a do_critical, the exclusive locks used in updating dynaras are on dynara elements (not the dynaras themselves as in the non-concurrent shmem case); and all changes to dynaras are immediately visible to all processes exporting/importing those dynara -- with the exception that do_final and do_critical groups of statements offer a means to prevent others from reading in-progress updates.

All this is accomplished by using exactly the same syntax that is used to work with dynara that are not in shared memory but is expressed by using just three new keywords: concurrent, do_final and do_critical. (The term concurrent-do is used to refer to either or both of do_final and do_critical when talking about properties that apply to both.) So, instead of thinking in terms of locks, it is the concurrent shared memory context that one has to keep in mind when writing code for these dynaras. Once again, when not in a concurrent-do, that shmem context states that the effect of every statement that changes a dynara is immediately visible to any other process with access to that dynara.

All concurrent shared memory work involving changes to dynara must take place within the confines of a transaction task; on the other hand, read-only concurrent tasks are fine and furthermore, it is permitted for a a txn task to call a non-txn task that imports the same concurrent shmem dynara as its caller and does ONLY read operations on those arrays -- no changes to them. Fortunately, that same transaction task can be used for RECORD_CLASS txn activity as well as for work on non-concurrent shmem dynara as well, possibly in the same shmem region as the concurrent ones.

What follows now is a discussion of the implementation and consequences thereof of concurrent shmem dynara functionality as manifested by deletes, inserts, reads/scans, do_final groups, do_critical groups, and one-line updates. The following examples and more are in shm.conc.demo.IQ. They all take place in the body of the txn task Update_Shmem_Region:

```plaintext
import: package `shmempkg`
    with_size_in_k 20480 with_legate
do Create_Shmem_As_Needed(`arena3`);
do Update_Shmem_Region;
do Destroy_Shmem(`arena3`); // just to clean up this test

global_defs:
define PROC txn task: Update_Shmem_Region
{
export concurrent from_shmem `arena3`:
    TUPLE[ INT, DATE, FLT ] ARRAY[ INT, INT 
        : with_default @ => [ -1, `2000-01-01`, 0.0 ] ]..myara
local: TUPLE[ INT, STR ] ..t2vv
...
}
```

Note that the concurrent keyword attribute for myara has been factored out to the export clause so that it could apply to multiple such dynara. On the other hand, the use of this factoring tactic is not mandatory and so, the concurrent keyword could be deployed in each dynara import/export.
Credit: Daytona’s concurrent dynara feature is implemented on top of Phong Vo’s highly sophisticated, one-of-a-kind cdt (hash table), vmalloc, and aso (atomic scalar operation) libraries. Note that while cdt offers two implementations of data structures with the same API as a hash table, only one of those implementations looks mostly like a conventional hash table. The second one, which is the one used for concurrent shmem dynara, is based on hash-tries, a recursive hashing data structure.

21.9.1 Concurrent Deletes Of Shmem Dynara Elements

Deletes are expressed in the usual way:

```plaintext
set ..myara[47] = \n;
set ..myara = \n;
```

However, in terms of the implementation, Daytona distinguishes between deleting a concurrent dynara element at the hash table level, which means getting it out of the hash table data structure so that it is inaccessible to further lookups by processes, and then using the malloc library to free the associated storage. What happens for concurrent dynara is that a Cymbal-level element deletion corresponds to first deleting it at the hash table level in a guaranteed safe manner and then putting that element in a queue to be freed by malloc later by a completely separate child process called the Legate. This separation of functionality enables Daytona to implement a strategy whereby shmem storage is only freed when it can be proved that no process is using it or for that matter, can use it. Indeed, the problem being solved here is that, just because an element is no longer officially part of the dynara does not mean that it had not been accessed before that moment by some other process which is now busy reading it or alternatively, perhaps working on a copy of it in a single-line update or in a concurrent-do group. In both cases, Daytona is ensuring that the second process is not going to crash and burn due to trying to work with freed storage, i.e., garbage. In particular, as will be explained in more detail later, in the second case, the second process will swap in its copy of the range portion into the now deleted (but still required to exist and so not freed) dynara element, not that anyone cares anymore because that element is no longer in the dynara.

So, how does Daytona determine when deleted concurrent dynara elements can be freed by the Legate using the malloc library? The criterion is that an object (either the domain or range portions of a dynara element) can be freed when the time it was put on the to-be-freed queue upon (Cymbal-level) deletion from the dynara is earlier than the minimum of the last times that all the processes using the shmem region for concurrent dynara have declared that they are not using shmem, which action happens at the start of their shmem transactions: clearly, the process is not using shmem just before it starts to use it! This is a minimax criterion. The implication is that the user has to write their concurrent shmem transactions in such a way that they return often enough so that unfreed storage does not pile up to the point of filling up the shmem region. (Incidentally, this is highly similar to the way that Daytona manages string garbage collection (and removal) in non-shmem Cymbal.)

Recall that the Legate is a clone of just one of the processes using the shmem region whose sole purpose is to free already deleted/detached malloc’d storage as soon as possible. It is called the Legate because it has been delegated the task of freeing storage. In principle, it could be delegated other common tasks as well. The Legate’s two main data structures, the "to-be-freed" queue (sorted by timestamp) and a "process-allclear-from-shmem" queue (also sorted by timestamp) are kept in the
associated shmem region for the use of all processes using concurrent dynaras stored in the region. Note that when a txn using shmem returns, it deletes its entry in the "process-allclear-from-shmem" queue: this way, processes that decide not to use shmem for a while do not prevent the Legate from servicing processes that are actively using it. The Legate’s only opportunities to free objects come when processes register their allclear timestamps at which points they also bump a semaphore causing the Legate to wake up. Daytona will refuse to run any process using concurrent dynaras unless there is a Legate running for the associated regions. The way to get a Legate going is to use the \texttt{with_legate} keyword to \texttt{Create Shmem As Needed}. Don’t worry if more than one process uses this keyword-argument on a given shmem region because Daytona will only allow one Legate to run -- but there has to be one. Also, whichever process succeeds in launching the Legate has to have transactions that use all of the concurrent dynara that are used by any processes using concurrent dynara in the associated shmem region.

As for the Cymbal deletion of all of the elements of a dynara, it would look like this one:

\begin{verbatim}
set ..myara = {};
\end{verbatim}

In the concurrent setting, this amounts to safely deleting each element individually one-by-one in the way already described, although no exclusive latches are gotten (see the read discussion below). After all, if someone is intending to read all of a dynara at the same time someone else is intending to delete all elements, then the application is at serious cross-purposes with itself and needs to synchronize using \texttt{TICKET\_BUNCHes} or else use some sort of mediation outside of Daytona. As is the case with all dynara, this operation ends with the dynara still in existence but empty.

21.9.2 Concurrent Inserts Of Shmem Dynara Elements

As with all dynara operations, whether in shmem or not, the syntax is the same as illustrated next for concurrent inserts.

\begin{verbatim}
set ..myara[ 477, 111 ] = [ .x+1, `2012-06-06`, 37.3 ];
\end{verbatim}

The implementation is subtly and slightly different however. While in the non-concurrent (shmem or not) case, the domain and range components of a dynara element are allocated in the same C struct, in the concurrent shmem case, they are allocated separately (in shmem, of course) with the domain portion containing a pointer to the range portion (called the range tuple). Inserts are straightforward enough: while, obviously, the dynara element has to be malloc’d before insertion, it must also be populated with the desired values before insertion: this prevents other processes from seeing an incomplete element in the process of being created. Then the hash table library makes the completed element visible to all atomically.

There is one special case though. In the example, suppose that \texttt{..myara[477, 111]} already exists! Then what looks like a vanilla insert actually isn’t. Instead, it must be treated as an update. Since the domain part is already in the dynara, all that is needed is to get it to have the new range tuple instead, which is done by Daytona atomically swapping it into place. This atomicity is achievable because it just suffices to change the domain portion’s pointer to point to the new range tuple (and to put its predecessor on the Legate’s to-be-freed queue). Fortunately, each supported platform has a way to atomically write (multi-byte) pointer values to locations in memory (as well as to atomically read such). Just to be clear, atomicity is required because otherwise the action to laying down 8 bytes of
pointer could in principle result in another process reading some of the new value and some of the old value, hence reading garbage.

Nonetheless, this swapping is not without further challenges either because when a process tries to swap, another process could have gotten there first with its range tuple swap for the same domain component, thus preventing the first process from getting its tuple into place, or alternatively, because of two other specific opportunities for another process to change the contended-for address before the given process can provably accomplish its swap. The following C code shows how such competition can be detected and compensated for. Of course, just because a process succeeds in getting its range tuple into place doesn’t mean that it is going to stay there for long.

```
if( ret_objp NEQ tmp_objp ) // i.e., dynara already has an element for this domain value
{
    myara_shmem_vbl_Range_Val *tstval, *curval, *prev_curval = NULL;
    // pointer swap in the new range tuple
    while( (tstval GETS (myara_shmem_vbl_Range_Val *) asogetptr( &(ret_objp->rng_val) ), TRUE)
        && tstval NEQ (curval GETS (myara_shmem_vbl_Range_Val *) asocasptr(
            &(ret_objp->rng_val), tstval, tmp_objp->rng_val ))
    )
    {
        if( prev_curval EXISTS && prev_curval NEQ curval && prev_curval NEQ tstval )
            myara_shmem_vbl->Delegate_Free_Range_Val( prev_curval );
        // do this to the other process' value -- which our goal is to replace
        myara_shmem_vbl->Delegate_Free_Range_Val( tstval );
        prev_curval = curval;
        asorelax(1);
    }
    if( prev_curval EXISTS && prev_curval NEQ curval && prev_curval NEQ tstval )
        myara_shmem_vbl->Delegate_Free_Range_Val( prev_curval );
    myara_shmem_vbl->Delegate_Free_Range_Val( tstval );
    ...
}
```

asogetptr atomically (hence safely) reads a pointer value. The asocasptr() function returns whatever pointer it finds at the location \&(ret_objp->rng_val) BUT it only does the swap (atomically) if that value is equal to tstval. asorelax is used to cause the calling process to sleep for 1 nanosecond, thus allowing other processes an opportunity to run themselves. These aso-prefix macros are part of Phong Vo’s atomic-scalar-operations library. (EQ, NEQ, NULL, TRUE, EXISTS are all obvious C macros.) The two other specific opportunities for another process to interfere with the logic of the given process are just after the setting of tstval and then after the setting of curval. Upon detecting that the dynara already contains an element for the specified domain value, the while loop atomically gets a pointer to the current range tuple value. Then the while loop uses the asocasptr compare-and-swap macro to loop as often as necessary to provably get this process’s new range tuple value into place. Specifically, since asocasptr will succeed in doing the swap only if the current/found value it finds is equal to tstval, if it is not, then the process must try again.
21.9.3 Concurrent Reads Of Shmem Dynara Elements

Due to a clever hash table implementation, reads are possibly the simplest since they just don’t get any locks or even latches and so are safe from collisions with writers and don’t copy shmem to shmem like updates do but still they have subtleties nonetheless in the concurrent setting.

```plaintext
set ..myara[ 477, 111 ] = [ 74, "2012-06-06", 37.3 ];
set .tu = ..myara[ 477, 111 ];
when( .tu = ..myara[ 477, 111 ] ) do Write_Line("Good -- still there!");
```

Just to get things going, the first statement is an assignment that creates (or updates!) an element of shared memory -- so no reading into program variables there. The next assignment does a read from shared memory into the process-local VBL `tu`. While the third statement also does a shmem read (of the same element), it also reflects the fact that just because a process writes an element to a concurrent shared memory dynara in a txn does not mean that it will get that same range value out if it should try to read that element immediately thereafter. That’s because some other process could have changed `..myara[477, 111]` between the execution of the two statements. We have seen this happen!

Worse, the element `..myara[477, 111]` could have been deleted from the dynara by another process, say, B, before process A could get hold of it to read. However, since this dynara has an @-default, process A would have reacted to the deletion by installing the @-default tuple, which would have been read instead; otherwise, of course, in the absence of an @-default, it would have been a fatal error to try to read an element that is not there. Still, there is a slim chance for some other process C to interleave with process A and cause `.tu` to be compared by process A with a tuple not equal to the @-default value. The reason is because when process A cannot find `..myara[477, 111]`, it continues forthwith to call the dynara’s Insert_Obj routine which simply does not and cannot return without succeeding and producing a pointer to the dynara element it was charged with inserting (even if some other process is simultaneously competing to delete it). However, still, by the time that the equality comparison is made, the range tuple (alone) for that very dynara element could have been changed by process C. However again, since the range tuples are in effect treated atomically and as a unit, all the components used in a range tuple comparison will be from a single consistent range tuple, not some Frankensteinian mixture. So that’s some good news and the other good news is that there cannot be any fatal error due to data structure corruption of any kind. So, obviously, in the concurrent setting, one cannot even talk about reads with talking about writes, as will continue to be seen.

The same story is repeated in this next example where once again the statements one process is executing can be transparently interspersed with the actions of other processes. It’s a real free-for-all. Assume for just this next discussion that there is no @-default for `myara`.

```plaintext
when( ..myara[ 477, 111 ] = ? )
{
    do Write_Words( ..myara[ 477, 111 ] );
}
```

As before, it is entirely possible for another process to delete `..myara[477, 111]` between a successful execution of the existence test and the subsequent attempt to Write out the range value, which will then fail with a fatal dynara-element-not-found error.
To avoid this unfortunate eventuality and furthermore, to achieve other efficiencies, one should use instead the following sophisticated simultaneous setting and existence testing of a VBL VBL defined previously (which strategy is also mentioned in Chapter 11):

```plaintext
when( set(t2vv, .myara[477, 111]) = ? ) // simul. setting and testing for existence
{
  do Write_Words( 477, 111, "->", ..t2vv );
}
```

What happens here is that the `t2vv` TUPLE VBL VBL gets assigned as its value the range tuple value VBL `.myara[477, 111]` (note the single dot!) if it exists whereas if it does not exist because the dynara is not defined at `[477, 111]`, then the accompanying existence test simply fails as it should (in this declarative setting). Assuming though that the existence test succeeds, then that range tuple value which is equal to `..t2vv` is available for Writing -- without any additional dynara lookups. What's more, not only is one additional dynara lookup avoided but actually so in total are one for each component of the range TUPLE when Writing, which would number three for the alternative in this case: `..myara[477, 111]#1`, `..myara[477, 111]#2` and `..myara[477, 111]#3`. However, in the case of doing updates with `concurrent` shmem dynara, it is necessary to enclose the when-set-do in a `do_final` -- see below.

Now, remembering that `.t2vv` is the shmem location of a range tuple, what are the consequences of another process deleting the dynara element at `[477, 111]` (or its range value due to an update) between a successful existence test (plus the assignment) and the Writing? The answer is nothing, it's all the same whether the deletion (or updating) occurs or not. Recalling that the operation of removing a dynara element is distinct from freeing the associated storage, the reason there are no consequences is because Daytona's garbage disposal mechanism via the Legate is architected in such a way that it will not actually free the range tuple in question until this process is done working with it, while in the meantime any process trying to look up the dynara element at `[477, 111]` will no longer be able to find it.

Another inescapable fact of life in this setting is that if one process is looping over all elements of a dynara, then if some other processes are adding and/or deleting elements of this dynara, then the first process may not visit every element that exists at any given point in time. This is because such loops advance by saving a pointer to the last element accessed/read and then subsequently asking the hash table library to produce the next one as a function of being given a pointer to the last one that the hash table library produced. Now if that last-produced one has been deleted (but guaranteed not freed!) by the time that the loop asks for the "next" element, then it is possible for the "next" element to be what would have been the next one were there no deletion which is ideal but unfortunately, there is also a smaller probability that an indeterminate number of elements will be skipped over. On the other hand, fortunately inserts and obviously updates cannot cause this phenomenon of skipping over a bunch of elements that are there and furthermore, the user need not worry about fatal errors. Here are two ways such a loop can look like:

```plaintext
  do Write_Line( sum( over .b#1 each_time( ..myara[ 2? ] = .b ) ));
  for_each_time .b ist( ..myara[ 2? ] = .b ) { do Write_Words(.b#1); }
```

However, when such loops are enclosed in a `do_critical`, then the `do_critical` will get an `fcntl` share lock.
on the relevant dynara for its duration and that will lock out all single-element deletes for that
duration, since they will be getting fcntl exclusive *latches* on the dynara.

### 21.9.4 Concurrent do_final Updates Of Shmem Dynara Elements

For concurrent dynara, Cymbal offers the *do_final* variant of its do-group as a way to obtain
consistent and efficient updates of dynara range TUPLEs. Consider this fragment of code illustrating all
the different ways of doing updates (on a single dynara element):

```c
export from_shmem "arena":
    concurrent TUPLE[ INT, FLT ] ARRAY[ INT, STR ] ..myara2

set ..myara2[ 47, "ABC" ] = [ 74, 34.34 ];
set .xx = 4321.0;
set ..myara2[ 47, "ABC" ] = [ ?, 1234.5+.xx ];
set ..myara2[ 47, "ABC" ]#1 = -1;
when( ..myara2[ 47, "ABC" ]#1 < 0 )
    set ..myara2[ 47, "ABC" ]#2 *= 202;
set ..myara2[ 47, "ABC" ]#1 = 300*$;
set ..myara2[ 47, "ABC" ] = [ ?, sqrt($#2) ];
```

In real application code, these assignments could be interspersed with other logic. Unfortunately, if
this code were executed, after each assignment, readers would be able to read the resulting dynara
element. The problem is that there could well be integrity constraints that specify relationships that
must hold among the various components of the element’s range TUPLE and so this serial updating is
allowing other processes to see inconsistent states. (Another way to think of these integrity constraints
is as invariant conditions that must hold for the data.) Furthermore, this code is very inefficient as it
directs Daytona to look up the same dynara element over and over again and as it turns out in the
concurrent setting, to make a whole bunch of ultimately unnecessary copies (and garbage for the
Legate).

What is needed is some new syntax that invokes an implementation that allows the updating
process to make exactly one temporarily local shmem copy of the range TUPLE and do as much
cogitating and manipulating as desired on that copy free from the attentions of other processes until
finally and atomically swapping it back into place at the last possible moment (with the Legate
eventually freeing the original version). At the Cymbal level, all that is needed after rewriting to use a
TUPLE VBL VBL is an embracing *do_final*:

```c
```
export from_shmem "arena":
    concurrent TUPLE[ INT, FLT ] ARRAY[ INT, STR ] ..myara2

    set ..myara2[ 47, "ABC" ] = [ 0, 0.0 ]; // just to get the elt to exist

do_final {

    // AFTER THIS ASSIGNMENT, ..tvv is a private shmem copy of the range TUPLE
    set .tvv = ..myara2[ 47, "ABC" ];
    set ..tvv = [ 74, 34.34 ]; // not an update!
    set .xx = 4321.0;
    set ..tvv = [ ?, 1234.5+..xx ];
    set ..myara2[ 747, "ABC" ] = ..tvv;
    set ..tvv#1 = -1;
    when( ..tvv#1 < 0 ) set ..tvv#2 = 202;
    set ..tvv#1 = 300*$;
    set ..tvv = [ ?, sqrt($#2) ];

    // just before this brace, the private copy is atomically swapped into place
    // and its predecessor is sent to the Legate

}

The do_final name indicates that the atomic swapping in of the new range TUPLE(s) (because more than
one TUPLE VBL VBL is allowed) occurs just before the final enclosing brace. At the C-level, this
swapping occurs in essentially the same way that it occurs for concurrent inserts that must turn into
updates.

Here are some rules with regards to concurrent-do groups (of both kinds). First, assignments that
define an alias (i.e., a VBL VBL) to a range TUPLE for a dynara element can only appear in a do_final
(or a do_critical). Here is what one looks like:

    do_final {
        ...
        set .vv = ..myara[ 47 ];
        ...
    }

Note that the assignment appears as a statement directly under the do_final (i.e., not nested within or
reachable from another statement in the do_final as would occur if it was even in the do-portion of a
when): that is the only way for such an assignment to be considered as being part of a do_final. By
analyzing the code of a concurrent-do, Daytona quietly classifies such range TUPLE aliases as being
read-only or not: read-only aliases do not do copy-modify-swaps. Obviously, a range TUPLE VBL VBL
is NOT read-only in a concurrent-do if it is used there to change the value of that range TUPLE.
Furthermore, passing a range TUPLE VBL VBL to a user-defined fpp as an alias causes Daytona to
conservatively conclude that the VBL VBL is not read-only (because that fpp could change the
associated range TUPLE). Daytona will not allow the same range TUPLE VBL VBL to be defined twice
by assignment in the same concurrent-do because that will surely cause one of the updates to be lost.
In the event that user defines two or more NON-read-only range TUPLE aliases for the same dynara
in the same concurrent-do or within a concurrent-do and another concurrent-do nested within it, then
Daytona will warn the user saying that if both of those aliases happen to refer to the same dynara element at runtime, then one of the updates is likely to be lost: caveat lector. Keep in mind that (user-specified) concurrent-do groups are solely for managing user-specified range TUPLE aliases: they do not interact in any way with concurrent dynara adds/deletes/updates that are specified without using range TUPLE VBL VBLs.

Note also that the do portion of a when statement can be a do_final or do_critical; likewise for a for_each_time, as both are illustrated by (shm.conc.6.IQ):

```plaintext
when( ..myara[ 48 ]#1 > 0 )
do_final {
    set .tuvv = .myara[ 48 ];
    set ..tuvv#1 = -1;
}
fet .tvv ist( .tvv = .myara[ ? ] and ..tvv#1 > 0 )
do_final {
    set ..tvv#1 += 100;
}
{ // do_final not even implied here since use is read-only  
do Write_Words( .d, "->", ..tvv#1 );
}
```

Daytona will infer a do_final for the do of a for_each_time that needs one (which assumes that the relevant range TUPLE VBL VBLs are in the fet VBL list and defined in the fet assertion) but it doesn’t hurt to be explicit about the matter.

As mentioned above, when the goal is to simultaneously text for the existence of a concurrent dynara element and if it exists, obtain a pointer to its range element and then to update that, then it is necessary to enclose the whole affair in a do_final:

```plaintext
do_final {
    when( set( tvv, .myara[ 9860 ] ) = ?
    ){  
        set ..tvv#1 ++;
        set ..tvv#3 = "ABC";
        _Show_Exp_To(..tvv)
    }
}
```

As for “updating” dynara domain TUPLE components, that amounts to creating a new element and deleting the old:

```plaintext
set ..myara2[ 747, "ABC" ] = ..myara2[ 36, "ABC" ];
set ..myara2[ 36, "ABC" ] = \n;
```

What would be the effect of process A changing ..myara2[ 747, "ABC" ] and deleting ..myara2[ 36, "ABC" ] while process B is in the midst of updating both in a do_final? The answer is that ..myara2[...]
747, "ABC"] will become whatever the last update is and ..myara2[ 36, “ABC”] will no longer exist even though process B thinks it accomplished an update.

The fact is that even when restricting attention to just do_final groups, there is still no serializability for actions on concurrent dynara. This can be seen due to the possibility of "lost updates" which can occur as follows.

1. Suppose at time 1, process A’s do_final reads (and copies) a value of 47 for ..ara[101].
2. Then, at time 2, process B’s do_final reads (and copies) a value of 47 for ..ara[101].
3. Then at time 3, process A’s do_final adds 1 to its 47 and copies out that 48.
4. Then at time 5, process B’s do_final adds 1 to its 47 and copies out that 48.

The result at time 6 is that ..ara[101] is 48 whereas if the do_final groups were executed one after the other then the result would be 49. That’s a lost update and a lack of serializability.

And the do_final situation is fraught with even more peril. Consider this two-concurrent-dynara scenario:

<table>
<thead>
<tr>
<th></th>
<th>Txn A</th>
<th>Txn B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A.F1  ← R1.F1</td>
<td>B.F1  ← R1.F1</td>
</tr>
<tr>
<td>2</td>
<td>A.F1  ← A.F1 - 100</td>
<td>B.F1  ← B.F1 - 200</td>
</tr>
<tr>
<td>3</td>
<td>A.F2  ← R2.F2</td>
<td>B.F2  ← R2.F2</td>
</tr>
<tr>
<td>4</td>
<td>A.F2  ← A.F2 + 100</td>
<td>B.F2  ← B.F2 + 200</td>
</tr>
<tr>
<td>5</td>
<td>R1.F1  ← A.F1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>R1.F1  ← B.F1</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>R2.F2  ← B.F2</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>R2.F2  ← A.F2</td>
<td></td>
</tr>
</tbody>
</table>

R1 and R2 are two range tuples, the likes of R1.F1 are field values in the range tuples, the A./B. values are for storage local to A/B, resp. and the arrows indicate assignment. The actions for each txn take place within a single do_final. The relevant integrity constraint is that R1.F1 + R2.F2 is constant.

There is plenty of trouble here. For the given order of execution, the state of the dynaras is inconsistent for a reader after each of the times 5, 6, 7, 8, when the copies of the range tuples are being swapped into place. This is a race actually where all the orderings are more or less inconsistent and for the given ordering, the result is permanently inconsistent after time 8. Also, there are two lost updates for A.F1 and B.F2 implying that this execution is not serializable.

Additionally, it is clear that even if there is just one updating transaction running, then there are very small windows between when the several swaps at the end of a do_final are occurring when readers can view an inconsistent database state. While this is inescapable because readers do not get locks, it does lend itself to the claim that these updating do_finals offer "eventual consistency" in a very short amount of time when they are not competing with each other to update the same tuples.

21.9.5 Concurrent do_critical Updates Of Shmem Dynara Elements

It can well be unacceptable if in fact an application’s updaters will compete to update the same
dynara elements and thus potentially create long-lasting inconsistencies for the given integrity constraints. In other words, since the use of \texttt{do\_final} has been seen to fail to guarantee readers any more than single dynara element consistency, the remaining inconsistency possibilities may be unacceptable -- but only when there are integrity constraints to be violated that are accompanied by the required competition, thus implying that more than one process is writing. Anyway, can something be done to improve this inconsistency situation?

The answer is yes and the solution is to use the \texttt{do\_critical} construct at the price of losing some concurrency due to getting exclusive locks instead of latches. A \texttt{do\_critical} reuses the implementation of a \texttt{do\_final} but extends it by simply requiring the implementation to get exclusive UNIX fcntl(2) locks on the dynara elements to be updated (not on the dynara themselves) when they are first read and of course releasing them immediately after all of the range tuple swaps have been made.

According to the two-phase locking theorem, this makes these \texttt{do\_critical} transaction analogs serializable, thus eliminating these lost updates that lead to long-lasting and/or permanent update inconsistencies. (Inconsistencies are eliminated because the actions of competing txns are equivalent to those of some serial execution and because each txn preserves consistency and so does so in series.) Furthermore, since readers get no locks, they remain able to get the lightning-quick eventual consistency previously mentioned and continue to be prey to the other pathologies that accompany reading such as incorrect aggregation. Note that deadlock when there are multiple range TUPLE VBLs is not a problem since fcntl will catch it, causing the would-be-offending transaction to abort, which in itself is no problem either because it has not yet had the chance to make any updates and thus, no rollback is required. Important: note that deadlock situations can be handled programmatically by enclosing the \texttt{do\_critical} in a try-else block written so as to cause the system to repeatedly retry that \texttt{do\_critical} again (up to some specified point) when deadlock happens (shm.stress.wr.crit.5.IQ). However, if deadlock occurs after the first range TUPLE VBL gets its temporarily local value, then there will be (semi)-permanent shm garbage that the system will not free -- but this would typically be a rare event and a small loss. Fortunately also, if a process dies for whatever reason, UNIX releases any fcntl locks it is holding. Also, as for the effects of signal handling on a sequence of final swaps in a concurrent-do, Daytona holds all interactive STOP signals during any txn and the user has the option of holding others.

Note that any inconsistencies having to do with inserts and deletes are not addressed by this mechanism because any inserts and deletes (and one-line updates) that occur in a \texttt{do\_critical} happen immediately, they are not delayed to the end of a some kind of section like the TUPLE VBL VBL updates are. Also, while one might think that the locking nature of \texttt{do\_critical} could safely allow Daytona to avoid the copy-swap, that’s not true because, since the readers don’t get locks, they would be in jeopardy of seeing partially constructed range tuples in the midst of being changed. Lastly, observe the finer grain nature of \texttt{do\_critical} as there can be several in a given Cymbal transaction and they obviously have less overhead than a txn itself.

As a technical note, fcntl can be used in this way because, since it does not require the addresses it locks to actually exist in a file, arbitrary addresses can be locked. Also, note that it is the dynara element itself (through the address of its domain portion) that is being locked, not its volatile range tuple.

As for examples, just change \texttt{do\_final} in any previous example to \texttt{do\_critical}. 

As for examples, just change \texttt{do\_final} in any previous example to \texttt{do\_critical}. 

Copyright © 2013 AT&T
All Rights Reserved.
September 15, 2013
21.9.6 Concurrent One-line Updates Of Shmem Dynara Elements

Recall these one-line updates of shmem dynara elements:

\[
\begin{align*}
\text{set } & .\text{myara2} [ 47, \text{"ABC" } ] = [ ?, 1234.5+.xx ]; \\
\text{set } & .\text{myara2} [ 47, \text{"ABC" } ]\#1 = -1; \\
\text{when( } & .\text{myara2} [ 47, \text{"ABC" } ]\#1 < 0 ) \\
& \text{set } .\text{myara2} [ 47, \text{"ABC" } ]\#2 \ast 202; \\
\text{set } & .\text{myara2} [ 47, \text{"ABC" } ]\#1 = 300*\$; \\
\text{set } & .\text{myara2} [ 47, \text{"ABC" } ] = [ ?, \text{sqrt}(\$\#2) ];
\end{align*}
\]

When the dynara involved is concurrent, then Daytona implements each one of these behind the scenes by using the same copy-modify-swap paradigm used for the do_final. (So, if do_critical functionality is needed, then that will have to be used instead of a one-line update.) However, since copying (and the accompanying Legate work) is resource-intensive, it makes sense to copy as little as possible by confining as much updating work as possible to a few concurrent-do groups (which will then involve only one copy per dynara element changed) instead of one copy per insert/delete/update statement outside of a concurrent-do.

21.10 How General/Useful Are Shmem Dynara?

While it is true that the only objects that Daytona currently allows to be in shared memory are dynara, these turn out to be fairly general after all. First, since at the least, Cymbal dynara map TUPLEs of scalars to TUPLEs of scalars, any of the scalar types can be stored in shmem. (And exotically enough, this includes TUPLE VBLs, since a VBL is a scalar (shm.toftv.Q, shm.conc.toftv.IQ).) The trick then to storing an individual element of an arbitrary scalar type is just to take a suggestive STR name as the domain value of some suitable dynara element and map that name to whatever scalar value is desired, even if there is just one element in the dynara.

Also, it has to be said that people are getting a lot of value out of key-value stores in cloud computing these days and these are essentially dynara.

Furthermore, since a dynara element has the flavor of a record (divided into domain and range portions), a dynara seems to be the analog of a table (with a single built-in unique index). Is it possible then to extend the analogy by supporting multiple indices for a dynara? The answer is yes. As will be seen, the indices will be dynara or BOXes themselves and it will be necessary to use VBL VBLs to arrange for multiple "indices" to index the same range TUPLE. This capability will first be described for conventional dynara and then, suitably adapted, for non-concurrent and concurrent shmem dynara.

21.10.0.1 Conventional Dynara As Tables With Multiple Indices

In order to treat dynara as tables with multiple hash and skip-list indices, the first realization is that the range TUPLE must contain all of the data so that no matter which index is used to locate the range TUPLE, all of the "record’s" data will be available. Furthermore, since there is no metadata (i.e., an rcd) to serve as the basis for code generation, it is necessary for the user to explicitly write all of the operations -- and to remember to do all of them. Here is an excerpt from dynara.as.tbl.2.Q that shows...
how to load up such a dynara-as-table using a join query:

```plaintext
define CLASS TU_TYPE = TUPLE[ (3) INT, DATE, FLT ]
export:
  TU_TYPE
    ARRAY[ INT : with_default @ => [ -1, -1, -1, "2000-01-01", 0.0 ] ] .myara
  TU_TYPE VBL ARRAY[ INT, INT ] .myara_idx2
  LIST[ TUPLE[ DATE .dr, TU_TYPE VBL .tuv ]
    : with_sort_spec[1] with_deletions_ok ] .myara_dr_idx
  LIST[ TUPLE[ INT .sno, INT .pno, TU_TYPE VBL .tuv ]
    : with_sort_spec[1, 2] with_deletions_ok ] .myara_sno_pno_idx
fet [ .ono, .sno, .pno, .dr, .wt_in_k ]
ist(
  there_is_an ORDER where(
    Number = .ono and
    Supp_Nbr = .sno and
    Part_Nbr = .pno and
    Date_Recd = .dr and
    Quantity = .qty
  ) and
  there_isa PART where( Number = .pno and Weight = .wt )
  and .wt_in_k = .qty*.wt/1000
)
// .sno, .pno are not unique key values
  set .myara[ .ono ] = [ .ono, .sno, .pno, .dr, .wt_in_k ];
  set .tuv = myara[ .ono ];
  set .myara_idx2[ .sno, .pno ] = .tuv;
  do Change so_that( [ .dr, .tuv ] Is_In .myara_dr_idx );
  do Change so_that( [ .sno, .pno, .tuv ] Is_In .myara_sno_pno_idx );
// just a check!
  when( .myara[ .ono ] != .(myara_idx2[ .sno, .pno ])
    ) do Exclaim_Line( "error! failed equality check" );
```

As an analog of a C typedef, note the handy Cymbal definition of TU_TYPE. The dynara myara contains the data; the other dynara and the two BOXes map one or more "field" values to a TUPLE VBL that is in effect a pointer to a range TUPLE in .myara, that being the data record. Thus these "index" structures behave like B-trees in that they store and map "field" values to what are effectively pointers to the actual data records. While .myara_idx2 functions as it should, from an application point of view, it is unsatisfactory because, since there are multiple records in ORDER with the same Supp_Nbr and Part_Nbr, only the last of the replicates can be stored in the dynara, which is necessarily
functioning as a Unique INDEX. So that justifies adding `.myara_sno_pno_idx` which is sorted LIST index that effectively maps `.sno, .pno` to each of its range TUPLEs and thus serves as a NonUnique INDEX. As for the check that the `.myara_idx2` index is working, note that, according to Cymbal precedence rules, the parentheses used in `.myara_idx2[ .sno, .pno ]` are absolutely necessary; if removed, the query will rightly fail to compile.

Here are some examples showing how to use these indices for retrieval:

```c
do Write_Words( .(.myara_idx2[ 501, 101 ])#1 );
do Write_Words( .(.myara_idx2[ 501, 101 ] ));

fet TU_TYPE VBL .tuv
ist( [ "8/20/86", .tuv ] Is_In .myara_dr_idx sorted_by_spec[ 1 ] )
{ do Write_Words( ..tuv ); }  // TEST: partial match retrieval
fet TU_TYPE VBL .tuv
ist( [ 422, ?, .tuv ] Is_In .myara_sno_pno_idx sorted_by_spec[ 1 ] )
{ do Write_Words( ..tuv ); }  
```

Each of these `for each` times retrieves multiple records. So, these LISTS are good at serving as NonUnique INDEXes because in these cases, their skip list index serves to speed up access to the data records.

Updates are refreshingly simple: just access the record through any one index and change it. It will necessarily be changed appropriately for all access paths. Here are three separate updates:

```c
set .myara[ 986 ]#4 += 31;
set (.myara_idx2[ 452, 139 ])#4 += 151;

fet TU_TYPE VBL .tuv
ist( [ 400, ?, .tuv ] Is_In .myara_sno_pno_idx sorted_by_spec[ 1 ] )
{ set ..tuv#4 += 31; }
```

In the case of updating through the LIST index (as sped up by its skip list index), multiple range TUPLEs will be updated. This is much faster than doing these updates by scanning the whole dynara `.myara` directly to find entries to update. Caveat: it should be obvious that it is a mistake to update/change the "fields" indexing these records since that will invalidate the indexing; instead, one should accomplish that intent by doing a delete and an insert.

On the other hand, deletes parallel inserts in that each index entry must be dealt with explicitly:
set .tvv = myara[ 986 ];
set .myara_idx2[ ..tvv#2, ..tvv#3 ] = ¬;
// .tvv must be used here instead of ? to identify the right range TUPLE
do Change so_that(
    [ ..tvv#4, .tvv ] Is_Not_In .myara_dr_idx sorted_by_spec[1] );
do Change so_that(
    [ ..tvv#2, ..tvv#3, .tvv ] Is_Not_In .myara_sno_pno_idx sorted_by_spec[1,2] );
set .myara[ 986 ] = ¬;

Note the care to delete the "primary" index last because when it is gone, then .tvv is pointing to freed
storage, i.e., garbage, and so could not be safely used to perform the other index deletes.

Note that when using easy/declarative parallelization, any LIST indices constructed in the parent
are automatically made available to all clones without copying (as usual).

The query dynara.as.tbl.3.Q is a variant of dynara.as.tbl.2.Q that shows a faster way to implement
NonUnique indices for dynara with the sole downside of not being able to use partial match retrieval
(as occurs when ? skolens appear in the subject TUPLE to an Is_In LIST). (Of course, the workaround
to that is just to make more indices instead of trying to get one index to serve multiple purposes.) Here
is the definition of such an index:

SET{ TU_TYPE VBL : with_deletions_ok }
   ARRAY[ INT .sno, INT .pno : with_default @ => {} ] .myara_sno_pno_idx

This dynara maps [.sno, .pno] pairs each to a SET of range TUPLE VBLs corresponding to the
.myara elements they refer to. The relative efficiency comes from the fact that dynara are faster for
lookups than BOXes.

Here is how to put TUPLE VBLs into such an index and how to read them out.

set .myara[ .ono ] = [ .ono, .sno, .pno, .dr, .wt_in_k ];
set .tuv = myara[ .ono ];
do Change so_that( .tuv Is_In .myara_sno_pno_idx[ .sno, .pno ] );
for TU_TYPE VBL .tuv
   ist( .tuv Is_In .myara_dr_idx[ "2013-06-07" ] )
   { do Write_Words( ..tuv ); }
export from_shmem `arenaT`:
  TUPLE[ INT, INT, INT, DATE, FLT ]
  ARRAY[ INT : with_default @ => [ -1, -1, -1, `2000-01-01`, 0.0 ] ] ..myara
  TUPLE[ INT, INT, INT, DATE, FLT ] VBL ARRAY[ INT, INT ] ..myara_idx2
  ...
  set ..myara[ .ono ] = [ .ono, .sno, .pno, .dr, .wt_in_k ];
  set .tuv = ..myara[ .ono ];
  set ..myara_idx2[ .sno, .pno ] = .tuv;
  ...

21.10.2 Concurrent Shmem Dynara As Tables With Multiple Indices

With concurrent shmem dynara, the good news is that now the loading of the dynara table can
be done in parallel and of course, all reading, deleting, updating, and inserting can be done in the
concurrent manner.

So, here is what parallel loading using easy parallelization looks like with concurrent dynara
(shm.conc.paraload.2.IQ):
define PROC txn task: Update_Shmem_Region
{
export concurrent from_shmem `arena3`:
  TUPLE[ INT, INT, INT, DATE, FLT ]
  ARRAY[ INT : with_default @ => [ -1, -1, -1, `2000-01-01`, 0.0 ] ] ..myara
  TUPLE[ INT, INT, INT, DATE, FLT ] VBL ARRAY[ INT, INT ] ..myara_idx2
  set ..tot_sects = 10;
  parallel_for 5
  with_clones_doing_the_do
  fet [ .ono, .sno, .pno, .dr, .wt_in_k ]
  ist(
    .sect_nbr Is_In [ 1 -> .tot_sects ]
    and
    parallelizing(
      there_is_an ORDER from_section[ .sect_nbr, .tot_sects ] where(
        Number = .ono and
        Supp_Nbr = .sno and
        Part_Nbr = .pno and
        Date_Recd = .dr and
        Quantity = .qty
      )
      and
      there_isa PART where( Number = .pno and Weight = .wt )
      and .wt_in_k = .qty*.wt/1000
    )
  ){
    set ..myara[ .ono ] = [ .ono, .sno, .pno, .dr, .wt_in_k ];
    set ..myara_idx2[ .sno, .pno ] = .myara[ .ono ];
  }
}

Note that this mode of parallelization accomplishes its ends by using clones. The other paradigm for using concurrent dynara is to have unrelated processes concurrently accessing the same dynara. For emphasis, note that the non-concurrent shmem dynara cannot productively do parallel loading because the clones would lock each other out and thus effectively serialize the process. And in the case of conventional/non-shmem dynara, the clones would be loading their own private (non-shared) portion which may be of value but is certainly not creating a single dynara for everyone to use.

On the other hand, updating must now be done carefully and explicitly step by step as illustrated in shm.conc.paraload.2.IQ.
import concurrent from_shmem `arena3`
TUPLE[ INT, INT, INT, DATE, FLT ]
    ARRAY[ INT : with_default @ => [ -1, -1, -1, '2000-01-01', 0.0 ] ] ..myara
TUPLE[ INT, INT, INT, DATE, FLT ] VBL ARRAY[ INT, INT ] ..myara_idx2
...
do_final {
    set .tvv = .myara[ 986 ];  // after this ‘set’, .tvv is a copy!
    set ..tvv#4 += 1011;
    // BUT in shmem, all updates are copies -- so must correct the index using the copy
    set ..myara_idx2[ ..tvv#2, ..tvv#3 ] = .tvv;  // sno, pno
}

In the concurrent shmem case, it is no longer sufficient to access the common range TUPLE just once in order to update it. The reason is because concurrent shmem updates begin by copying the target range TUPLE, which will become the new range TUPLE at the end of the concurrent-do. This necessarily invalidates all other indices for that dynara element because they would still be pointing to the old, now deleted, soon enough to be trashed, range TUPLE. Fortunately, as demonstrated above, it is easy enough to update each of the other dynara indices to work with the new TUPLE. Other approaches may not work -- so, use this one! This one is characterized by locating the target range TUPLE from the base dynara, not one of the indices, and then using that to update the indices. Incidentally, there are two ways to abort such an update: call Abort or use a goto. The price is that there will be unreclaimable garbage (i.e., the unused shmem copy of the range TUPLE); so don’t do this too much.

The only incremental change for deletes is that since they use a range TUPLE VBL VBL, they must appear in a concurrent-do:

do_final {
    set .tvv = .myara[ 986 ];
    set ..myara_idx2[ ..tvv#2, ..tvv#3 ] = ~;
    set ..myara[ 986 ] = ~;
}

Finally, of course, the view capturing a parallelized shmem hash-join discussed earlier carries over into the concurrent setting as SQL_CONC_SHMEM_HASH_JOIN_VU in conc.para_hashjoin.1xM.1.IQ.
22. Networking And Distribution

Assuming a basic knowledge of how to do parallelization and programming with pipes and fifos in Cymbal, relatively few additional constructs are needed to write and run Cymbal programs on different computers that communicate with each other over a network. This is done of course using sockets which are a generalization of pipes to networks. Thus, all that is really needed to do Cymbal network programming is a couple of network-extensions to `new_channel()`.

This chapter explains the ins and outs and technical implications of getting Cymbal programs to talk to each other over a network. It goes on to explain how to use a particular application of these ideas, i.e., the `pdq` Daytona query server. The user is welcome to use `pdq` to handle the details of sending and processing Daytona queries remotely and receiving the results. `pdq` is not only useful stand-alone: in fact, it forms the basis for the Daytona JDBC, Perl DBI and PYTHON DBAPI drivers.

In the typical network communication setting, a process on one machine, called the client, wants to get another machine to do something. This can only happen if there is a process, called the server, already running on the other machine which is configured to accept and handle such requests.

This chapter presumes some working knowledge of networking and sockets. The best books in the world for learning how to do network programming at a very deep level are the several volumes of *UNIX Network Programming* written by W. Richard Stevens.

22.1 Simple Network Programming With `_fifo_`

One of the simplest ways to do interprocess communication over a network is to run a server daemon process that is always attempting to read commands from a CHAN(_fifo_) on that machine. All the client process then has to do is to use ssh or rsh to cause the server machine to write the desired command into the _fifo_. Here is the code for a simple server which will just echo whatever lines it finds in the _fifo_ up to some agreed upon number (fifo_u.4.IQ):
local: STR: .file, .line; INT: .cnt;

set [.cnt, .file] = read( from _cmd_line_ but_if_absent [ 3, "fifo_file" ] );
set .fifo = new_channel(via _fifo_ for .file with_mode _update_)
otherwise{ do Exit( 1 ); }
while( .cnt > 0 ) renewing_with { set .cnt--; } do {
  set [.line] = read_line( from .fifo )
  otherwise_switch{
    else { do Exclaim_Line( "error: strange exit status in fifo_u.4.IQ" ); }
  }
  flushing do Write_Line( .line );
}
// this daemon process will exit after reading the agreed upon messages
// otherwise, it would have remain blocked on reading more from the fifo
do Close( .fifo );

And here is a sample client invocation to the server running on distant.machine.org:

    rsh distant.machine.org 'echo Greetings from afar > ~services/fifo_file'

### 22.2 Socket-Based Client-Server Basics

The taxonomy here is as follows. Both client and server processes must open appropriate I/O channels to read and write to, where a single CHAN supports both reading and writing. The CHAN is obtained by calling `new_channel()` with the `via` argument equal to either `_tcp` or `_unix_domain` and with the `mode` being equal to one of `_client`, `_iterative_server`, or `_concurrent_server`, with `_client` being the default. (_unix_domain_ is not yet implemented.)

An iterative server is one which can only handle one client at a time from start to finish. A concurrent server is able to handle multiple clients simultaneously by virtue of creating one child process per active client to handle the connection for that client. As discussed towards the end of this chapter, in Cymbal, a concurrent server is created by using TENDRILS in conjunction with a CHAN(_concurrent_server_).

As with any conversation, someone has to say goodbye first, which in this setting is called doing the active close. Thus two network programming paradigms arise according to which of the client and server do the active close, the other close being the passive close. (On the other hand, only the server can say hello first since in socket programming, if the server has not been started, its machine will simply tell any would-be clients that the expected service is not available (_conn_refused_).)

Here is the specification for the CHAN(_tcp_) `new_channel` FUN that Daytona currently supports:
otherwise_ok
CHAN(_tcp_) FUN new_channel_socket( // a variant of overloaded new_channel
   with_name STR(=) = _default_name_,
   (0) for_type STR(=) = "",
   via manifest _3GL_TEXT = _tcp_ ,
   for_addr STR(*)|IPORT(?)|IHOST_PORT /*|ISVC|IHOST_SVC|UDPATH*/,
   with_mode manifest INT = _client_,
   with_patience manifest INT = _wait_on_block_,   // for _client_ only
   with_bufsize INT = -1,
   (0) with_locking = Ä,
   (0) with_whole_msgs = Ä,
   (0) with_msg_terminator STR(=) = "0,
   with_max_q INT = 20   // for _iterative_server_ only
)

Note that the for_addr can take arguments that have been explicitly typed as IPORT or IHOST_PORT or alternatively, a STR argument which of course has to be convertible at runtime to either an IPORT or IHOST PORT. So, in the former case, compile-time typechecking will prevent runtime errors while in the latter case, offering an unconvertible for_addr STR argument to new_channel will result in an error which is fatal by default but can be caught by using a try block looking for exception "new_channel_socket: bad STR for_addr". The reason for allowing STR for_addr arguments is that in an environment which has to support both IPv4 and IPv6, and where ip_str_for_ihost is returning a STR that can represent either, by allowing a STR for_addr argument to new_channel_socket(), the user can avoid writing code that contains a switch statement with multiple cases invoking different fully typed new_channel_socket() calls.

The with_max_q argument tells the system not to queue up any more clients than that specified number.

22.3 Where The Server Says Goodbye First

Here is the complete Cymbal for a client process that sends a single one-line shell command to the server and prints out the multi-line response (and then quits) (tcli.=1.1.IQ).
set [.iport] = read( from _cmd_line_ bia[ "127.0.0.1:12001" ] )
otherwise{ do Exit(1); } // handles malformed IPORT
set .to_srvr_chan = new_channel( for_addr .iport with_mode _client_ )
otherwise { do Exit(2); }
do Write("Enter a one-line command to be executed: ");
set [.line] = read_line(bia[ "" ])
otherwise do Exit(3);
flushing to .to_srvr_chan do Write_Line(.line);
do Write_Line("Command expands into:");
loop {
    set [.line] = read_line( from .to_srvr_chan bia[ "" ])
    otherwise_switch{
        case( = _instant_eoc_ ){ break; }
        else { do Exit(4); }
    }
do Write_Line(.line);
}
do Close(.to_srvr_chan);

Obviously, the client has to know which IP address and port the server is listening on. As is customary
with pipe-like I/O channels, it is necessary to flush the message buffer when programmer wants the
message to be sent. Also, it always good practice in this kind of programming to explicitly call Close
since the closing of a socket causes the kernel to send out the FIN packets needed to help properly
terminate a TCP conversation.

Here is the code for the server that will do what this client wants, which is to evaluate a single
one-line command (itsvr.=1.1.IQ):
local: STR .line
do Write_Words( "server starting at", shell_eval( "date" ) );
set [ .iport ] = read( from _cmd_line_ bia[ ^127.0.0.1:12001^IPORT ] )
    otherwise{ do Exit( 1 ); } // handles malformed IPORT
set .listen_chan = new_channel(
    for_addr .iport with_mode _iterative_server_ with_max_q 50 )
    otherwise{ do Exit( 2 ); }
loop {
    set .client_chan = new_channel( for .listen_chan with_mode _accept_ )
    otherwise { do Exit( 3 ); }
    set [ .line ] = read_line( from .client_chan )
    otherwise with_msg "bad read" do Exit( 4 );
    to .client_chan do Write( shell_eval( .line ) );
    do Close( .client_chan );
}
do Close( .listen_chan );

Note that a server has to get two kinds of CHAN(_tcp_), one to listen for incoming connections and then another for each new such client to communicate with that client, leaving the original channel free to listen for yet more clients. Here is the new_channel declaration specifying how to get a client channel from a server listening channel.

CHAN(_tcp_) FUN new_channel_accept( // a variant of the overloaded new_channel()    with_name STR = _default_name_,
    via _3GL_TEXT = _tcp_,
    for CHAN(_tcp_),
    (0->1) with_mode manifest INT = _accept_,
    with_bufsize INT = -1,
    (0->1) with_client_vbl manifest alias STR = accept_client )

It is the CHAN(_tcp_) type of the for argument that is sufficient to determine the use of this particular variant of new_channel which is the _accept_ variant.

Note that since the protocol is to evaluate just one command per client connection, it makes sense for the server to do the active close once the result of evaluating the single command has been sent to the client channel. Indeed, if the Close call is omitted, then this client will hang indefinitely waiting for its read_line to return _instant_eoc_, which would only happen then when the server process itself died, whenever that would have been.

The ancillary VBL argument with_client_vbl allows the user to get a STR representation of the IPv4-based or IPv6-based IPORT belonging to the client.

As it turns out, there are three kinds of IPORTS to use in specifying a server, depending on the IP address portion of the IPORT. The first is where the IP address is for one of the interfaces on the host machine, the second is where it is the local IP address 127.0.0.1 that is used for two processes communicating on the same machine, and lastly, there is the wildcard IP address 0.0.0.0 which on a
multi-homed machine implies that connections will be taken from any of the active network interfaces. As for choosing a suitable port for a server to advertise for its services, ports from 1024 through 49151 are recommended, if they are not already likely to be used by other registered network services available on the given machine.

Once again, an IHOST_PORT can also be used to specify the server+service desired.

### 22.4 Where The Client Says Goodbye First

Suppose that the client has an unpredictable number of shell commands to get evaluated during any given TCP conversation with the server. Obviously then, only the client will know first when the conversation is over. Here is such a client (tcli.1+.2.IQ):
Of interest here is the identification of all user-handlable error conditions for using new_channel to get a _client_ mode CHAN(_tcp_). The new_channel call status is _conn_refused_ if the server says that there is no service being offered at the requested port (specified in the IPORT), or under some circumstances, when the server’s queue for that service is full. The call status is _host_unreachable_ if the network sends back an ICMP message saying that it doesn’t know how to reach the host. The call status is _timed_out_ if for example, the remote machine had been successfully connected to the network but someone had just turned the machine off. It can take over 3 minutes and 44 seconds for a connection to time out. If that is too long, then using a smaller _with_patience_ argument will shorten it. The call status is _addr_unavail_ if the IPORT is a one that can’t be worked with like `0.0.0.0:0`IPORT. The call
status is _iport_forbidden_ if the user does not have the permissions necessary to work with the given IPORT. This can happen if firewall rules forbid making such connections or if a non-root user is trying to access a low-numbered port.

Notice that the client is using Cymbal’s support for (multi-line) messages by using the keyword ending_with to define the end-of-message token. The server uses the same mechanism to write its multi-line answer back to the client (itsvr.1+.1.IQ):

```
local: STR .line, .client
set [ .iport ] = read( from _cmd_line_ bia[ `0.0.0.0:12001`IPORT ] )
  otherwise{ do Exit(1); } // handles malformed IPORT
set .listen_chan = new_channel( for .iport with_mode _iterative_server_ )
otherwise_switch {
  case( = _addr_in_use_ ){ with_msg "address in use: need to wait or use different port" do Exit( 2 ); }
  case( = _addr_unavail_ ){ with_msg "no interface has this address "+(STR)ip_of(.iport) do Exit( 3 ); }
  case( = _iport_forbidden_ ){ with_msg "forbidden to open port "+port_of(.iport) do Exit( 4 ); }
  default{ do Exit( 5 ); }
}
loop {
  set .client_chan = new_channel( for .listen_chan with_client_vbl client )
  otherwise_switch {
    case( = _conn_aborted_ | = _iport_forbidden_ ){ continue; }
    else{ do Exit( 6 ); }
  };
  do Write_Line( "The next client is .client"ISTR );
  loop {
    set [ .line ] = read_line( from .client_chan bia[ "" ] )
    otherwise_switch{
      case( = _instant_eoc_ ){ break; }
      else { do Exit( 7 ); }
    }
    flushing ending_with "\n####\n" to .client_chan do Write( shell_eval( .line ) );
  }
  do Close( .client_chan );
}
do Close( .listen_chan );
```

When opening a channel to listen for IP traffic on, .new_channel_call_status will equal
_addr_unavail_ if the IPORT is not associated with any active network interface.
_.new_channel_call_status_ will equal _iport_forbidden_ if the permissions that the process is
running under do not allow the process to open the specified port. _.new_channel_call_status_
will equal _addr_in_use_ if another process is currently listening at the specified network interface and
port. Due to the nature of TCP, the _addr_in_use_ situation also occurs after such a server has executed
an active close and exited as a process. For a period of time (depending on the OS) ranging from 1 to 4
minutes after server process exit, the server’s IPORT is still considered to be in use; in fact, the
associated TCP connection is considered to be in the TIME_WAIT state. The user can verify this by
examining the output of _netstat -a_, a generally useful command to use when debugging network
programming problems:

<table>
<thead>
<tr>
<th>Local Address</th>
<th>Remote Address</th>
<th>Swind</th>
<th>Send-Q</th>
<th>Rwind</th>
<th>Recv-Q</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>localhost.12001</td>
<td>localhost.64236</td>
<td>57344</td>
<td>0</td>
<td>57344</td>
<td>0</td>
<td>TIME_WAIT</td>
</tr>
</tbody>
</table>

In this case, the defunct server was listening on port 12001. The purpose of the TIME_WAIT state is to
ensure that any packets associated with conversations involving the previous server incarnation have
enough time to be discarded by the network on the basis of having lived too long to be of further
interest.

When opening a channel to accept an incoming TCP connection, there are only two failures that
Daytona offers the ability to catch and recover from: _conn_aborted_ and _iport_forbidden_.
_conn_aborted_ happens when the client sends a RST packet and aborts the attempted connection for
whatever reason.

Note the useful with_client_vbl option to _new_channel_ when it is creating an accept or client channel
from an listening channel. This keyword takes an ancillary variable as an argument whose IPORT
value identifies the next client whose request for a conversation was accepted.

### 22.5 A Concurrent Server Template

Instead of handling each client request to completion before moving on to the next request as
an iterative server does, a concurrent server clones a child process to handle each client request as it
comes in. This leaves it free to accept new requests while its children are active handling previous
requests. Note that a _concurrent_server_ must also include customizable code to create and look after
the child processes that are actually handling the requests. The following query consvr.2.IQ gives an
example of a concurrent server. This can be adapted or customized to new applications in two kinds of
ways. The first has to do with the operation of the server as a server, e.g., specifying what the default
IP address is that the server listens to, what does the server do with its children who run into trouble
executing their tasks, how does the server log the work it does, what is the maximum number of
children it handles, etc. The second kind of adaptation of course is specifying exactly the procedure by
which each child handles client requests. That is captured below in the _Work_With_Client_CHAN_ task
which would be rewritten to address the specific needs of the application.
/*
This gets a sequence of one-line commands from any given client
and evaluates them into one or more lines of output.
The client does the active close.
Works with clients tcli.1+[123].IQ
*/
local: STR .line; IPORT .client

// customized for application: could be more than one
import: PROC( CHAN(_tcp_) .cli_chan ) Work_With_Client_Chan
set [ .iport, .max_child ]
    = read( from _cmd_line_ bia[ "0.0.0.0:12001"IPORT, 20 ] )
    otherwise( do Exit(1); ) // handles malformed IPORT

set .listen_chan = new_channel( for_addr .iport with_mode _concurrent_server_ )
otherwise_switch {
    case( = _addr_in_use_ ){
        with_msg "address in use: need to wait or use different port"
        do Exit( 2 );
    }
    case( = _addr_unavail_ ){
        with_msg "no interface has this address "+(STR)ip_of(.iport)
        do Exit( 3 );
    }
    case( = _iport_forbidden_ ){
        with_msg "forbidden to open port "+port_of(.iport)
        do Exit( 4 );
    }
    default{ do Exit( 5 ); }
}

set .child_bndl = new_bundle();
loop {
    do Cleanup_Children;
    // Don't accept more than the maximum number of requests.
    when( .child_bndl.Tendril_Cnt >= .max_child )
        { do Sleep( ^.5s"TIME"); continue; }
    set .cli_chan = new_channel(
        for .listen_chan with_patience 5 with_client_vbl client )
    otherwise_switch {
        case( = _conn_aborted_ | = _iport_forbidden_ | _timed_out_ )
            { continue; }
        else{ do Exit( 6 ); }
    }
}
// in general, choose which application PROC to call with
do Handle_Client( Work_With_Client_Chan );

do Close( .cli_chan );
}
do Close( .listen_chan );

define PROCEDURE : Cleanup_Children()
{
  local: TENDRIL .oldkid
  loop {
    set .oldkid = next_waited_for_tendril(
      for_bundle .child_bndl with_patience _fail_on_block_ )
    otherwise_switch {
      case( = _no_such_kid_ | = _would_block_
        | = _timed_out_ | = _interrupted_ )
      {
        // These are reasonable returns when no children are waiting.
        // Break out of wait loop to look for new requests.
        break;
      }
    }
    switch_on( .oldkid.Status.Kind ){
      case( = _exited_ ){
        when( .oldkid.Status.Value = 0 ){
          flushing do Write_Words( "Child", .oldkid.Index,
            "(" , .oldkid.Sys_Id.Pid, ") terminated normally" );
        } else {
          do Exclaim_Words("Child", .oldkid.Index, "(" , .oldkid.Sys_Id.Pid,
            ") failed, returning", .oldkid.Status.Value );
        }
      }
      case( = _killed_ | = _stopped_ ){
        do Exclaim_Words( "Child", .oldkid.Index, "(" , .oldkid.Sys_Id.Pid,
          ") received signal", .oldkid.Status.Value );
      }
      do Free( .oldkid );
    }
  }
}

define PROCEDURE( PROC( CHAN(_tcp_) ) .custom_handler )
Handle_Client
{
  local: TENDRIL .newkid
set .newkid = new_tendril(
    for_bundle .child_bndl
    executing(
        do Close( .listen_chan );
        do .custom_handler( .cli_chan );
        do Close( .cli_chan );
    )
);
flushing do Write_Words( "Child", .newkid.Index, "(",
    .newkid.Sys_Id.Pid, ") will handle client on", .client );
}
global defs:
// one of possibly several application-specific PROCs
define PROC( CHAN(_tcp_) .cli_chan ) task:
    Work_With_Client_Chan
{
    loop {
        set [ .line ] = read_line( from .cli_chan bia[ "**STR(*) ] ]
     otherwise_switch{
            case( = _instant_eoc_ ){ break; }
            else { do Exit( 7 ); }
        }
        flushing ending_with "
    to .cli_chan do Write( shell_eval( .line ) );
    }
}

This concurrent server begins by establishing a CHAN(_tcp_) for a _concurrent_server_ to listen on. The utility of creating the listening channel using _concurrent_server_ is that the SO_REUSEADDR socket option is applied to condition the listening channel. In simple language, this means that it becomes possible to restart a server that has active children busy handling requests.

The program continues with an unconditional loop that first handles the bookkeeping for any children that have finished. Then it refuses to clone more children and sleeps instead if that would make too many in service. Otherwise, it listens for new clients. If it times out listening for new clients, then it loops around to clean up any children that have finished in between time. Once it gets a client, it calls Handle_Client which clones a child process to do the work specified by the application-driven (and provided) Work_With_Client_Chan task.

22.6 pdq Network Query Server

As should be clear by now, Daytona does not have any daemon system processes that are needed to support querying on the same machine as the data. This is in great contrast to traditional relational DBMS which have many daemon server processes running all the time. However, in a network setting, in order to handle queries initiating from remote systems, it is simply inescapable that
there has to be a daemon process always running and listening on a socket for work to do.

Daytona’s network query server is a concurrent server called *pdq* for *Polyclient Daytona Query server*. In contrast to the daemon server processes employed by traditional relational DBMS which try to implement entire operating systems, *pdq* is really just a straightforward networking program written using the ideas developed in the first part of this chapter to serve as a kind of remote shell interface to the usual Daytona shell commands for compiling and executing Cymbal queries. In this way, *pdq* is like a network version of the shell *DS* command or the *Dice* GUI interface. *pdq* employs a simple ASCII protocol by which remote (or even local) clients may ask for and receive query services. Here is an example of (three) client messages to send a query, run it, get the answers back and then terminate the connection:

```
DS_OPEN cym robert
#####
DS_QQ
    with_title_line "Get phone numbers of suppliers from St. Paul"
do Display with_format _table_
each [ .supplier, .phone ]
each_time(
    there_is_a SUPPLIER where(
        Name = .supplier
        and Telephone = .phone
        and City = "St. Paul"
    )
);
#####
DS_CLOSE
#####
```

Clearly, the message syntax is simple: it starts with a command and ends with the `#####` end-of-message token. *pdq* is explained in an extensive man page included in this document as well as the Daytona man pages themselves.

Of course, a client program has to be written to send these messages and receive the answers back. Just like *pdq.cy* itself, this can be accomplished by writing a Cymbal program making special use of the features explained in this chapter. In fact, that has already been done in a generic way with the Cymbal program *pdc* that comes with *pdq*. For more information on that, just do *DS Man pdc*.

As it turns out, *pdq* supports other clients as well. These include JDBC, Perl DBI, and Python DBAPI. JDBC is a Java-based interface specification that enables Java programs to access database functionality for any DBMS that writes a driver that listens and speaks the JDBC API. Daytona has such a JDBC driver thus enabling, for example, Java GUIs to access Daytona databases from Java. Crystal Reports (and Business Objects in general), for example, has a JDBC interface. Likewise, Daytona has a Perl DBI database driver and Python DBAPI driver.
23. Advanced Topics For Record Class Descriptions

When record class descriptions are annotated appropriately, Daytona will support multiple files per record class (horizontal partitioning), retrieving FILE information at run-time, stdin, pipe and fifo data sources, index-free retrieval, in-memory data files, fields constructed by means of user Cymbal and C functions, data compression, and schema evolution. As a forward-looking comment, please note that an appendix contains a grammar describing the supporting data dictionary constructs.

23.1 Horizontal Partitioning

23.1.1 Multiple Files Per Record Class

On any given computer, there comes a time when data file sizes become so big that the files become unwieldy. Either there isn’t enough room for them on the disk (or even on the machine) or the time it takes to make their indices is so substantial that it begins to inhibit making changes. To this end, Daytona enables the records for a single record class to be stored in arbitrarily many files. Most importantly, this horizontal table partitioning is done in such a way that the end user is totally unaware of whether it is happening or not. The multiple files for a record class may all be on the same disk, or on different disks or via the magic of a network file system, on one or more different machines.

The theory behind the way Daytona accomplishes this is simple. At the conceptual level, an object record class is observed to have one or more attributes whose values serve to define partitions of that record class. For example, consider billing data which is coming into the database in monthly installments from different regions of the country. The goal is to be able to easily add new months of data to the system as well as to just as easily remove data for older months as it becomes less and less interesting. A secondary goal is to be able to quickly search all the billing activity for a given region without having to scan records belonging to other regions. So, clearly, all the data for each month ought to go into its own files according to the region of the country it came from. Conceptually, the data modeller would observe that the Month and Region attributes for the object records in the BILLS class serve to generate partitions of the records on the basis of what their month and region are. Or to put it differently, for any given Month and Region, all the BILLS records with that month and region would be considered to be in the same record partition.

Or, to be mathematical about it, if the TUPLE of partition attributes is considered to be a function which maps object records into TUPLES of partition attribute values, then the partitions are the level sets of this function, i.e., the inverse images of given attribute value TUPLES.

Having identified a group or TUPLE of partition attributes for an object record class, the Daytona user is free to divide up the corresponding data file records into files where the data file records in each file all have the same values for the partition attributes.

In terms of the BILLS example with its two partition attributes Month and Region, the BILLS data file records would be partitioned into files according to their month and region. It would be acceptable...
to have two files whose records all had the same Month and Region but it would not be acceptable (and would, in fact, be impossible) for the records in any given file to have more than one value for the Month or Region partition attributes.

Knowing that all records in a given file have the same values for the partition attributes enables Daytona (and the user) to store the values for those attributes in the record class description instead of in the actual data file where they would, of course, be grossly redundant.

```
select *
from BILLS
where Region = 'NW'
and Amount > 100.0
```

23.1.2 Partitioning, Not Subclassing

This process is called horizontal partitioning because if the records in a horizontally partitioned
record class are sorted according to their partitioning FIELDS, then each partition of records is identifiable by cutting across the table horizontally at the points where the values for the horizontally partitioning FIELDS change. Obviously, another way to split up the information in a record class is to partition it vertically into files. In that case, groups of FIELDS are identified and all the values for each group are stored in their own files. In that way, a single object record is being implemented as several data file records being stored in each of several files, which is quite the opposite from horizontal partitioning.

Since each (horizontal) partition of records is a subset of the record class, they could be thought of as being subclasses of the record class. However, in Daytona, the word subclass is reserved instead to refer to the situation where it is the objects themselves, not their RECORDS, that are organized into classes and subclasses. This can happen in at least two ways.

One is the restriction-subclassing setting where additional membership conditions are placed on objects in order to generate new, smaller classes of objects which are clearly subclasses of their parent classes, whose members satisfy less restrictive conditions. For example, a BLUE_CAR is a CAR. In this case, the BLUE_CAR RECORDS have the same FIELDS as any CAR RECORD.

The other is the familiar object-oriented, isa generalization-hierarchy, where for example, a CAR is a VEHICLE. What typically happens when classes of objects are nested in a generalization hierarchy is that the information available to describe the members of a subclass is greater than that for (proper) members of any superclass, whereas of course, the information available to describe superclass members is necessarily available to describe subclass members. For example, describing a CAR by mentioning its transmission characteristics is not possible for a VEHICLE not known to be a CAR whereas the Speed attribute for VEHICLES clearly is available to describe CARS as well since a CAR is a VEHICLE. Consequently, a RECORD that describes a CAR will have all the FIELDS that are used to describe VEHICLES as well as more that are proper to describing CARS alone. So, the subclass (subset) relationship is really between the object CLASSes CAR and VEHICLE themselves and not (strictly speaking), between their corresponding RECORD CLASSes. However, for convenience, Daytona will nonetheless abuse the terminology and say anyway that the CAR RECORDCLASS is a subclass of the VEHICLE RECORDCLASS.

In short, this is just a long way of saying that in the Daytona context, the word ‘subclass’ will be used in either the restriction or generalization-hierarchy sense which is very much different from either horizontal and vertical partitioning. Indeed, horizontal and vertical partitioning have to do fundamentally with splitting up the information about objects into multiple files in the filesystem-based implementation, a goal that is not central or even relevant to the notion of subclassing.

23.1.3 Horizontal Partitioning As Represented In Rcds

So, here are the steps involved in creating and using horizontal partitioning (hparti) in Daytona (examples follow):

1. Determine what the partition FIELDS are for the given situation and define them under the FIELDS description in the rcd with the special Is_A_Partitioner role.
2. Divide up the data into files where the records in any given file all have the same values for the partitioning FIELDS. (Remember that the partitioning FIELD values are not stored with the data.)

3. Include a PARTITIONING_FIELDS description under each FILE description under the rcd’s BINS description to state which FILE is associated with which partitioning FIELD values.

4. Define any keys for the record class in the usual way. Any associated indices will be built on a per FILE basis.

5. Write and run queries, feeling free to make reference to the partitioning FIELDS in queries in the same manner as regular partitioning attributes.

Note that Daytona requires that the horizontal partitioning (hparti) FIELDS comprise a unique key for reaching the data files; in other words, for any given set of values for the FIELDS, there is at most one data file that is associated with them. Among other things, this can help to prevent poor load balancing when using parallelization strategies on horizontally partitioned tables. If the horizontal partitioning FIELDS that seem reasonable to the user do not have this uniqueness property, then it is easy to rectify the situation by introducing a new, last horizontal partitioning FIELD sequence number that is artificial in nature in that its purpose is to unquify the TUPLE of all horizontal partitioning FIELDS. There is no need, in general, to refer to that uniquifying FIELD in queries but it can be referred to in specific situations to help out, as is the case when parallelizing so as to split each bin among several clones. See rcd.ORDERB for an example of introducing a uniquifying attribute.

Recall that the notion of a Unique KEY is a BIN (i.e., data file) notion, not a RECORD_CLASS notion. In other words, KEY uniqueness is only guaranteed within each data file. Should it be needed then, the way to get a truly unique key over the entire RECORD_CLASS is to concatenate the horizontal partitioning KEY with a Unique KEY on non-hparti FIELDS.

The only other restriction is that hparti FIELDS must be scalar-valued, i.e., they cannot be LIST/SET-valued. Other than that, they can have any scalar datatype (that can be Written) and there can be as many of them as desired.

Daytona will take the hparti information provided by the user and will add additional notes and description subtrees to the rcd. The subtrees will have <__System_Generated, yes> notes attached; that’s a signal for the user to ignore them.

Here are the FIELD descriptions for the partitioning FIELDS for rcd.PARTED in $DS_DIR/EXAMPLES/usr/orders/aar.orders. Recall that since Daytona generates the __Position notes, the user should not pay any attention to them. The Default Value notes are used to specify what value the system should assume is the FIELD value when the user is adding a new record to a table and hasn’t specified a value for that FIELD.
Here is an example taken from rcd.PARTED that illustrates how each one of PARTED's FILEs would be described under the BINS node.
The descriptions of other FILEs for this record class are siblings of the FILE description shown. Notice that the FILE description of PART.1 is an ordinary one up to its PARTITIONING_FIELDS child. Here for each of the partitioning FIELDs, the user must supply the particular Value that that FIELD has for every record in the associated FILE. Please note that an appendix contains a grammar for these data dictionary constructs and others.

There is one technical consideration. The fastest processing of an hparti record class will occur when all the files or bins that belong to it and are referenced more than once in the execution of the query are open at the same time. Unfortunately, the operating system may have unexpectedly low limits on the maximum number of files that can be open at any given time -- although to some extent, individual users can raise their limit by using ulimit and in general, sysadmins can raise the system-wide limit well into the thousands. At any rate, in order to avoid attempting to violate the current limit, an appropriate integer, even as low as 1, may need to be the value of the RECORD_CLASS’s Max_Open_Bins attribute instead of _all_, which itself means the total number of bins that the system can find when it looks, i.e., at query compile time for static hparti and at query runtime for dynamic hparti.

Finer grained control over how many bins are open at a time may be gained by using in addition
the more specific Max_Open_Seq_Acc_Bins and Max_Open_Random_Acc_Bins, which correspond to the maximum number of bins that can be opened during sequential or random accesses (of data files), respectively. For example, since many sequential accesses only visit a bin once, there is no advantage in keeping more than one bin open at a time and so a value of 1 for Max_Open_Seq_Acc_Bins is ideal. (One exception occurs when doing blind appends (which are sequential) that add records to more than one bin, in which case to avoid the thrashing concomitant with lots of closing and re-opening files, Max_Open_Seq_Acc_Bins should be set equal to the number of bins that are being appended to.) On the other hand, a random access query could well search and re-search several bins and therefore, it would be good to keep all or many of them open to avoid the costs of too many opens and closes due to revisiting but obviously only if the operating system / environment can be tuned to allow it. However, if one needs to support the querying of a table horizontally partitioned into thousands of data files, then probably the best value for Max_Open_Random_Acc_Bins is 1 in the event that it is prima facie unlikely that the query will visit the same BIN more than once.

Please note that it is perfectly OK if query processing attempts to open more than the appropriate number of open bins for a given RECORD_CLASS. What will happen in that instance is that the least recently used bin will be closed to make way for opening the latest requested bin.

However, if there are too many open bins across all record classes, the operating system will declare that there are "too many open files" (or if pipe_ files are being used, "too many zombie/defunct child processes") and an often fatal error condition will arise in the program. As to how many file openings are associated with a bin for a given RECORD_CLASS, it varies. Firstly, it varies by the number of accesses: there is at most one RECORD_CLASS access per there_isa and each access maintains its own policy on the number of open bins. For an access that does not do any updates, there is one file opening for the data file and then an additional one for any index used (including the .siz file) unless it is a cluster B-tree in which case both a B-tree and the .siz file would be opened. Any access that does the updating for a given RECORD_CLASS in a transaction (and there will only be one) has to open the data file, all index files (including the .siz file) as well as the free tree if it is present. These rules give an upper bound on the number of open files since Daytona may find a way to open fewer. Clearly, when updating a bunch of tables in a single transaction, it may be necessary to raise ulimit nofiles or to use close_on_return.

In contrast to the value of Max_Open_Bins which must either be _all_ or an INTEGER constant, the values of Max_Open_Seq_Acc_Bins and Max_Open_Random_Acc_Bins may in addition be shell expressions that will be echo’d at runtime to yield an INT (or _all_) value; either of these latter Max_Open_ values must evaluate to a positive integer not greater than Max_Open_Bins, provided that in the dynamic hparti case, Max_Open_Bins is a positive integer (i.e., not _all_).

<Max_Open_Random_Acc_Bins, ${MAX_OPEN_RAND_BINS:=_all_}>

For users who wish to force Daytona to close RECORD_CLASS accesses, this can be accomplished by placing them in a transaction task that uses the close_on_return concept.

### 23.1.4 Static Horizontal Partitioning With FILE_INFO_FILE

Originally, it was anticipated that horizontally partitioned tables would only contain 10 or 15 bins at most. However, almost from the outset, users wanted to store single tables using hundreds of
bins. Fortunately, the mechanism described in the previous section will support this; the only
downside is that the rcd can become enormous and therefore, somewhat awkward for the DBA to store
and edit. In particular, Daytona customers have stored tables with 18144 bins in this way; they tended
to use backtalk to edit their rcd until the FILE_INFO_FILE feature arrived.

The FILE_INFO_FILE feature is a convenience which enables rcds for a horizontally partitioned
table to have a size independent of the number of bins being used to implement the associated table.
Essentially, FILE_INFO_FILE provides a way for the rcd to point to a Daytona data file which contains
the uniquely descriptive information about each bin (so that this information can be kept out of the rcd
file itself). (The reason why horizontal partitioning in this section is called static is because the FILE
information stored in the file-info file is compiled into each executable that refers to that table:
consequently, executables have a static unchanging-at-runtime understanding of what is in the table.
Dynamic horizontal partitioning will be discussed later in this chapter.) Here is an example BINS from
the ORDERA record class in the orders application:

```bash
#{ BINS
   <Max_Open_Bins, 2>
   <Default_Random_Acc_Bufsize, 1024>
   #{ FILE_INFO_FILE ( ordera_fls )
      <Source, $ORDERS_HOME>
      <Unit_Seps, |:>
      <Tuple_Delims, "[]">  
      <__Position, 1>
   }
   #{ FILE ( __DUMMY )
      <Unit_Sep, :>
      <Comment_Beg, #>
      <Source, __DUMMY>
      <Indices_Source, __DUMMY>
   }
   #{ PARTITIONING_FIELDS
      #{ FIELD ( Region ) <Value, __DUMMY> }#
      #{ FIELD ( Category ) <Value, __DUMMY> }#
   }#
}
```

The FILE_INFO_FILE ordera_fls is the name of a Daytona data file. The FILE_INFO_FILE description
indicates that the Source of ordera_fls is $ORDERS_HOME, that its first Unit_Sep is |, that its second
Unit_Sep is :, etc. Here are several lines from ordera_fls:
FILE_INFO_FILEs like _ordera_fl_ consists of records whose format is:

1. The Name of the bin
2. The Source of the bin
3. The Indices_Source for the bin
4. A TUPLE-valued FIELD containing in order the partitioning FIELD values that pertain to the associated bin.

Notice how the delimiters used in _ordera_fl_ correspond to those specified in the FILE_INFO_FILE description. For the first _ordera_fl_ record given above, its bin is being said to have value 7 for the Region FIELD and "B" for the Category FIELD.

Daytona assumes that the records in the FILE_INFO_FILE are sorted lexicographically by the Cymbal ordering of the partitioning FIELD values. This may or may not be identical with how UNIX sort(1) would sort the records on that list-valued FIELD.

There must be exactly one FILE description nested within the FILE_INFO_FILE description. This is taken to be a generic description of the bins listed in the FILE_INFO_FILE. For example, the sole Unit_Sep for each bin listed in _ordera_fl_ is ::. Notice that the Self, Source, and Indices_Source attributes for the FILE description all have __DUMMY values: this is because the actual values for the bins in question are kept in the FILE_INFO_FILE. The same holds for the partitioning FIELD values.

There is no provision for storing index statistics for each bin described using FILE_INFO_FILE. Index statistics for the last bin run by Sizup will be stored under the FILE description unless <Maintain_Index_Stats, no> is used.

As illustrated in rcd.ORDERB from the sample orders application, a BINS description can contain both FILE and FILE_INFO_FILE sub-descriptions and even multiple FILE_INFO_FILE sub-descriptions themselves. rcd.ORDERB and its FILE_INFO_FILEs also illustrate that if a Source value is missing from the FILE_INFO_FILE data records, then the Source will be determined from the Default_Data_File_Source note under the FILE_INFO_FILE description, if any, or else (as usual) from the Default_Data_File_Source note under the ancestor BINS description, if any, or else from the Default_Data_File_Source note in the associated apd, if any, else from the directory of the aar.

While, in principle, thousands of bins can now be easily used to implement a table, some restraint is usually warranted. As the number of bins goes up, the time it takes to join from another table to the horizontally partitioned table can be overwhelmingly dominated by the time it takes just to open and close the bins.
23.1.5 there_is_a_bin_for

There are occasions where it is desired to use Cymbal to find out information about the bins themselves like, for example, how many there are or what the different possible values are for the horizontal partitioning attributes. Using there_isa is an imperfect tool since there_isa SUPPLIER is referring to instances of SUPPLIER records, not to the bins themselves containing SUPPLIER records. So, a new construct is needed and that construct is there_is_a_bin_for, as illustrated by (binquery.1.Q):

```verbatim
with_format _table_
do Display
  each[ .date_added, .info_source, .group_nbr, .rating ]
each_time(  
   there_is_a_bin_for PARTED where(  
    Date_Added = .date_added  
    and Info_Source = .info_source  
    and Group_Nbr = .group_nbr  
    and Rating = .rating
   )
  );
```

This query will produce one line of output per PARTED bin, not per PARTED record. Clearly, it would be illegal to refer to the value of a non-horizontal-partitioning attribute when using there_is_a_bin_for. there_isa_bin_for is a synonym.

there_is_a_bin_for can take the following alias VBL keyword arguments (sys.env.cy):

```verbatim
( 0->1 ) with_bin_vbl alias STR(*),  
( 0->1 ) with_source_vbl alias STR(*),  
( 0->1 ) with_indices_source_vbl alias STR(*),  
( 0->1 ) with_rec_map_spec_file_vbl alias STR(*),  
( 0->1 ) with_mode_vbl alias INT,  
( 0->1 ) with_mod_time_vbl alias INT,  
( 0->1 ) with_acc_time_vbl alias INT,  
( 0->1 ) with_stat_time_vbl alias INT,  
( 0->1 ) with_byte_size_vbl alias INT(_huge_),  
( 0->1 ) with_nbr_bin_recs_vbl alias INT(_huge_),  
( 0->1 ) with_nbr_bin_rec_slots_vbl alias INT(_huge_),  
( 0->1 ) with_bin_min_rec_len_vbl alias INT,  
( 0->1 ) with_bin_max_rec_len_vbl alias INT,
```
the bin, and the length of the largest record in the bin. Here is an example (parted.7.Q) of their use:

```plaintext
with_format _table_
do Display
each[ .info_src, .bin, .source, .bin_siz ]
each_time(
  there_exists [ .tmp_bin_siz, .tmp_bin, .tmp_source ] such_that(
    there_is_a_bin_for PARTED
      with_bin_vbl tmp_bin
      with_source_vbl tmp_source
      with_byte_size_vbl tmp_bin_siz
    where(
      Group_Nbr Is_In [ 5, 2, 4, 3, 1 ]
      and Info_Source = .info_src )
    and .bin = .tmp_bin
    and .source = .tmp_source
    and .bin_siz = .tmp_bin_siz
  ));
```

Note that *there is a bin for* also works for non-horizontally partitioned record classes, although obviously it cannot provide information on any partitioning attributes because there aren’t any.

### 23.1.6 rcd.HPARTI_BIN

Likewise, Daytona provides a way to run Cymbal queries over the FILE_INFO FILE itself, which not coincidentally is in the Daytona flat file format. In $DS_DIR/aar.sys, rcd.HPARTI_BIN describes the generic FILE_INFO_FILE:
Note that the Source FIELD is defined to be SAFE_STR so as to support pipe queries using unconstrained syntax.

Once again, the user's FILE_INFO_FILE has the format of a Daytona table but for static hparti, there is no accompanying rcd to use to run queries against it. However, there is a way to use this generic rcd.HPARTI_BIN for this purpose. To run this query (show_ff.JS):

```
select * from HPARTI_BIN;
```

against the FILE_INFO_FILE ordera_fl for RECORD_CLASS ORDERA in the test suite, proceed as follows:
cd $ORDERS_HOME
DS_PROJ="" DS_APPS=sys DS_PATH=$DS_PATH:$DS_DIR \ 
DS_SFLAGS="-sql_out_fmt _table_" \ 
DS Compile Q/show_fif.IS &&
rm -f my_file_info_file &&
l ln -s ordera_fls my_file_info_file &&
./R

This assumes the execution takes place in the same directory as ordera_fls; otherwise in general, $HPARTI_BIN_DIR may be set as needed. The result of running this query is to produce a table containing the contents of ordera_fls; obviously, other types of queries can be run on FILE_INFO_FILEs as well.

Of course, if using Cymbal, then using_source can be used with the there_is_a to select an appropriate FILE_INFO_FILE at run-time instead of doing the symbolic linking (hparti_bin.1.IQ):

```
with_format _table_
do Display each[ .file, .source, .indices_source, .hparti_key ]
each_time(
    there_is_a HPARTI_BIN
    using_source "cat $ORDERS_HOME/ordera_fls"
where(
    File = .file and
    Source = .source and
    Indices_Source = .indices_source and
    Hparti_Key = .tmp_hparti_key )
and .hparti_key = "[" +sum( over .a
    each_time( .a Is_In .tmp_hparti_key ) with_sep ":" )+ "]"
);
```

```
DS_PROJ="" DS_APPS=sys DS_PATH=$DS_PATH:$DS_DIR \ 
DS Compile Q/hparti_bin.1.IQ &&
```

`./R`

Fortunately, when using dynamic hparti (described in the next section), Daytona automatically creates a custom rcd.<recls> HPARTI_BIN for these purposes and others so that there is no need to maneuver a generic rcd.HPARTI_BIN into service.

Actually, the schema for HPARTI_BIN is the original or Version 1 form. Subsequently, Version 2 was introduced for the sole purpose of supporting the specification of record-level compression by means of a Rec_Map_Spec_File FIELD. Its rcd is found in $DS_DIR/aar.sys2 (not aar.sys):
Note the inclusion of the `Rec_Map_Spec_File` which is where, optionally, the path to a compression dictionary is placed. Version 1 is the default for static horizontal partitioning; in order to use Version 2 for static horizontal partitioning, put a `<Version, 2>` note on the corresponding `FILE_INFO_FILE` node in the rcd.

Actually, there is yet another version for `HPARTI_BIN` rcds. Dynamic hparti has proved so popular that users found themselves wanting to add their own information to the information that
Daytona is storing on each BIN. To that end, Version 3 builds on Version 2 by adding two FIELDs for the user to use for their own purposes: User_Data_1 and User_Data_2. These are two LIST[ (0->) STR ] FIELDs. Essentially, they can be used to store an arbitrary amount of data for two reasons. The first is that each LIST can have an arbitrarily large number of elements. The second reason is that the user can use the trick whereby the first LIST contains the names of user-provided sub-FIELDs whose values appear at the corresponding position in the second list. Its rcd is found in $DS_DIR/aar.sys3 (not aar.sys):

```plaintext
#{ RECORD_CLASS ( HPARTI_BIN )
  <Multiplicity, 0->>
  <Comment, extends Version 2 HPARTI_BIN \n   by adding User_Data_1 and User_Data_2>
  <__Last_Validated_For_Rcd_UTC, 2004-07-18@19:40:03>
#{ BINS
  #{ FILE ( pipe_fif )
    <Source, cat ${HPARTI_BIN_DIR:-.}/my_file_info_file>
    <Comment, the pipe_ form supports querying user fif>
    <Version, 3>
    <Data_Interface, DC>
    <Unit_Seps, ||>
    <Tuple_Delims, "[]"}
    <Comment_Beg, %} #
  }#
  #{ KEYS
    <Indices_Banned, yes>
    <Siz_Access, no> }#
  #{ FIELDS
    #{ FIELD ( File )
      <Type, STR>
      <__Position, 1> }#
    #{ FIELD ( Source )
      <Type, (0->1) SAFE_STR>
      <Default_Value, "">
      <__Position, 2> }#
    #{ FIELD ( Indices_Source )
      <Type, (0->1) STR>
      <Default_Value, "">
      <__Position, 3> }#
    #{ FIELD ( Rec_Map_Spec_File )
      <Type, (0->1) STR>
      <Default_Value, "">
      <__Position, 4> }#
    #{ FIELD ( Hparti_Key )
      <Type, LIST[ (1->) STR ]>
```
Version 3 is not the default for either static or dynamic hparti. In order to use Version 3, put a `<Version, 3>` note on the corresponding FILE_INFO_FILE node in the rcd. The system-generated rcd.DYNTUPLEX_V3_HPARTI_BIN in the test suite shows what a Version 3 HPARTI_BIN rcd looks like. hparti.v3.1.IQU illustrates some use of Version 3.

### 23.2 Dynamic Horizontal Partitioning

The horizontal partitioning discussed so far is static in nature because the way that Daytona implements the user’s hparti specifications is to compile all the information about the BINS into the query executable. Consequently, that information is static in that whenever the executable is invoked, that constant information and only that information about the BINS is used to compute the answers to the query. This arrangement has a few drawbacks.

First, in the very common situation where a moving window of data is being stored, clearly, when using static hparti, it would be pretty inconvenient to store the actual names of the data files in the data dictionary because when periodically old ones are removed and new data files added, any executables with the previous period’s configuration would become incorrect and would have to be recompiled to keep current. What is typically done in that situation to compensate is to store constant surrogates (e.g., TIME slots like `^45d13h`TIME) for the data file names in the data dictionary and outside of Daytona, arrange for the periodic redefinition of which surrogate name refers to which current data file, often by carefully manipulating UNIX filesystem symbolic links. In the case of using TIME surrogates, they would be arranged in a modulo ring where, in the steady state, when a new data file is ready for loading, the TIME slot for the oldest is reused (after the oldest data file is removed). So, this indirection is inconvenient as it requires some careful, often intricate, programming to maintain the association between actual DATE_CLOCKs and TIME slots. Worse, end-user queries often have to refer explicitly to a function that performs this mapping.

Second, suppose one wanted to store a table implemented using many thousands or millions of files (which may also be coming and going daily). Any corresponding executable might be a challenge for the C compiler to make due to array size limitations and would be pretty large to boot. And it might be unnecessary to create such a monster in common cases because it could well be infeasible or at least infrequent to want to run a query that visited every single data file anyway -- so, why pay for storing information on all the files when only a subset will be visited?

Dynamic horizontal partitioning is a feature that Daytona offers to eliminate these inconveniences and offer other flexibilities. By means of arranging for each invocation of a query executable to construct for itself at runtime a data structure (an array of C structs) describing which (if not all) of the
contemporary BINS that the query needs to work with, dynamic hparti enables the user to avoid
indirection and refer directly to the the data files by their true names (no surrogates) in the data
dictionary. This also speeds C compilation of the fio files and reduces executable size, which is a good
thing if there are hundreds of executables. Of course, this does mean that there will now be some time
spent at execution time to create the in-memory data structures describing the BINS that the query
needs to use.

In order to condition an hparti rcd to be dynamic hparti, one must:

• attach a <Dynamic, yes> note to the BINS description in the rcd

• ensure that the following notes are present in each FILE_INFO_FILE description in the rcd:
  <Version, 2>, <Unit_Seps, ||>, <Comment_Beg, %>, <Tuple_Delims, ^[\]]^>.

• in the likely event that there will be a large number of BINs, use Max_Open_Bins,
  Max_Open_Seq_Acc_Bins, Max_Open_RANDOM_Acc_Bins notes on the BINS description to avoid
  the "too many open files" fatal error (or the corresponding error of too many zombie/defunct
  child processes when using pipe Files).

Optionally, a Using_Source note can be attached to the BINS description to enable additional
flexibility (cf., rcd.DYNORD and rcd.DYNORD2, resp.).

It is important to note that dynamic hparti is just an option to implement horizontal partitioning
differently from the static version; it remains the same basic concept; it does not change how it is used
by the query writer, except that instead of using surrogate concepts for hparti FIELDS, FIELDS can be
used that are directly meaningful to the application such as DATE_COCKs.

Here is an example of how querying becomes simpler and more natural than by using INT
surrogates. Suppose that the data is being stored by day, i.e., the DATE_CLOCK timestamp for each
record is such that all the records for any given day are completely contained in one BIN whose hparti
FIELD Date_Placed (say) has the DATE value that agrees with the values of a non-hparti
DATE_CLOCK timestamp FIELD for the records in that BIN. (For example, a record with a
DATE_CLOCK timestamp FIELD value of ^2011-07-07@2PM would go in the ^2011-07-07 BIN.)
Then all of the following DSQL queries will automatically select for processing only those BINs whose
Date_Placed value is in the indicated ranges (cf., netfio2.1.S):
Suppose in another situation, one wanted to store the records in BINs according to smaller time intervals, like for every minute. Then make the hparti FIELD a DATE_CLOCK whose values are the 1 minute interval start times like "2006-06-06@8:05am". Suppose this FIELD were called Starting_DC. Then to ensure that only the right BINs get opened, the user just needs to express appropriate constraints on Starting_DC, as illustrated by (netflo4.1.Q):

```sql
select count(*) from NETFLOWED4
where Starting_DC between "2006-06-06@8:05am"DC and "2006-06-06@8:07am"DC ;
select count(*) from NETFLOWED4
where Starting_DC > "2006-06-06@8:06am"DC // same as >= "2006-06-06@8:07am"DC
```

Furthermore, it is possible to use variable length time slots, as would occur when some BINs are being used to store real-time data that is being delivered and stored with each minute’s worth of data stored in its own BIN whereas the older, now historical, data is being (re-)stored in BINs by the hour, all in the same table. This is actually the case for NETFLOWED4 where the hparti FIELD is a DATE_CLOCK named Starting_DC indicating (as above) the starting time point for the BIN. Then Starting_DC can be used as before. Of course, one must continue to be aware of what the Starting_DC are so as to choose inequality bounds that capture the desired BINs. See netflo4.1.Q for an example of this.

23.2.1 Configuring And Using Dynamic Horizontal Partitioning

The implementation of dynamic hparti is based on the idea of treating one or more FILE_INFO_FILEs as regular Daytona data files. The idea is that the DBA will use Daytona transactions exclusively to add, delete, and update these FILE_INFO_FILEs (which themselves are describing the actual data files that store the data records). Then whenever a query starts up, it executes (behind the scenes) a read-only transaction to pull out just the information it needs from the
current state of the FILE_INFO_FILES. It should be clear that the reason why only Daytona update
mechanisms should be used to modify the FILE_INFO_FILES is because the queries (constantly) need to
get at this information and so cannot tolerate colliding with updaters. A fortiori, no-one should use a
text editor to modify a FILE_INFO_FILE when dynamic hparti is being used!

With dynamic hparti, since the DBA (not the end-user) will now be using Daytona to manage the
FILE_INFO_FILE information, they will need to have an rcd that supports that "metadata" activity.
Daytona will automatically process the user-created rcd.recls to create such an rcd by creating an
rcd.recls_HPARTI_BIN (as in rcd.DYNORD_HPARTI_BIN), which looks like this:

```plaintext
#{ RECORD_CLASS ( DYNORD_HPARTI_BIN )
  # BINS
   <Max_Open_Bins, _all_>
   <Max_Open_Seq_Acc_Bins, _all_>
   <Max_Open_Random_Acc_Bins, _all_>
  #{ FILE ( dynord_fl1.1 )
      <Source, $ORDERS_HOME>
      <Unit_Seps, |>
      <Comment_Beg, %>
      <Tuple_Delims, ^[]^
      <Version, 2>
      #{ PARTITIONING_FIELDS
        #{ FIELD ( Number ) <Value, 1> }
        #{ FIELD ( Fif_File ) <Value, $ORDERS_HOME/dynord_fl1.1> }
      }#{
  }
  }
  #{ FILE ( dynord_fl2.2 )
      <Source, $ORDERS_HOME>
      <Unit_Seps, |>
      <Comment_Beg, %>
      <Tuple_Delims, ^[]^
      <Version, 2>
      #{ PARTITIONING_FIELDS
        #{ FIELD ( Number ) <Value, 2> }
        #{ FIELD ( Fif_File ) <Value, $ORDERS_HOME/dynord_fl2.2> }
      }#{
  }
  }
  #{ FILE ( dynord_fl3.3 )
      <Source, $ORDERS_HOME>
      <Default_Data_File_Source, $(ORDERS_DATA:-$ORDERS_HOME)>
      <Default_Data_File_Indices_Source, ${INDICES_HOME:-$ORDERS_HOME/indices_dir}>
      <Unit_Seps, |>
      <Comment_Beg, %>
      <Tuple_Delims, ^[]^
```
Each record of the `recls_HPARTI_BIN RECORD_CLASS` describes one FILE that itself contains `recls` records. This rcd.`recls_HPARTI_BIN` is placed after its corresponding rcd.`recls` in the (same) aar. Note the metadata character of this rcd in that the contents of the FILE `dynord_fls.2` are being described by this `HPARTI_BIN` rcd, not the contents of any data file. (By the way, it is perfectly OK to have just one `FILE_INFO_FILE` for a `RECORD_CLASS`; it is just that the more general case is being illustrated here.) And as a FILE, each of its records have FIELDS like Source and Rec_Map_Spec_File that are describing a particular data file (named by the FIELD File).

Clearly, this rcd.`recls_HPARTI_BIN` is a more detailed and complete description of the `recls` metadata than is provided by the generic rcd.`HPARTI_BIN` described previously. Indeed, one significant improvement is that the Hparti_Key FIELD is an appropriately typed TUPLE in the case of rcd.`recls_HPARTI_BIN` whereas it is just a LIST[STR] in the generic case.

In fact, note the presence of Rec_Map_Spec_File. There was no such FIELD in the
FILE_INFO_FILEs for static horizontal partitioning as described above. The format of the static (original) FILE_INFO_FILEs is said to be Version 1; FILE_INFO_FILE records that have been extended to include Rec_Map_Spec_File are said to be Version 2. Dynamic hparti will only work with Version 2 FILE_INFO_FILEs; clearly, this Version supports introducing new compression dictionaries on a (new) data file by data file basis as time goes by, which is very valuable when storing, say, a 10 year moving window and wanting to introduce new compression dictionaries 5 years in to handle changes in the nature of the data as time goes by and not be restricted to just one dictionary for the whole lot. Also, it supports having some data files compressed and others not. On the other hand, the FILE_INFO_FILEs for static hparti can be either Version 1 or Version 2, with Version 1 the default (as happens when no Version note is attached to the FILE_INFO_FILE description). For dynamic hparti, a <Version, 2> note must appear explicitly in each FILE_INFO_FILE description.

The Partitioning FIELDS Number and Fif_File are for the convenience of the implementation and are what support having multiple FILE_INFO_FILEs for the original data RECORD_CLASS BINS.

Most importantly, the last FIELD Hparti_Key for a recs_HPARTI_BIN record is a TUPLE which carries the hparti FIELD values for the corresponding data file. The DBA will be referring to the values of Hparti_Key FIELDS in order to specify the deletion and/or addition of new data files to the original RECORD_CLASS. The implication then is that the DBA will be writing queries/updates against recs_HPARTI_BIN.

FYI, in contrast to static partitioning, there is no need to sort the records in the FILE_INFO_FILEs according to Hparti_Key values (because the system does that when they are read in at query runtime).

It’s important to know when creating the contents of any FILE_INFO_FILE that all hparti FIELD values should be written as they would appear when written out using Write() at the plaintext user-level; in other words, they should not be encoded as they might appear when stored in a DC data file. For example, an Hparti_Key FIELD IP address component should have the same form as 192.168.1.1 and not appear in the special cryptic HEKA-like encoding that it does in DC data files. This is contrary to what one would otherwise expect but the reason is that these values are just like what would otherwise appear naturally in the rcd for which the FILE_INFO_FILE is a partial substitute (and of course, it is more pleasant to read and write them in human-readable form).

Remember rcd.recls_HPARTI_BIN is generated automatically by Daytona; the user should refrain from making any modifications to it because any user modifications will either be ignored or else cause failures. The user may notice that this rcd.recls_HPARTI_BIN, which they should not modify, does not contain any reference to non-trivial default values for Source, Indices_Source, or Rec_Map_Spec_File. That’s fine, any notes for Default_Data_File_Source, Default_Data_File_Indices_Source, or Default_Rec_Map_Spec_File that appear in the base rcd.recls even on a FILE_INFO_FILE by FILE_INFO_FILE basis are correctly handled, but by a different mechanism. In fact, they are handled by the Cymbal code automatically generated by Daytona in recs_fio_pkg.cy; there is no need whatsoever for the user to examine the contents of this Cymbal package. It is being mentioned here just to forestall the user removing it and/or its associated archive for whatever reason.

There are occasions when it is required that the recs_HPARTI_BIN be implemented with <Indices_Banned, yes>. This can be accomplished by introducing a <Indices_Banned_For_Fif, yes> note on the BINS description in rcd.recls. This will cause the .siz and free tree for recs_HPARTI_BIN not to be built and consequently, no reuse of freed record slots will occur and furthermore, no Cymbal txn
updating will be allowed because that requires a .siz file.

When dynamic hparti is specified, for any and all FILE_INFO_FILES, the rcd must specify:

```plaintext
[Unit_Seps,  ||]
<Comment_Beg, %>
<Tuple_Delims, ^[\]]^>
```

Remember this is for the FILE_INFO_FILE, not the data files. However whatever any data file uses for these delimiters must be used by all of them. This is not as restrictive as it may seem because the type of the Source FIELD in a FILE_INFO_FILE is SAFE_STR implying for example that | can be used within Source FIELD values (as would occur when pipe_Sources use |s). The FILE_INFO_FILES can have different Source notes. Empty lines in FILE_INFO_FILES are supported. Also, supported are single comments extending from the comment character to the end of the line: obviously, this supports running update transactions on these files. No other comments are supported, i.e., any attached to fields other than the last. Also, the data files must all have the same values for the following attributes, once any default values are taken into account: Reuse_Freed_Space, Read_Only, Concurrent_Access, Random_Acc_Bufsize, Seq_Acc_Bufsize, and Pad_To.Len.

One constraint to keep in mind is that no running program is going to be sensitive to any updates made to the FILE_INFO_FILES for its dynamic hparti RECORD_CLASSES by other programs while it is executing. That means it cannot see them or know them or respond to them. This is because when it starts, it reads in the current state of the FILE_INFO_FILES (as needed) in order to determine what and where its data files are -- and then for the duration of its run, that information remains static and unchanged. The one limited exception to the last statement is described in the next sub-section.

Ordinarily, unless instructed otherwise, query executables will call Sizup to assess if the indices are up-to-date with regards to the data. However, for dynamic hparti, this is not done for the data files. The reason is that the data files are probably too numerous to tolerate a longish Sizup execution, even to verify that their indices appear to be in good order. On the other hand, assuming no +trustme is in effect, Daytona executables will call Sizup to ensure that any dynamic FILE_INFO_FILES have their indices in good order.

In summary, the recommendation is that dynamic horizontal partitioning be used if a FILE_INFO_FILE is being used, as would probably be the case if there were a lot of data files; otherwise, if the RECORD_CLASS metadata configuration doesn’t change often and there are not an unwieldy number of data files under BINS and either there are no surrogate hparti FIELDSs or no-one is complaining about their complexity, then static horizontal partitioning is simpler and perfectly adequate.

### 23.2.2 How To Add And Sizup A New Hparti Bin Without Interfering With Queries

Since the main reason for dynamic hparti is allow the system to add and delete new BINS while reading and updating queries are running at the same time, the question arises how to add a new BIN, including running Sizup on it, without disturbing readers and writers on the table. While one always has the option of requiring the users of the RECORD_CLASS to use with_dirty_reads_ok and ignoring_failed_opens, here is a different strategy.
1. For a new data file, add a temporary/fake entry to the corresponding HPARTI BIN. This is accomplished by choosing nonsense values that no query would want to work with for the Hparti Key. Here is an example:

local: TUPLE[ INT(_long_), STR(*), INT(_long_) ] .hparti0_key = [ -1, "none", -1 ]
insert into DYNORD_HPARTI_BIN
("Number", "Pif_File", "File", "Source", "Indices_Source", "Hparti_Key")
values (1, "${ORDERS_DATA:-$ORDERS_HOME}/dynord_flsl.1",
"ORDER.12", "SAFE_STR", ",", .hparti0_key);

2. Then run Sizup on the fake entry, as is done with this command:

    Sizup -m -FYI -fl @DYNORD=ORDER.12 -app misc

3. Then update the fake HPARTI BIN entry to be correct. Here is an example:

    local: TUPLE[ INT(_long_), STR(*), INT(_long_) ] .hparti0_key = [ 4, "A", 1 ]
    update DYNORD_HPARTI_BIN
    set Hparti_Key = .hparti0_key
    where "File" = "ORDER.12"

The result is a correctly configured and Sizup’d BIN ready for other queries to use. Note that Daytona uses locking to keep all Cymbal/DSQL accesses the the HPARTI_BIN files from interfering with each other.

23.2.3 Enabling A Process To Create And Work With A New HPARTI BIN

There are occasions when a process is ingesting new records into a dynamic hparti RECORD_CLASS that an otherwise valid record is encountered that does not belong to any existing HPARTI_BIN. Ideally, the process would be able to create the appropriate new HPARTI_BIN and then continue on to add the new record to it and for that matter, to continue on reading/writing to all of that RECORD_CLASS BINs. Certainly, Daytona supports running a transaction that will create a new HPARTI_BIN record describing the HPARTI_BIN and Daytona will also support the process calling Sizup to create and initialize the HPARTI_BIN itself. However, without doing something different, the process will not be able to add that record to the new HPARTI_BIN because the process itself has no usable record that the bin exists. The reason is that a process will read in all HPARTI_BIN information for a given table when it first needs any of it and then never update that information (which is kept in memory). So, in order for this process to add that new record to a new HPARTI_BIN, some means has to be provided to cause it to update its own understanding of what HPARTI BINs are out there on disk. That is the function of the optional there isa new keyword updating bin stats. Here is an example of its use taken from the test query hparti.addnewbin.1.IQU; in a separate transaction to add new HPARTI_BINs, one finds:
do Change so_that(
    there_isa_new DYNORD_HPARTI_BIN updating_bin_stats where(
        Number = 3 and
        Fif_File = "${ORDERS_DATA:-$ORDERS_HOME}/dynord_fl3.3" and
        File = "ORDER.N2" and
        Source = "${ORDERS_DATA}SAFE_STR(_plain_) and
        Indices_Source = "${ORDERS_DATA}" and
        Rec_Map_Spec_File = "" and
        Hparti_Key = [ 101, "A", 2 ]
    )
);

This will not only add a new record to DYNORD_HPARTI_BIN, which is a table on disk, but it will also update an internal program memory structure called DYNORD_Bin_Stats; it is this latter structure that contains the process’s understanding of the BINS implementing the DYNORD table. Once DYNORD_Bin_Stats is updated with the new information, then the process will be able to find the BIN information it needs in order to add the new record (in a separate transaction (in this same process)!)

Rhetorically, how many such new HPARTI_BINs can be added by a single process? The answer is at least 10. It can be more than 10 if a Can_Grow_Stats_By_Pct note is attached to the (first) FILE_INFO_FILE node in the rcd for the base RECORD_CLASS (i.e., the one being dynamically partitioned, in this case, DYNORD). For example, <Can_Grow_Stats_By_Pct, 10> means that .1 * initial_bin_count new HPARTI_BINs can be added by the current process.

There are several constraints on using updating_bin_stats however:

- updating_bin_stats must be used with a there_isa_new, i.e., an unconditional add.
- In the associated do Change, explicit values must be provided for all HPARTI_BIN attributes.
- Some use of the base table must occur before trying to add a new HPARTI_BIN (so as to cause Daytona to read in information to create and populate the Bin_Stats).
- The new HPARTI_BIN must have Hparti_Key field values that put it at the end of the lexicographic Cymbal ordering of the Hparti_Key field values. For example, if the single Hparti_Key FIELD is a DATE_CLOCK, then the Hparti_Key FIELD value for the new HPARTI_BIN must be a DATE_CLOCK greater than all existing values. If this is not possible in general, then it can be artificially made possible by introducing a new first Hparti_Key FIELD which is some kind of sequence number.
- This is all about adding new HPARTI_BINs; it offers nothing to support deleting nor updating their information in the Bin_Stats.

Once again, hparti.addnewbin.1.IQU shows this feature in action.

If these constraints are prohibitive, then a less simple solution is the following. Have a parent process able to install a new BIN; have child processes that do all of the loading; when a child discovers that a new BIN is needed, then it sends a message back to its parent (on their shared pipe) with the particulars and it exits; the parent then builds the new BIN and invokes another child process to continue loading (by using that new BIN when needed).
23.2.4 How To Handle Millions Of Dynamic Hparti BINs

Dynamic horizontal partitioning is intended to scale indefinitely to handle millions of data files. Of course, at some point in this scenario, any given query in practice is simply not going to visit all of the data files that exist in any feasible time period. Furthermore, for the invocation of each query, there is a setup cost incurred to read in and process the FILEINFO_FILE information, all of which, when read in, has to fit comfortably in memory. So, Daytona offers a way for the user to control which data files a query invocation is even allowed to work with.

By attaching a Using_Source note to the same BINS description in the rcd that <Dynamic, yes> has been attached, the user provides Daytona with an invocation of another query executable that will produce the desired subset of FILEINFO_FILE information. Here is an example of such a note (rcd.DYNORD2):

<Using_Source, $ORDERS_HOME/dynord2_bin_select.X +T ${START_REGION:-1} ${END_REGION:-15}

This note tells Daytona that in order to produce rcd information on the data files, i.e., in order to produce the information carried in the FILEINFO_FILE(S), it should execute the given command. This particular command passes a couple runtime arguments to this simple Cymbal program (dynord2_bin_select.Q):

```
set [.start_region, .end_region ] = read( from _cmd_line_ bia[ 1, 15 ] );
do Filter_It( .start_region, .end_region );
global_defs:
define PROC( INT .start_region, INT .end_region ) txn task: Filter_It
{
    fet ( there_is_a DYNORD2_HPARTI_BIN
           where( Hparti_Key = .hparti_key )
           and .hparti_key#1 >= .start_region & <= .end_region
    ){
        with_format _data_ where( this_isa DYNORD2_HPARTI_BIN ) do Describe;
    }
}
```

By default, this will produce a listing of the entire contents of the FILEINFO_FILES. However, by appropriate setting of the shell environment, it can also produce whatever subset is desired at the moment of executing the query. Manipulating the shell environment is just one way to influence the execution of the query: any way consistent with executing an arbitrary command at the shell level is fine. Note that the Cymbal FUN put_shell_env enables a running Cymbal program to change its shell environment. And of course, if this flexibility is not desired, Using_Source does not have to be used at all.

The DBA may notice that Daytona creates an rcd.recls_HPARTI_BIN_PIPE; the system uses this to handle the Using_Source option. No-one should modify this rcd.
23.3 Directory-based Horizontal Partitioning

There are times when it is inconvenient to use static horizontal partitioning whereby the BINS are described explicitly in the rcd or else in the FILE_INFO_FILE. In those latter two scenarios, the number of BINS and the filesystem location of each BIN have to be known at query compile time. There are occasions where (more) runtime flexibility is required in that the location and number of BINS is constantly changing and so the first thing a query executable would like to do is to determine a match between what BINS are currently available and which BINS the user would like to query in the current invocation of the query executable. Then just those consensus BINS would be queried.

These needs can be met by using directory-based horizontal partitioning. The essence of this approach is for the application to encode the horizontal partitioning attributes into the UNIX filesystem paths to the data files and then to access the files by means of the using_source keyword. The implication is that the Daytona database management system is no longer actually aware of the horizontal partitioning per se, as seen, for example, by the fact that the horizontal partitioning FIELDS no longer appear in the rcd (and thus cannot be used in SQL SELECTs or Cymbal there_isas): instead it is the application itself that needs to explicitly take care of how to compute the horizontal partitioning FIELD values (from the file paths) as well as keeping track of where the BINS are in the filesystem.

There are two basic kinds of directory-based horizontal partitioning, one where the data is coming from files in the filesystem, which may or may not be indexed, and the other where the data is coming from pipes. In the former case, the BINS node (for the ORDIRECT record class) would look something like this:

```bash
#{ BINS
  #{ FILE ( ORDER )
    <Source, $ORDERS_HOME/Ordirect/1/A>
    <Unit_Sep, :>
    <Comment_Beg, #>
    <__Max_Rec_Len, 77>
    <__Nbr_Of_Recs, 1000>
    <__Size, 34675>
    <__Min_Rec_Len, 20> }
}
```

Of course, the rcd would contain a FIELDS description as well as perhaps a KEYS description. There would be nothing in this rcd to suggest that it was associated with horizontal partitioning; for example, there would be no PARTITIONING_FIELDS. The understanding here is that every directory-based FILE worked with will be named ORDER and will be located at run-time using a using_source directory path. (So, for the purposes of querying, the value and even the existence of the Source note in the rcd is irrelevant since the location of the ORDER file will be determined by the query at run-time.) Such queries are illustrated by (dirparti.1.Q):
set .path_box = [ .path-"/ORDER" :
    [.path] = ptokens( upto "\n"
        for "ls $ORDERS_HOME/Ordirect/*//*/ORDER" ) ];

with_format _table_
in_lexico_order
with_col_labels [ "Region", "Category", "Count",
    "Min_Date_Placed", "Total_Quantity" ]
do Display
each_tuple_of
    { [.region, .category, count(), min( over .dp ), sum( over .qty ) ]:
        .path Is_In .path_box and
        [.region, .category ] = stokens( for .path matching ".*/\([^/]+\)/\([^/]+\)\$"
            and
            there_is_an ORDIRECT using_source .path
            where( Date_Placed = .dp and Quantity = .qty )
    };

What we see here is that there are two variables region and category that take the role of being
horizontal partitioning attributes and have their values computed by doing pattern matching on the file
paths that are the values of the using_source keyword. For this example taken from the test suite, here
are the file paths down to the BIN, relative to $ORDERS_HOME:

    Ordirect/1/A/ORDER
    Ordirect/1/B/ORDER
    Ordirect/1/C/ORDER
    Ordirect/2/A/ORDER
    Ordirect/3/A/ORDER
    Ordirect/3/B/ORDER

The values of region come from the single digits and the values for category come from the single letters.
In this scenario, to get any indices built, Sizup must be called deliberately on each FILE by the likes of:

    Sizup -FYI -app misc -rec ORDIRECT -source $ORDERS_HOME/Ordirect/1/A

A <Trustme, yes> note on the KEYS description will prevent any query from automatically
attempting to run Sizup on the FILE specified for this record class in the rcd.

For the second scenario which is based on pipes, the rcd would have a BINS description
something like this used for ORDIRECT2:

Copyright © 2013 AT&T
All Rights Reserved.
September 15, 2013
This is a typical pipe_ FILE description. Then queries would look much the same as they did for the first scenario although not identical:

```plaintext
set .path_box = [ .path : [.path] = ptokens(  
    upto "\n" for "ls $ORDERS_HOME/Ordirect/*/ORDER" ) ];
with_format _table_  
in_lexico_order  
with_col_labels [ "Region", "Category", "Count",  
    "Min_Date_Placed", "Total_Quantity" ]
do Display  
each_tuple_of  
{ [.region, .category, count(), min( over .dp ), sum( over .qty ) ]:  
    .path Is_In .path_box and  
    [.region, .category ] = stokens( for .path  
        matching ".*/\([\^/]+\)/\([\^/]+\)/ORDER" )  
    and  
    there_is_an ORDIRECT2 using_source "cat " + .path  
    where( Date_Placed = .dp and Quantity = .qty ) );
```

Note that, as expected, using_source values in this scenario specify the various pipe commands needed to produce the data. The interesting thing about both of these queries is that Daytona will automatically close the previous channel and then open a new channel each time the value of using_source changes.

### 23.4 Targetable rcds

On occasion, the user has several tables that could be described by rcds that have the same FIELDS and KEYS and only differ in the location (and contents) of the actual data files. In this case, it would be a helpful simplification to describe all the tables with just one rcd that is targetable to each and any of the tables by the simple means of redefining the shell environment. To accomplish this, one or more of the following Daytona system parameters would need to expand differently according to the shell environment specific to the targeted table:

1. The DS_PATH environment variable.
2. The Source and Indices_Source notes for a FILE description under the BINS description.
3. The Source note for a FILE_INFO_FILE description under the BINS description.
4. The value of the Source and/or Indices_Source FIELDS in an actual FILE_INFO_FILE.
5. The Fio_Dir value under the RECORD_CLASS description.

There are of course default mechanisms for several of these concepts as illustrated by Default_Data_File_Source. The Fio_Dir note is an optional note that directs Daytona to place that RECORD_CLASS’s fio files in the UNIX filesystem directory specified by the note, which would probably be different from where they would go by default, that being the same directory as that for the aar describing the RECORD_CLASS. The Fio_Dir should be an absolute path; it is fine for it to be a shell expression. The fio files are the ones created by Daytona to support I/O out of that RECORD_CLASS’s data files; they are named with .[cho] extensions appended to the RECORD_CLASS name itself. In the targetable rcd scenario, each target table needs to have its own fio files and so clearly, to prevent confusion and wasteful recompilation, the Fio_Dir note allows each to be placed in its own special location. If there is no Fio_Dir note on the RECORD_CLASS description, then the system attempts to determine an Fio_Dir from a Default_Fio_Dir note on the corresponding APPLICATION and PROJECT description, if any, in that order.

So, to reuse a given rcd for a different targetable table, it is necessary to change the shell environment appropriately and then to run Sizup and/or compile-and-run queries in that environment.

23.5 Additional Kinds Of FILE Descriptions

23.5.1 stdin_FILE Descriptions

By using a stdin_FILE description, a Cymbal executable can read a stream of records piped to it from another process while still realizing the benefits of reading data by means of an rcd. Here is how a record class is defined as having its data file records coming from stdin.

```#
#{ RECORD_CLASS ( NUMBER )
  <Multiplicity, ^0->^>

  #{ BINS

    #{ FILE ( stdin_NUMBER )
      <Data_Interface, DC
      <Unit_Sep, ^\t^>
      <Comment_Beg, ^#^>
    }#

    . . .

  }#
#
```

All that is necessary is for the name of the FILE to begin with "stdin_"; in fact, "stdin_" alone is sufficient.
23.5.2 pipe_ and pipein_ FILE Descriptions

A pipe_ or equivalently and more specifically a pipein_ FILE description is used when a Cymbal program would like to use the RECORD_CLASS paradigm to read and process a stream of records being piped to it by a given shell command specified by a Source attribute value. Here is an example of such a FILE description:

```
#{ FILE ( pipe_SUPP_ )
  <Data_Interface, DC>
  <Source, cat $HOME/d/SUPP>
  <Unit_Sep, ^\t^>
  <Comment_Beg, ^#^>   }#
```

All that is necessary is for the name of the FILE to begin with "pipe_" or "pipein_"; in fact, "pipe_" or "pipein_" alone is sufficient. At runtime, a using_source keyword argument to an associated there_isa will result in that associated pipe definition being used instead of what the rcd specifies.

By explicitly using the --validate_only keyword to Sizup, the user can induce Sizup to perform all data validation tests on the data file records produced by a pipe. This can be very useful for debugging since Daytona assumes that the data in its RECORD_CLASSESes is valid and consequently, if not, Daytona will behave badly and even mysteriously so.

23.5.3 stdout_ and pipeout_ FILE Descriptions

stdin_ and pipein_ FILE descriptions are used to employ the RECORD_CLASS paradigm when reading data into a program. Their predictable opposites, stdout_ and pipeout_ FILE descriptions, enable the RECORD_CLASS paradigm to be used when writing data out of a Cymbal program. For example, with a solitary BIN like this:

```
#{ FILE ( pipeout_ )
  <Source, "cat - >> ${DATA_DIR:-.}/mydata"}>
  <Unit_Sep, |>
  <Comment_Beg, #>
```

a blind append like this in a loop:

```
do Change so_that(
  there_is_a_new ORDER
  using_no_index
  using_source "sort -k 1,1 | gzip >>${DATA_DIR}./cur_file/.gz"ISTR
  where( Number = .onbr and Supp_Nbr = .sno and Part_Nbr = .pno ... ))
```

will not only produce a stream of ORDER records but will also sort, compress and store them in a file whose name is based on the current value of the Cymbal variable cur_file. This use of blind appends has all the usual advantages including automatic conversion of FIELD values to their data file form and the automatic sorting of the FIELD values according to their order in the rcd.
23.5.4 fifo FILE Descriptions

The UNIX fifo or named pipe is a special file that acts like a pipe. The principal advantage that a fifo has over a pipe is that any two processes can communicate via a fifo whereas one must be a child of the other if they wish to communicate by means of pipes. This is accomplished simply by creating a fifo named by a file system path and opening that file for reading and writing. Incidentally, this implies that fifos can be used for inter-system communication by means of commands like rcp or scp. Multiple processes can read and write to fifos at the same time with the understanding that reads and writes will be atomic as long as they are less than at least 4096 bytes, possibly more on some platforms (Solaris pipes are 5120 bytes long). By default, an open fifo for read blocks for an open-for-writing to occur and conversely, thus providing a way to synchronize readers with writers.

Fifo files are specified in the rcd by having the file name begin with fifo as in the following FILE node from a horizontally partitioned record class:

```bash
#{ FILE ( fifo_SUED )
  <Source, $ORDERS_HOME>
  <Unit_Sep, \t>
  <Comment_Beg, #>
  <__Position, 1>
  <__Size, 4532>
  <__Max_Rec_Len, 58>
  <__Nbr_Of_Recs, 103>
  #{ PARTITIONING_FIELDS
    #{ FIELD ( Dummy_Date ) <Value, 8-28-78> }#
  }#
}
```

When Sizup is called to work on a fifo, it does nothing unless it is called with -create, at which point it will create the fifo if it doesn’t already exist. That’s really the only special thing that Daytona does in support of fifo FILE descriptions.

As illustrated by the following test query, the way to write to a fifo FILE is to use blind appends.
set .cten = new_tendril( executing {
    fet [.nbr, .nm, .city, .phone] ist(
        there_is_a SUED where( Dummy_Date = "1-1-84"
            and Number = .nbr
            and Name = .nm
            and City = .city
            and Telephone = .phone )
    }
    do Change so_that(
        there_is_a_new SUED using_no_index where(
            Dummy_Date = "8-28-78" // selects the fifo_BIN
            and Number = .nbr
            and Name = .nm
            and City = .city
            and Telephone = .phone )
    )
    )
    set .ii = 0;
    fet there_is_a SUED using_no_index where( Dummy_Date = "8-28-78" )
    do {
        set .ii++;
        with_format _data_ where this_isa SUED do Describe;
        when( .ii > 10 ) break;
    }
    _Wait_For_Tendrils

Also, it is important to use using_no_index when working with a fifo_FILE because, for example, they do not even have a .siz file. Obviously, in practice, one process would be writing to the fifo_ and another reading from it. The SUED RECORD_CLASS illustrates have one FILE be a fifo_ and another being a regular UNIX file.

23.5.5 Bufsize Notes For FILE Descriptions

Bufsize notes can be used to instruct Daytona to use buffers of specified size for random or sequential accesses to the files associated with particular record classes. This can serve to speed up access considerably. Indeed, these buffers can be specified to be large enough to contain the entire data file, if desired, thus improving efficiency by causing the entire data file to be read into memory. This is not the same as what occurs when boxes are used because in the box case, the information has already been parsed and converted into INTs or FLTs or whatever as needed and furthermore, the index to the box is in memory as well. This contrasts with reading the data file alone into memory buffers in its pristine UNIX flat file format.

Anyway, to specify the buffer size in bytes for a random access method (like a B-tree) to use for a data file, just place the likes of a <Random_Acc_Bufsize, 4096> note under the appropriate FILE description for that file in the rcd. For sequential access, use the likes of a <Seq_Acc_Bufsize, 1048576> note. Buffers of size modulo 1024 are enforced.
As for suggested Random Acc_Bufsize values, 4096, the size of memory page, or 8192, a common size for a disk block, is probably best to use for big data files: the idea is that for big files, when an index points Daytona to a particular data file offset, it is unlikely that there will soon be another access to records within 4K-8K of that offset: hence, the unit read in is of no value for caching and should therefore be as small as possible but probably not smaller than the various units of transfer and storage. Some file systems support direct I/O (i.e., the disabling of the caching of file system blocks in virtual memory); in such a case, there is never a chance for any caching and therefore, all the more reason to keep Random_Acc_Bufsize values small.

On the other hand, the Seq Acc_Bufsize value, by its very nature, can afford to be large since every record that is read in will be used by the query. When striping data across multiple disks, it is recommended that Seq_Bufsize be set to be twice the stripe width. So, when striping in 64K chunks across 8 disks, Seq_Bufsize should be set to 1048576. The principle is that when a read for Seq_Bufsize bytes comes down to the disks, it will cause them all to read in parallel and to be able to take advantage of scheduling a couple of requests against the disk controllers at the same time. Under the right circumstances, striping has been observed to increase the I/O rate over the default non-striping configuration from 5 MB/sec to 25 MB/sec. The Veritas file system and volume manager are highly recommended.

The sequential or random access data buffer sizes for all files in a given record class can be specified by placing Default_Seq_Acc_Bufsize or Default_Random_Acc_Bufsize notes under the appropriate BINS description for that record class. The sequential or random access data buffer sizes for all files in a given application can be specified by placing Default_Seq_Acc_Bufsize or Default_Random_Acc_Bufsize notes under the APPLICATION description contained in the apd.*whatever* member of aar.*whatever*. This member can be edited by using a DS Vi command like:

```
DS Vi apd.orders
```

In fact, both apd.orders, rcd.PARTED, and rcd.PERSONNE from the sample orders application contain examples of Seq_Acc_Bufsize/Random_Acc_Bufsize use.

### 23.5.6 Read Only Notes For FILE Descriptions

When a `<Read_Only, yes>` note is included in a FILE description, Daytona considers that FILE to not be accessible by the user for any reason that would lead to its modification. Clearly, this would include any kind of update transaction as well as Sizup being asked to do batch adds, packing, padding, creating, truncating, or deleting faults. In Sizup's case, it further causes Sizup to forbear from getting a file lock and/or lock file for the associated FILE; this does leave an associated Sizup invocation vulnerable to unexpected changes made by Sizups running with a different rcd that do modify the data file. However, the main utility of `<Read_Only, yes>` is that it enables the user to build indices for data they do not have the ability to change, perhaps because they are not even the owner. By specifying `<Read_Only, yes>` and a suitable Indices_Source note that points to a directory that they control, the user can build indices and use Daytona on data owned by someone else who has not even put the data in Daytona by creating an rcd for it, etc. A dynamic, non-rcd way of achieving this end is to specify `-ro_data` as an option to a Sizup invocation If a Read_Only rcd...
annotation is not present for a FILE description, then a `Default_Read_Only` annotation will be used instead as soon as it is found by looking at any ancestor FILEINFO_FILE, BINS, or APPLICATION description.

23.6 Notes For FIELD Descriptions

23.6.1 Filtered Fields Via Filter_Funs

There are occasions where the user would like to filter FIELD values on their way in and out of Daytona data files. For example, there may be a security requirement for data records to contain encrypted FIELD values. In this situation, the user will want to provide a function to decrypt the FIELD value just after Daytona reads it but before it has a chance to do anything with it. Conversely, there will be a need to encrypt FIELD values just before Daytona writes them out into data files.

To support these and similar needs, Daytona provides the `Input_Filter_Fun` and `Output_Filter_Fun` notes for FIELD descriptions in rcds. For example,

```c
#{ FIELD ( Weight )
  <Type, ( 0->1 ) FLT>
  <Input_Filter_Fun, my_mult_by_100>
  <Output_Filter_Fun, my_div_by_100>
  <__Position, 4> )#
```

The `Input_Filter_Fun` value is a C filter function that maps the sequence of characters representing the FIELD value in the DC data file to a value of the indicated Cymbal type for the FIELD (for the user’s Cymbal program to work with when reading data in). The `Output_Filter_Fun` is defined conversely and serves to translate Cymbal values for the FIELD to the forms they take in the data file (when doing updates).

In this case, `my_mult_by_100` is multiplying FIELD values by 100 before Daytona gets to do its normal processing on them. `my_mult_by_100` is defined in `usr.env.c` for the sample project so that it will automatically be included in executables that need it:

```c
double my_mult_by_100( str )
  const char *str;
{
  return( 100.0 * dbl_for_str( str ) );
}
```

These filter functions must either have a Cymbal FUN task (not a helper) definition somewhere or they must have a Cymbal import and a C definition somewhere. This can be accomplished using the methods described in Chapter 6 under "Global Environment And User C-Extensions". Also, a Cymbal package is a good way to define both kinds of Filter_Funs using Cymbal.

Important caveat: in either case, whether the Filter_Fun is defined in C or Cymbal, if its Cymbal
type uses STR/RE/LIT/THING or HEKA/HEKSTR parameters or return values, these must be declared/defined as STR(=), not STR or STR(*), latter pair being the same thing. As a special case, note that from the standpoint of the Filter_Funs, the FIELDS in the (disk) data file are of type STR(=). The implication in all cases is that the filter functions must be defined or imported as being C_external.

Caveat: If an Output_Filter_Fun is used, then an Input_Filter_Fun must be used. Furthermore, Daytona assumes that each string FIELD value in the data file is mapped to itself by the composition function Output_Filter_Fun o Input_Filter_Fun. If this is not the case, then the indexes will not be maintained properly. In the future, it will be possible for multiple values in the Input_Filter_Fun domain to map to the same range element but not yet. (This would be similar to the same DATE being represented by multiple strings using any of the several DATE formats.)

FIELDS with Input_Filter_Fun and Output_Filter_Fun may have missing values, just like other FIELDS. If one should have a Default_Value, then that Default_Value is expressed as a Cymbal constant and therefore, will not be transformed by Input_Filter_Fun.

B-tree indexes cannot be built on FIELDS employing Filter_Funs unless the DC form of the FIELD data is ASCII. For that matter, in general, putting non-ASCII characters in DC data files is asking for trouble -- to avoid doing so, recall that BASE64 types support storing any kind of bytes in DC data files.

There is no need to malloc the result of a string-valued Input_Filter_Fun or Output_Filter_Fun since Daytona copies it to other space before working with any other FIELD. In general, any malloc’ing that may go on in Input_Filter_Fun or Output_Filter_Fun takes place outside of Daytona’s garbage collection system: it is the user’s responsibility to free such malloc’d space when it is no longer needed.

In the special case of a LIST/SET-valued FIELD, any Filter_Fun is for the scalar elements of the FIELD value, not for the FIELD value itself, which of course is a LIST or SET.

Caveat: Sizup does not do any typechecking or validation of the contents of FIELDS that use an Input_Filter_Fun. It is the user’s responsibility to ensure that Input_Filter_Fun returns without error (or with suitable error messages) and returns a value meaningful to the application. This is because architecturally, Daytona considers the data file representation of these kinds of FIELDS to be terra incognita: it has no way of knowing or understanding what goes on there: the management of that representation is solely the responsibility of the user.

Not infrequently, the user wishes to store the possibly lengthy values of an enumeration as short INT (or HEKA) codes in DC data file FIELDS -- and to use Input/Output Filter_Funs to transparently translate back and forth between the short/economical data file record representation and the Cymbal-level FIELD representation. Furthermore, whenever a new member of the enumeration appears, the system should automatically give it the next highest INT (or HEKA) code. And the mapping between codes and enumeration values should be stored in a table which can be read and updated by multiple processes concurrently. And all this logic is to be encapsulated in a Cymbal package. Can this be done? Yes: see that portion of $DS_DIR/EXAMPLES/usr/orders/liborders.cy package defining supplier_for_int_str() and int_str_for_supplier() as implemented using the SUPP_MAP table.

23.6.2 Practically Unbounded KEYS Via Message_Digest

By default, for technical implementation reasons, Daytona key values cannot exceed about 120
bytes in length. Now while short keys are good for minimizing storage space and execution times, there are occasions when working with key values longer than 120 bytes is worth the extra cost. Fortunately, this is now possible by using a Message_Digest note attached to designated INDEX descriptions in rcds. The result of doing that is that the only thing remaining that constrains the length of key values is the maximum record length constraint for records containing those key values.

Here is an example of a Message_Digest rcd annotation:

```plaintext
#{ KEY ( md )
   <Fields, [ Number, Supp_Nbr, Part_Nbr, Date_Recd, Date_Placed ]>
   #{ INDICES
      #{ INDEX ( md )
         <Kind, btree>
         <Message_Digest, md5>
      )
   )
}
```

There are only two values allowed for Message_Digest: md5 and sha1. These are names for two message digest algorithms. These message digest algorithms take strings of arbitrary length and map them to ASCII hash strings of length 32 and 40, respectively. It is these ASCII hash strings that are put into Daytona’s B-trees to support indexed retrieval. Both md5 and sha1 Message_DIGESTs work correctly for Daytona. In the literature, sha1 is the better message digest algorithm but at this time, there is no other information that would recommend one over the other for Daytona purposes -- except that md5 clearly produces shorter hash values. Note that regardless of the choice, any key-value input to md5/sha1 that is shorter than 32/40, respectively, is not processed into a hash value. Note also that this Message_DIGEST feature works for KEYS that may or may not use SET-/LIST-valued FIELDS. In all cases, the Message_DIGEST INDEX is only used if values are provided for all FIELDS in the associated KEY.

### 23.6.3 Using Indices_Banned For Low Overhead Record Classes

When working with small files that are known to be valid, the overhead of involving (and invoking) Sizup and having a .siz file present can be too much given that simple sequential access by scanning all record characters upto newlines is considered sufficient. To achieve this end and yet maintain the benefits of using an rcd, simply append an `<Indices_Banned, yes>` note off the KEYS description in the associated rcd. Here is an rcd fragment illustrating how this is done:
The _Indices Banned_ note hanging off the KEYS node informs the system neither to build nor to expect any indices whatsoever; this includes not building a .siz file -- nor for that matter, a free tree. Consequently, in this circumstance, there will be no reuse of the slots for deleted records.

### 23.6.4 RECORD_CLASSes defined on partial records

If a `<More_Fields_Follow, yes>` note is placed on the FIELDS description in an rcd, Daytona is instructed that the data file will contain more FIELDS per record than are described by the rcd (cf., rcd.HALFPART, rcd.HALFPART2). With read queries, Daytona will simply ignore any characters that come after the last FIELD mentioned in the rcd (halfrec.1.Q); update transactions are forbidden because they would create inhomogeneous files containing records with differing numbers of FIELDS. Daytona does support building indices for such record classes; obviously, the corresponding KEYS can only make reference to FIELDS that the rcd does.

### 23.7 Record-Level Data Compression

Daytona offers a simple procedure whereby records can be compressed to a fraction of their original size and yet retain the indexability and other functionality that Cymbal offers their uncompressed versions. Success in compression varies according to the nature of the data but achieving compressed data taking 25-35% of its uncompressed, plaintext volume is quite realizable in practice. Obviously, there are tremendous space savings here and hence much more information can be stored for the same amount of money but these advantages come with the price that, obviously, the data must be uncompressed to use it -- and compression/decompression is computationally expensive. However, depending on the speed of the CPUs, the time it takes to decompress records may be more than paid back by the resultant decrease in the amount of time it takes to read the compressed data in from disk. Furthermore, when using an index to retrieve compressed records, the decompression time is small relative to the disk seek and read times which are on the order of 8 milliseconds per seek these days. That’s a long time on machines whose cycle times are a few nanoseconds.
Indeed, the hallmark of Daytona record compression is that it does maintain the efficient indexability of individual records. Daytona uses a static dictionary compression method that compresses the records one at a time. The indices point to the starts of compressed records. Consequently, only the pointed-to records need to be decompressed. Contrast this with using `compress` or `gzip` to compress an entire file at a time; obviously, to get any particular record in such a file decompressed, it would be necessary to decompress the entire file at least up to the point of where that record appeared. Thus the non-record-sensitive compress-the-entire file approach is essentially useless for serious database management and so, Daytona has its own strategy which works much better for data management, although it does not achieve the compression efficiencies of `gzip`.

To achieve the best compression possible, before beginning the record-level compression procedure, first do the following:

1. Use the HEKA types as much as possible to achieve field-level compression.
2. For FIELDS that have sizeable values in a not-too-large set, use an Input Filter Fun to map that set of values into a set whose values take as few bytes as possible.
3. Use a FIELD Default Value equal to the mode of the value distribution for each FIELD.
4. Arrange the order of the FIELDS so that the FIELDS with the least information in them (in the sense of entropy) appear at the end of the record; these will also tend to be the ones that are the least accessed. (For example, a FIELD whose mode takes up 90% of its value distribution is carrying less information than one whose mode has 20% of the distribution and which has lots of different values with sizeable probabilities of occurring.)

Once these recommendations have been put into effect, then a good sign that they have been effective is that the ends of most records will consist of a sequence of field separators. Daytona will do a great job of compressing those.

Obviously, the first step to doing record-level compression is to find a good dictionary. Daytona does this with the Find_Dict.1 program:

```
# Find_Dict.1 +?
usage: Find_Dict.1 <data_fl> [ <max_dicts=-1> <lo_water=256> <hi_water=2560> \ 
  <test_len=100> <frac_sampled=.10> ]
cat fl_of_fl | Find_Dict.1 @ [ <max_dicts=-1> <lo_water=256> <hi_water=2560> \ 
  <test_len=100> <frac_sampled=.10> ]
ls fl_pat | Find_Dict.1 @ [ <max_dicts=-1> <lo_water=256> <hi_water=2560> \ 
  <test_len=100> <frac_sampled=.10> ]

com_char = % by default
```

Find_Dict.1 scans the one or more data files that are fed to it and produces a sequence of dictionaries that are located in the same directory as the path give for the data. Please note that it will only produce a dictionary once it has read enough data. The defaults for the arguments are fine. (The `com_char` refers to the one used in the data file being processed.) By the way, Find_Dict.1 can run indefinitely when given enough data to sample -- unless the `max_dicts` argument is given. Usually there
is no point in letting it generate more than 20 or so dictionaries.

Eval_Dicts is used to find the best dictionary of a set of them by using each one to compress each of a set of test files:

```python
# Eval_Dicts +?
usage: Eval_Dicts <fl_of_dict_fl> <fl_of_data_fl>
```

The first argument to Eval_Dicts is a (path to a) file which contains one or more Find_Dict. file dictionary paths, one to a line. Likewise, the second argument is a file path containing test data file paths, one to a line. Eval_Dicts produces a report which identifies the best dictionary of the lot. (Since the compression ratios listed in this report divide compressed sizes by uncompressed sizes, the lower the fraction, the better the compression.)

Having found the best dictionary of the lot, use Cmpl_Dict.1 to compile it into a form which Daytona can use to compress and decompress records:

```python
# Cmpl_Dict.1 +?
usage: Cmpl_Dict.1 <dict_fl> [ <method=DC_Dict1> [<out_cmpld_dict_fl>] ]
# Cmpl_Dict.1 my_data.dict20 DC_Dict1
# ls my_data.dict20.CD
```

The method to use here is currently always DC_Dict1. The default output file name is the input file name suffixed with .CD. Here are a couple variants:

```python
# Cmpl_Dict.1 my_data.dict20
# Cmpl_Dict.1 CORDER.dict DC_Dict1 CORDER.CD
```

Use Dict_Encode (Dict_Decode) to manually compress (decompress) data:

```python
# Dict_Encode +?
usage: Dict_Encode <comp_dict_fl> [ <in_fl> ] [ <out_fl> ]
Dict_Decode <comp_dict_fl> [ <in_fl> ] [ <out_fl> ]
Dict_Map_Rec <comp_dict_fl> # prints out dictionary
# Dict_Encode my_data.dict20.CD <test_fl >out_fl
# Dict_Encode my_data.dict20.CD test_fl2
# Dict_Decode my_data.dict20.CD test_fl2.DZ >test_fl13
```

Dict_Encode produces a compressed file whose name by default is that of the source file suffixed with .DZ. If Dict_Decode is offered a file to decode whose name does not end in .DZ, it will abort with an error message. If these conventions become a nuisance, just pipe the data in and out of the executables.

Caveat: These compression techniques will not work properly if the file to be compressed contains escaped newlines.

Given that the data file has been record-level-compressed by Daytona, to use that file in Cymbal programs where that data will be automatically decompressed (and compressed), simply annotate the FILE description in the appropriate rcd with a Rec_Map_Spec_File note:
The curious choice of terminology, *Rec_Map_Spec_File*, is understandable upon realizing that this entire mechanism is really just associated with mapping records from some special format on disk to the ASCII flat file form which Daytona can understand natively (the so-called DC format). The mapping doesn’t have to be compression per se -- in theory, it could also be encryption or some hybrid or something else again. If a *Rec_Map_Spec_File* rcd annotation is not present for a FILE description, then a *Default_Rec_Map_Spec_File* annotation will be used instead as soon as it is found by looking at any ancestor FILE_INFO_FILE or BINS description.

Note that a compiled dictionary can be printed out in human-readable form by doing:

```
Dict_Map_Rec <comp_dict_fl>
```

### 23.8 Schema Evolution And Data Migration

When the database schema changes from one release of an application to the next, the application developer begins to feel the need to provide to the application user an automated way to migrate their data from one release to the next. Daytona facilitates the writing of such migration code by providing the application developer with an easy way to refer to multiple versions of the same record class in the same Cymbal query.

Versions may be specified for record classes by placing a Version note under the RECORD_CLASS node in the corresponding rcd, as in:

```
#{ RECORD_CLASS ( ORDER )
   <Multiplicity, ^0->^>
   <Version, ^4.1.2 -- May 19, 1992^>
   ...
}#
```

If no version is specified explicitly for the rcd, then it inherits the Version for its associated application, if any. Such a Version is specified by means of a Version note in the corresponding apd, as in:

```
#{ APPLICATION ( orders2 )
   <Default_Data_Interface, DC>
   <Version, 2.0>
}#
```
Otherwise, the record class is not considered to be versioned. Versions may be elements of the following types: STRINGS, INTS, FLTS, DATES, THINGS. (Although, of course, when writing versions in apd’s and rcd’s, the usual hatted quoting convention for rcd’s and apd’s should continue to be observed.)

The `from_version` keyword is all that is needed in a Cymbal description to refer to the data stored for a particular version of a record class:

```cymbal
there_is_a SUPPLIER from_version ^1-1-84^ where(
   Number = 101
   and Name = .x )
```

Please note that the argument to the `from_version` keyword must be a constant: it cannot be variable dereference or function call. (See also the discussion of `with_version` for queries in Chapter 6.)

Now what does all of this have to do with schema evolution and data migration? The answer is simple: to migrate the data from one version of a record class to another, just write Cymbal that reads the data, record by record, from the first version of the record class and writes the transformed records out in the format specified by the second version of the record class.

For example, consider the following Synop-produced specification for Version 1.0 of record class SUPPLIER:
RECORD_CLASS: SUPPLIER
    Version: 1.0

BINS:

FILE: SUPP
    Source: /home/oursys/rel.1.0
    Unit_Sep: "::
    Comment_Beg: "#"

KEYS:

KEY 1: Unique [ Number ]
    INDEX 1: Unique btree

KEY 2: Unique [ Name ]
    INDEX 2: Unique btree

KEY c: Non-unique [ City ]
    INDEX c: Non-unique btree

KEY t: Non-unique [ Telephone ]
    INDEX t: Non-unique btree

FIELDS:

Field 1: INT(_short_) Number
    Min_Value: 300
    Max_Value: 4000

2: STR(30)  Name
    Validation_RE: "^\([A-Z]+[-a-z.,_0-9]*\)+$"RE

3: STR(25)  City
    Multiplicity: 0->1

4: STR(25)  Telephone
    Multiplicity: 0->1

The next version of this record class is given by:
RECORD_CLASS:  SUPPLIER
  Version:  2.0

BINS:

FILE:  SUPP.2.0
  Source:  /home/oursys/rel.2.0
  Unit_Sep:  "|"
  Comment_Beg:  "%"

KEYS:

  KEY 1:  Unique  [ Number ]
    INDEX 1:  Unique btree

  KEY c:  Non-unique  [ Location ]
    INDEX c:  Non-unique btree

FIELDS:

  Field 1:  INT(_long_)  Number

  Field 2:  STR(1)  Rating
    Multiplicity:  0->1

  Field 3:  STR(25)  Telephone
    Multiplicity:  0->1
    Validation_RE:  "^[0-9-]+$"RE

  Field 4:  STR(*)  Location
    Multiplicity:  0->1
    Default_Value:  "Murray Hill"

Notice how many things have changed from one version to the next:

- The Name field has been deleted.
- The Rating field has been added.
- The City field has been renamed Location.
- The City/Location field now appears in the records after the Telephone field.
- The type of the Number field has been changed from INT(_short_) to INT(_long_).
- The type of the Location field has been changed from STR(25) to STR(*).
- KEY 2 and KEY t have been dropped.
- The delimiter and comment character have been changed.
- The Min_Value and Max_Value constraints on the Number field have been dropped.
• The data file has a new name and is stored in a new directory. (All that is necessary is that the file path of the new data file be different from the original one.)

There are additional changes as well that are not specified in the rcds:
• All Number field values are to be increased by 1000.
• All Location field values are to be written in all capital letters.
• All Rating field values are to be initially missing.

The entire data migration can be accomplished by executing the following simple query (evolve.2.IQA):

do Evolve_SUPPLIER;

global_defs:

define PROC transaction task: Evolve_SUPPLIER
{
  for_each_time [ .nbr, .city, .phone ]
  is_such_that(
    there_is_a SUPPLIER from_version "1.0"
    where(
      Number = .nbr
      and City = .city
      and Telephone = .phone
    )
  )
  do Change( so_that(
    there_is_a SUPPLIER from_version "2.0"
    where(
      Number = .nbr+1000
      and Location = upper_of(.city)
      and Telephone = .phone
      and Rating Is _absent_
    )
  ));
}]

This query should be pretty much self-explanatory at this point. Note that arbitrary functions can be called to generate new values for the fields. The Rating Is _absent_ specification could have been left out since that is the default behavior when doing Cymbal adds.

Observe that the nature of the migration is specified concisely in two places: 1) declaratively, in the rcds and 2) procedurally, in the query. The creation of the data under the new release is done by the query above and by Sizup, which creates the indices.

Please note that an empty SUPP.2.0 file needs to exist before running this query; Sizup -create can create such a file plus empty indices, all ready to be inserted into. It’s also important that the two rcds
for SUPPLIER be in different aars (with different names) that are in different directories; this will avoid a collision when Tracy writes SUPPLIER.c for both. DS_APPS and DS_PATH must, of course, have values that enable Tracy to find the relevant rcds; DS_APPS=orders_1:orders_2 is illustrative. And of course, a Source note for an old data file in the old rcd must evaluate to something different than the corresponding Source note does in the new rcd for the new version of the data file -- or else, data will be written to a file being read from and chaos will result.

Note that a Cymbal description which does not use a from_version keyword is associated with the first rcd for its record class found in the sequence of archives specified by $DS_APPS. Once migration is complete, just set $DS_APPS so that the first aar searched is the new one and then the migrated queries will work without needing to use a from_version keyword, even though the aar may still contain Version specifications.

These techniques do not fully address the problem of migrating the queries from one version to another; it should be clear though that queries that do not use the from_version keyword and which reference FIELDS that don’t change from one version to the next can probably migrate without change.

If there are many tables to be migrated, then the user may wish to use backtalk to generate the general form of the migration code by processing the corresponding rcds.

The preceding discussion assumes that the old and new instances of the database are being managed by the same release of Daytona. If at the outset this is not the case, then it must be made so by doing the following:

1. Migrate the old aar under the old release of Daytona to the new release of Daytona. The Daytona release notes will specify how this can be done; typically, this migration can be forced by running Sizup with the -Modifiai option.
2. Regenerate the file I/O files (e.g., SUPPLIER.[ch]) for the Daytona-migrated old aar to the new release of Daytona by using the -obj option to DS Resync.
3. Prepare the old data to work with the new Daytona release. Since the Daytona data format either almost never changes or changes in a completely upwardly compatible way, this amounts to ensuring that the indices present for the old data are consistent with the new Daytona release. The Daytona release notes will indicate whether the index structure has changed or not; usually, it does not. If it has changed, then DS Resync running under the -indices option and under the new Daytona release will recreate the indices for the record classes contained in the Daytona-migrated old aar.
4. Of course, once the preceding steps have been done, use the new Daytona release to run Tracy on the user’s schema evolution queries for migrating the data from its (Daytona-migrated) old aar format to the format specified by the new aar.
24. Conclusion

Cymbal is a very powerful fourth-generation language which synthesizes the best of declarative and procedural constructs. The declarative portion of Cymbal alone is able to capture queries that:

- make general use of quantifiers and negation,
- make simultaneous reference to segments of all types and levels in relational and hierarchical tree schemas,
- make reference to an arbitrary number of files, including multiple files per record class,
- make perhaps nested calls to the aggregate functions \textit{sum}, \textit{avg}, \textit{min}, \textit{max}, and \textit{count}, as well as to the second-order aggregate functions \textit{var}, \textit{stdev}, \textit{covar}, and \textit{corr} (which, incidentally, are not necessarily available in SQL),
- apply several aggregate functions simultaneously to groups making up very generally defined partitions of object classes,
- make reference to subqueries within queries,
- define sets and lists by means of first-order logic membership conditions,
- define in-memory tables (known as boxes) complete with ordering criteria and indices,
- make use of scalar- and tuple-valued multidimensional associative arrays,
- tokenize I/O streams,
- use SQL \textit{select} statements,
- re-use data buffers to improve performance,
- define new variables whose values aren’t taken from database fields,
- handle missing values correctly in a flexible manner,
- make use of LIST/SET-valued fields,
- use system and user-defined functions, procedures, and predicates, and
- use generalized transitive closure concepts.

The procedural portion of Cymbal takes the language into the realm of being a high-level programming language. The procedural portion has essentially the functionality of awk, sed, and the programming language part of the Korn shell as well as the base functionality of Perl (except much faster). These procedural constructs allow for many more queries to be expressed than would be possible using declarative Cymbal or SQL alone. Of particular note is the “for_each_time” loop construction which allows for a sequence of procedural commands to be executed each time an assertion is true for values of variables used in those commands.
Cymbal is made available to users of Daytona’s DC flat file DBMS by means of the Tracy Cymbal-to-C code translator.
DSQL Grammar

The grammar for DSQL is an extension of that for the query portion of ANSI standard SQL. Specifically, DSQL conforms to the ANSI SQL2 draft standard for the next generation of SQL. In addition, DSQL uses Cymbal syntax to permit users to access any of Daytona’s built-in functions and predicates as well as any further functions that the user may wish to add. DSQL also includes the var() and stdev() aggregate functions which SQL does not. It also supports all of Cymbal’s datatypes which include the HEKA, RE (regular expression), and TEXT datatypes not found in ANSI SQL. This section introduces the user to the DSQL grammar.

Incidentally, the ANSI standard portion of Daytona’s implementation of SQL is not case sensitive, although of course the added Cymbal syntax does remain case-sensitive. This means that the SQL portion of DSQl queries can be typed using any mixture of upper- or lower-case letters. However, Daytona does require that whenever a column name is made into an UPLOW using Daytona’s standard algorithm, it must equal the desired field name in the appropriate Daytona record class description. So, for example, if "dept_head" is a DSQL column name, then "Dept_Head" must be the corresponding field name in the record and not, say, "DEPT_Head". For more details on this and other DSQL matters, please see Chapter 4.

The terminology used for this grammar is taken from the ANSI standard. It could be better. The "Spec" abbreviation used below means Specification and "Expr" means Expression. Other abbreviations should be easy to figure out.

Syntactic classes here are indicated by underlined italic strings whereas, with the exception of the grammar symbols, the other characters are to be taken literally. The grammar symbols are “::=”, “;”, “|”, “[”, “]”, “^”, “*”, and directed double quotes. “[a^*]” is read as meaning \( A \mid a \) where “\( A \)” denotes the empty string, “[a^*]” is read as meaning \( A \mid a \mid a a \mid \ldots \), and “[a^+]” is read as meaning \( a \mid a a \mid \ldots \).

As for some of the classes of grammar terminals, a lower is a string of characters beginning with a lower-case letter and continuing with only lower-case letters, digits, and underscores. An upper is defined similarly. A classmb is an upper which is the singular form of an entity class name. A preposition is a lower which is a preposition. An uplow begins with an upper-case letter and continues with any mixture of letters, digits, and underscores containing at least one lower-case letter. A cstring is a string enclosed in undirected double quotes, a literal is a string enclosed in matching single quotes, and a hatted is a string enclosed in hats.

cstrings are written using double quotes like C’s strings with \ as the escape character. SQL’s strings are written using single quotes with ’ as the escape character. So, “abc” is a cstring and ‘abcd’
is an `sqlstring`. A c is an ASCII character.

Comments may be written using the C-style /* and */ notation or by using ANSI SQL’s new-line terminated sequences beginning with -- .

```sql
-- this is an ANSI SQL comment.
```
DSQL Grammar

Statements

\[
\text{SqlProgram} ::= \text{SqlTopStmt} \mid ; \text{SqlTopStmt} \mid ^* \\
\text{SqlTopStmt} ::= \text{SqlQuery} \mid \text{SqlTransaction} ; \\
\text{SqlStmt} ::= \text{SqlQuery} \mid \text{SqlModification} ; \\
\text{SqlQuery} ::= \text{QueryExpr} \mid \text{OrderLimitParallelClause} \mid ^* \\
\text{QueryExpr} ::= \text{InterQueryTerm} \\
\mid \text{QueryExpr} \text{ union } \text{all} \text{ InterQueryTerm} \\
\mid \text{QueryExpr} \text{ union } \text{InterQueryTerm} \\
\mid \text{QueryExpr} \text{ except } \text{all} \text{ InterQueryTerm} \\
\mid \text{QueryExpr} \text{ except } \text{InterQueryTerm} ; \\
\text{InterQueryTerm} ::= \text{QueryTerm} \\
\mid \text{InterQueryTerm} \text{ intersect } \text{all} \text{ QueryTerm} \\
\mid \text{InterQueryTerm} \text{ intersect } \text{QueryTerm} ; \\
\text{QueryTerm} ::= \text{QuerySpec} \mid \text{TableConstructor} \mid ( \text{QueryExpr} ) ; \\
\text{SubQuery} ::= ( \text{QueryExpr} ) ; \\
\text{QuerySpec} ::= \text{select} \{ \text{all} \mid \text{distinct} \} \text{ SelectList } \text{ TableExpr} ; \\
\text{SelectList} ::= ^* \mid \text{SelectValueExpr} \mid , \text{SelectValueExpr} \mid ^* \\
\text{SelectValueExpr} ::= \text{ValueExpr} \mid \{ \{ \} \text{ as } \} \text{ ColLabel} \mid ^* \\
\text{ColLabel} ::= \text{literal} \mid \text{identifier} ; \\
\text{OrderLimitParallelClause} ::= \text{OrderByClause} \mid \text{LimitToClause} \mid \text{ParallelForClause}
\]
OrderByClause ::= order by SortSpec [ , SortSpec ]*

LimitToClause ::= limit to ValueSpec [ -> ] ValueSpec [ ] [ ]

ParallelForClause ::= parallel for ValueSpec

SortSpec ::= SortColumn [ asc | desc ]

SortColumn ::= integer | ColumnSpec | SetFunSpec | FunctionCall

TableConstructor ::= values ValueExprSeq

Modifications And Transactions

SqlTransaction ::= SqlModification
| begin [ work ] ; SqlStmt [ ; ] ; SqlStmt [ ] ; end [ work ]

SqlModification ::= Deletion | Insertion | Update

Deletion ::= delete from tablename [ ] CorrelationSpec [ ] [ ] WhereClause [ ]

Insertion ::= insert into tablename [ ] CorrelationSpec [ ]
| ( ColumnName [ ] , ColumnName [ ] ) [ ] NewValues

NewValues ::= values ( ValueExprSeq ) [ ] , ( ValueExprSeq ) [ ]
| QuerySpec

Update ::= update tablename [ ] CorrelationSpec [ ]
| set SetClause [ , SetClause ] [ ] [ ] WhereClause [ ]

SetClause ::= ColumnSpec = ValueExpr

Table Expressions
TableExpr ::= FromClause WhereClause GroupByClause HavingClause ;
FromClause ::= from TableReference , TableReference * ;
TableReference ::= tablename CorrelationSpec * | SubQuery CorrelationSpec ( ColumnName , ColumnName * ) * | TableReference JoinType join TableReference JoinSpec |
| ( TableReference ) ;
JoinType ::= inner | left | outer * ;
JoinSpec ::= on SearchCondition |
| using ( ColumnName , ColumnName * ) ;
tablename ::= projname . | appname . | identifier ;
CorrelationSpec ::= as identifier | identifier ;
WhereClause ::= where SearchCondition ;
GroupByClause ::= group by ColumnSpec , ColumnSpec * ;
HavingClause ::= having SearchCondition ;

Search Conditions
SearchCondition ::= BooleanTerm | SearchCondition or BooleanTerm ;
BooleanTerm ::= BooleanFactor | BooleanTerm and BooleanFactor ;
BooleanFactor ::= BooleanPrimary | not BooleanPrimary ;
BooleanPrimary ::= Predicate | ( SearchCondition ) ;
Predicates

```sql
Predicate ::= ExistsPred | ComparisonPred | QuantifiedPred | InPred | NullPred | BetweenPred | LikePred |
        | nowpred [ ValueExpr I , ValueExpr I* ] |
        | ValueExpr nowpred I ValueExpr I* |
        | ValueExpr nowpred I TaggedArg I* |
        ; |
nowpred ::= uplow |
        ; |
ExistsPred ::= exists SubQuery |
        ; |
ComparisonPred ::= ValueExpr I not I? comparop ValueExpr |
                | ValueExpr I not I? comparop SubQuery |
        ; |
not ::= not | ! |
        ; |
comparop ::= = | < | > | >= | <= | <> |
        ; |
QuantifiedPred ::= ValueExpr I not I? comparop quantifier SubQuery |
        ; |
quantifier ::= all | some | any |
        ; |
InPred ::= ValueExpr I not I? in InSpec |
        ; |
InSpec ::= SubQuery | ( ValueExprSeq ) | IntervalExpr |
        ; |
NullPred ::= ColumnSpec is I not I? null |
        ; |
BetweenPred ::= ValueExpr I not I? between ValueExpr and ValueExpr |
        ; |
```
LikePred ::= ValueExpr | not | like | escape | ValueSpec | ValueSpec |

Function Call Terms

ValueExpr ::= ValueSpec
            | ColumnSpec
            | ValueExpr arithop ValueExpr
            | + | - | ValueExpr
            | CaseSpec
            | SetFunSpec
            | (: base_type | ( subclass ) | ) ValueExpr
            | cast ( ValueExpr as ColumnType )
            | trim( | leading | trailing | both | trim | from | ValueExpr )
            | substring ( ValueExpr from ValueExpr for ValueExpr )
            | WindowFunSpec
            | row | ( ValueExprSeq )
            |
ValueExprSeq ::= ValueExpr , ValueExpr *
               |
CaseSpec ::= case WhenClauseList end
            | case ValueExpr TermWhenClauseList end
               |
WhenClauseList ::= when SearchCondition then ValueExpr | else ValueExpr |
               |
TermWhenClauseList ::= when ValueExpr then ValueExpr | else ValueExpr |
               |
IntervalExpr ::= [ ValueExpr -> | ValueExpr ] | TaggedArg |
               |
arithop ::= + | - | * | / | %
           | ++ | ** | % | << | >>
            |
function ::= lower
            |
SetFunSpec ::= count ( )
              | setfun ( distinct ValueExpr )
              | setfun ( all ValueExpr )
setfun ::= count | avg | max | min | sum
| median | stdev | var

WindowFunSpec ::= row_number ( )
| row_number ( ) over ( [ PartitionByClause ] , [ OrderByClause ] )
| row_number ( ) over ( [ OrderByClause ] , [ PartitionByClause ] )

PartitionByClause ::= partition by ColumnSpec [ , ColumnSpec ]*

args ::= ( ValueExprSeq )
| ( [ TaggedArg ]* )

TaggedArg ::= ^ preposition ^ ValueExpr | ^ preposition ^

Other Terms

ValueSpec ::= . lower | literal

ColumnSpec ::= ColumnName | identifier . ColumnName

ColumnName ::= identifier

identifier ::= lower | uplow | upper | hatted

literal ::= integer | float | sqlstring | cystring | date | text

Data Definition Statements (for Daisy only)

DDLSstmt ::= CreateStmt | DropStmt

CreateStmt ::= CreateTable | CreateIndex | CreateFileBin | CreateFifBin

CreateTable ::= create table tablename [ version 'version' ] application app [ ]
( ColumnDefn [ , ColumnDefn ] )
| create table tablename [ version 'version' ] application app [ ]
like tablename [ version 'version' | ]
create table tablename [ version 'version' | ] [ application app | ]
file 'filename' | Marker | as SqlQuery | ;

Marker ::= DelimDefn | tuple_delim 'cc' | comment 'c' | ;

DelimDefn ::= delim 'c' | delims 'cc' | ;

ColumnDefn ::= ColumnName ColumnType | ColumnAttr | ;

ColumnAttr ::= | not | nullable
| default | literal
| min | literal | max | literal
| validation_re | literal | partitioner | ;

CreateIndex ::= create | unique | cluster | index | indexname | on | tablename
| version 'version' | | ( | ColumnName | , | ColumnName | ) | ;

CreateFileBin ::= create file bin 'binname' | on | tablename [ version 'version' | ]
| partition values | ( | literal | , | literal | ) | | ;
| source | 'sourcename' | | Marker | | ;

CreateFifBin ::= create file_info_file bin 'binname' | on | tablename
| version 'version' | | source | 'sourcename' | | Marker | | ;
| file model | Marker | ;

DropStmt ::= DropTable | DropIndex | DropFileBin | DropFifBin | ;

DropTable ::= drop table tablename | version 'version' | | erase | ;

DropIndex ::= drop index indexname | from | tablename | version 'version' | ;

DropFileBin ::= drop file bin 'binname' | from | tablename
| version 'version' | | erase | ;

DropFifBin ::= drop file_info_file bin 'binname' | from | tablename
// version 'version' I erase I
;
ReadStmt ::= read I from literal I into ReadVbl I , ReadVbl I*
;
ReadVbl ::= I Type I . lower
;
Cymbal Grammar

Cymbal’s grammar defines a rich language with a variety of syntactic forms designed to promote the easy and natural expression of queries. This section presents a formal exposition of the grammar.

Syntactic classes here are indicated by underlined italic strings whereas, with the exception of the grammar symbols, the other characters are to be taken literally. The grammar symbols are “ ::= ”, “ ; ”, “ | ”, “ [ _ ] “, “ [ _ ]? _ _ ”, “ [ _ ]* _ _ ”, “ [ _ ]+ _ _ ”, and directed double quotes. “ [a]” is read as meaning a, “ [aL]” is read as meaning Λ | a where “ Λ ” denotes the empty string, “ [aL+]” is read as meaning Λ | a | a a | . . . , and “ [aL+” is read as meaning a | a a | . . . .

Grammar non-terminals are represented by underlined UPLOWS as illustrated by Program.

Classes of grammar terminals are represented by underlined LOWERS. Here are some special ones: a **lower** is a string of characters beginning with a lower-case letter and continuing with only lower-case letters, digits, and underscores. An **upper** is defined similarly. A **class_name** is an **upper** which is the singular form of an entity class name. A **preposition** is a **lower** which is a preposition in the sense that it is an English phrase that takes a noun phrase argument. An **uplow** consists of a sequence of letters, digits, and underscores for which the first letter is upper-case and some subsequent letter is lower-case. A **string** is a string enclosed in undirected double quotes, a **literal** is a string enclosed in matching single quotes, and a **thing** is a string enclosed in hats.
Procedural Constructs

Please note that C semicolon usage conventions can also be used.

\[
\begin{align*}
\text{Program} & ::= \text{EnvStmt} \; ] \; \text{CmdSeq} \; ] \; \text{EnvStmt} \; ] \; \text{GlobalDefs} \; ] \\
\text{BraceProg} & ::= \{ \; \text{FppEnvStmt} \; ] \; \text{CmdSeq} \; ] \\
\text{CmdSeq} & ::= \text{ProgAtom} \; ] \; \text{ProgAtom} \; ] \\
\text{ProgAtom} & ::= \text{lower} : \; ] \; \text{Command} \; | \; \text{lower} : \; ] \\
\text{Command} & ::= \text{Assignment} \; | \; \text{When} \; | \; \text{Switch} \\
\text{GlobalDefs} & ::= \text{global defs} \; ] \; \text{gbladj} \; ] \; : \; ] \; \text{GlobalDef} \; ] \\
\text{gbladj} & ::= \text{with_version} \; \text{Subject} \\
\text{GlobalDef} & ::= \text{TaskDef} \; | \; \text{DeclFppDef} \; | \; \text{ClassDef} \\
\text{ProCall} & ::= \text{KeywordArg} \; ] \; \text{do procedure} \; \text{KeywordArg} \; ] \\
\text{Assignments} & ::= \text{set TgtAtom} = \text{Subject} \\
\text{set TgtAtom} = \text{FunCall} \; \text{otherwise} \; \text{NonWhenCmd} \\
\text{set TgtAtom} = \text{FunCall} \; \text{otherwise_switch} \; \text{SwitchCases}
\end{align*}
\]
| set TgtAtom ++ | set TgtAtom -- |
| set TgtAtom arithop = Subject |
| set [ AssignTgtSeq ] = SomeSubjects |
| set . BoxVbl = Aggregate |
| set [ AssignTgtSeq ] = read( [ KeywdArg ]* ) |
| set TokensAsn |
| set TupleAggsAsn |

\[
\text{TgtAtom} ::= \text{ValCall} \mid \text{skolem} \\
\]
\[
\text{AssignTgt} ::= \text{TgtAtom} \mid [ \text{AssignTgtSeq} ] \\
\]
\[
\text{AssignTgtSeq} ::= \text{AssignTgt} \ [ , \text{AssignTgt} ]* \\
\]
\[
\text{arithop} ::= - \mid + \mid * \mid / \\
\]

**Grouping**

\[
\text{Do} ::= \{ \text{do} \} \text{BraceProg} \\
\text{SqlStmt} \mid \$[ \text{SqlProgram} ]\$ \\
\]

**Conditionals**

\[
\text{When} ::= \text{when} ( \text{Assertion} ) \text{NonWhenCmd} \\
\text{else when} ( \text{Assertion} ) \text{NonWhenCmd} ]* \\
\text{else} \text{NonWhenCmd } ]* \\
\]
\[
\text{when} ::= \text{when} \mid \text{if} \\
\]
\[
\text{Switch} ::= \text{switch} ( \text{Subject} ) \text{SwitchCases} \\
\]
\[
\text{SwitchCases} ::= \{ \text{do} ]* ( \[ \text{Case} ] , \text{Case} ]* ]* \\
\text{switchelse} \text{Do } ]* \\
\]

Copyright © 2013 AT&T
All Rights Reserved.
September 15, 2013
Case := case (ExtendedPred) Do ;

switch := switch | switch_on ;

switchelse := else | default ;

Loops

Loop := loop BraceProg |
while | until (Assertion) |
LoopModifier |
loopstart BraceProg else Do |

| loopstart BraceProg while | until (Assertion) |

LoopModifier := |
renewing_with BraceProg |
before_doing_the_first BraceProg |
after_doing_the_last BraceProg |

loopstart := loop | do |

ForEachTimeLoop := |
ForEachTime SomeVblSpecs is_such_that BoundedAsn |
LoopModifier |
Do else Do |

| ForEachTime SomeVblSpecs BoxFormerPred BoundedAsn HybridBoxKeywdArg |
LoopModifier |
Do else Do |

| ForEachTime SomeVblSpecs Is_In Aggregate UseBoxKeywdArg |
LoopModifier |
Do else Do |

| ForEachTime SomeVblSpecs CompoundPred |
LoopModifier |
Do else Do |
**ForEachTime** ::= for_each_time | for_the_first_time | for_the_last_time

**Branches**

**Break** ::= break [ ( integer ) ]? | leave [ ( integer ) ]? 

**Continue** ::= continue [ ( integer ) ]? | loop_again [ ( integer ) ]? 

**Goto** ::= goto lower | go lower 

**Return** ::= return [ ( Subject ) ]? 

**Update Commands**

**ChangeProCall** ::= do Change so_that ( Desc ) | do Change so_that ( UpdateBoxAsn ) 

**UpdateBoxAsn** ::= SomeSubjects [ Is_In | Is_In_Again | Is_Not_In ] 

Aggregate [ UseBoxKeywdArg ]* 

**Examples Of Cymbal Updates**

**Deletions**

```
do Change so_that( 
    there_is_no class_name [ IdNote ]? [ DescTail ]? )
```

```
ForEachTime [ SomeVblSpecs ] is_such_that [ BoundedAsn ]{
    do Change so_that( this_is_no class_name )
}
```

(Of course, the this_is_no must correspond to some there_is_a in the BoundedAsn.)

**Additions**

```
do Change so_that( 
    there_is_a class_name [ IdNote ]? [ DescTail ]? )
```
do Change so_that(
    there_is_a_new class_name [ IdNote [ ] DescTail [ ] ] ) )

Updates

ForEachTime [ SomeVblSpecs is_such_that [ BoundedAsn { do Change so_that(
    this_is_a class_name [ IdNote [ ] DescTail [ ] ] ) )
}

when( BoundedAsn )
do {
  do Change so_that(
    this_is_a class_name [ IdNote [ ] DescTail [ ] ] )
}

(Of course, the this_is_a must correspond to some there_is_a in the BoundedAsn.)
Declarative Constructs

Assertions

Assertion ::= SatClaim
  | SomeSubjects ExtendedPred
  | not Assertion
  | Assertion and Assertion
  | Assertion or Assertion
  | if Assertion then Assertion
  | if Assertion then Assertion else Assertion
  | BoundedAsn iff BoundedAsn
  | for_each SomeVblSpecs
      [ such that BoundedAsn ] conclude BoundedAsn
  | for_each SomeVblSpecs [ if ever( Assertion then Assertion ) ]
  | existential SomeVblSpecs such that BoundedAsn
  | somehow BoundedAsn
  | there_exists_count ExtendedPred [ over SomeVblSpecs ]
      such that BoundedAsn [ implies BoundedAsn ]?
  | BoundedAsn

SatClaim ::= SomeSubjects CompoundPred
  | Pred [ SubjectSeq ]
  | SomeSubjects OpCond
  | TokensAsn
  | TupleAggsAsn
  | BoxAsn
  | _true_ | _false_

not ::= ! | not | its_not_so_that

iff ::= iff | if_and_only_if

existential ::= there_exists | there_does_not_exist | there_exists_no

BoundedAsn ::= ( Assertion ) | Desc
TokensAsn ::= ListTerm = tokens( new_channelKeyWdArgs readKeyWdArgs ) ;

TupleAggsAsn ::= ListTerm = aggregates ( of [ PartialAggFunCallSeq ] each_time BoundedAsn ) ;

BoxAsn ::= SomeSubjects Is_In Aggregate [ UseBoxKeywdArg ]* SomeValCalls BoxFormerPred BoundedAsn [ HybridBoxKeywdArg ]* ;

BoxFormerPred ::= Is_Something_Where | Is_The_Next_Where
| Is_The_First_Where | Is_The_Last_Where ;

UseBoxKeywdArg ::= in_selection_order
| in_reverse_selection_order
| in_arbitrary_order
| in_random_order
| in_lexico_order
| in_reverse_lexico_order
| sorted_by_spec [ SubjectSeq ]
| with_candidate_index_vbl Vbl
| with_selection_index_vbl Vbl
| with_sort_index_vbl Vbl
| with_candidate_index Subject
| with_selection_index Subject
| with_sort_index Subject
| as_quantile Subject ;

BuildBoxKeywdArg ::= with_no_duplicates
| with_duplicates_ok
| with_default_arbitrary_order
| with_default_selection_order
| with_lexico_order
| with_reverse_lexico_order
| with_sort_spec [ SubjectSeq ]
| with_sort_specs Tuple
| with_candidate_index_vbl Vbl
| with_candidate_indices_stored
with_selection_index_vbl \ Vbl
| with_selection_indices_stored
| with_sort_index_vbl \ Vbl
| with_sort_indices_stored
| with_random_indices_stored
| selecting_when \ BoundedAsn
| stopping_when \ BoundedAsn

HybridBoxKeywdArg ::= UseBoxKeywdArg
| with_no_duplicates
| with_duplicates_ok
| selecting_when \ BoundedAsn
| stopping_when \ BoundedAsn

Descriptions

\[
\begin{align*}
\text{Desc} & ::= \text{somedesc} \ class \ name \ [ \text{DescKeywdArg} \ [* \text{DescTail} \ ]]
\end{align*}
\]

| | | |
| | | |

somedesc ::= there_is_a \ | \ there_is_an \ | \ there_isa \ | \ tisa
| there_is_a_new \ | \ there_isa_new
| there_isa_next \ | \ there_isa_next
| there_isa_bin_for \ | \ there_isa_bin_for
| this_is_a \ | \ this_is_an \ | \ this_isa \ | \ thisa
| there_is_no \ | \ this_is_no

\[
\begin{align*}
\text{DescKeywdArg} & ::= \text{using_no_index} \ | \ \text{using_siz}
\end{align*}
\]

| | | |
| | | |

\[
\begin{align*}
\text{DescTail} & ::= \text{where( DescTailBody )}
\end{align*}
\]

\[
\begin{align*}
\text{DescTailBody} & ::= \ \text{NoteSeq}
\end{align*}
\]
\[
\begin{align*}
\text{NestedDescSeq} & \quad \mid \text{NoteSeq} \quad \text{and} \quad \text{NestedDescSeq} \\
\text{IdNote} & \quad ::= \quad \text{named} \quad \text{Subject} \quad \mid \quad \text{meaning} \quad \text{Subject} \\
\text{NoteSeq} & \quad ::= \quad \text{Note} \quad \downharpoonleft \quad \text{and} \quad \text{Note} \quad \downharpoonright^* \\
\text{Note} & \quad ::= \quad \text{CommonNote} \quad \downharpoonleft \quad \text{but_if_absent} \quad \text{BoundedAsn} \quad \downharpoonright^? \\
\text{CommonNote} & \quad ::= \quad \downharpoonleft \quad \text{noteprep} \quad \downharpoonright^? \quad \text{AttributeExpr} \quad \text{ExtendedPred} \quad \downharpoonleft \quad \text{where} \quad \text{BoundedAsn} \quad \downharpoonright^? \\
& \quad \mid \quad \downharpoonleft \quad \text{noteprep} \quad \downharpoonright^? \quad \text{AttributeExpr} \quad \text{SimplePred} \quad \text{Subject} \quad \text{that} \quad \text{ExtendedPred} \\
& \quad \mid \quad \text{Attribute} \quad \text{Is} \quad \downharpoonleft \quad \_\_\text{absent}\_ \quad \mid \quad \_\_\text{present}\_ \quad \downharpoonright^? \quad \text{where} \quad \text{BoundedAsn} \quad \downharpoonright^? \\
\text{Attribute} & \quad ::= \quad \text{uplow} \\
\text{AttributeExpr} & \quad ::= \quad \text{Attribute} \\
& \quad \mid \quad \text{AttributeExpr} \quad \text{Tuple} \\
& \quad \mid \quad \text{AttributeExpr} \quad \# \quad \text{IntPosition} \\
& \quad \mid \quad \text{AttributeExpr} \quad . \quad \text{StructMbr} \\
\text{where} & \quad ::= \quad \text{where} \quad \mid \quad \text{for\_which} \\
\text{that} & \quad ::= \quad \text{that} \quad \mid \quad \text{that\_is} \\
& \quad \mid \quad \text{which} \quad \mid \quad \text{which\_is} \\
& \quad \mid \quad \text{who} \quad \mid \quad \text{who\_is} \\
& \quad \mid \quad \text{one\_which} \quad \mid \quad \text{one\_which\_is} \\
\text{noteprep} & \quad ::= \quad \text{the} \quad \mid \quad \text{one\_of\_the} \\
\text{NestedDescSeq} & \quad ::= \quad \text{NestedDesc} \quad \downharpoonleft \quad \text{and} \quad \text{NestedDesc} \quad \downharpoonright^* \\
\text{NestedDesc} & \quad ::= \quad \text{somedesc} \quad \text{class} \quad \text{name} \quad \downharpoonleft \quad \text{IdNote} \quad \downharpoonright^? \\
& \quad \mid \quad \downharpoonleft \quad \text{LinkPredPhrase} \quad \downharpoonright^? \quad \text{DescTail} \quad \downharpoonright^? \\
\text{LinkPredPhrase} & \quad ::= \quad \text{that} \quad \text{SimplePred} \\
& \quad \mid \quad \text{that} \quad \text{SimplePred} \quad \text{class} \quad \text{name}
\end{align*}
\]
Predicates

ExtendedPred ::= PredPhrase | ConjPredPhrase | DisjPredPhrase

DisjPredPhrase ::= ConjOr1PredPhrase [ " | " ] ConjOr1PredPhrase I+

ConjOr1PredPhrase ::= PredPhrase | ConjPredPhrase

ConjPredPhrase ::= PredPhrase [ & ] PredPhrase I+

PredPhrase ::= CompoundPred | not CompoundPred

CompoundPred ::= SimplePred [ Subject |? | KeywdArg ]*

SimplePred ::= uplo | Is_In | Equals
   | < | <= | > | >= | ~>

Equals ::= = | Eq | Equals

OpCond ::= ~ [ VblSpecSeq ] such_that BoundedAsn
Subjects: Constants, Variables, Function Calls, and Aggregates

Subject ::= constant | ValCall | FunCall | BoundedAsn
        | . ArrayVbl [ SubjectSeq ] | ( Subject )
;
SubjectSeq ::= Subject [ , Subject ]
;
ListTerm ::= [ SubjectSeq ] | . TupleVbl | . ArrayVbl
;
SomeSubjects ::= Subject | ListTerm
;

Constants

constant ::= integer | float | string | literal
          | date | re | cre | text | blob
          | variable | function | predicate | procedure | class_name
          | lower | uplow | upper | thing
          | skolem | symbolic_constant
;
skolem ::= ?
;
symbolic_constant ::= _no_dot_
          | _last_signal_
          | _all_
          | _true_ | _false_
          | _right_ | _left_
          | _asc_ | _desc_
          | _nbr_of_args_ | _nbr_substi_
          | _start_ | _next_ | _end_
          | _last_ | _prev_ | _to_eof_
          | _stdin_ | _stdout_ | _stderr_
          | _prev_chan_ | _cmd_line_ | _popkorn_
          | _read_ | _write_ | _append_
          | _update_ | _append_update_ | _clean_slate_update_
          | _file_ | _string_ | _text_
          | _pipe_ | _bipipe_ | _fifo_ | _ccp_
;
Variables

\[
\begin{align*}
\text{Variable} & ::= \text{lower} \\
& \quad | \text{Variable} \quad \text{Tuple} \\
& \quad | \text{Variable} \quad \# \quad \text{IntPosition} \\
& \quad | \text{Variable} \quad . \quad \text{StructMbr} \\

\text{ValCall} & ::= [ . \quad ]^* \quad \text{Variable} \\

\text{ValCallSeq} & ::= \text{ValCall} \quad [ , \quad \text{ValCall} \quad ]^* \\

\text{SomeValCalls} & ::= \text{ValCall} \quad | \quad [ \quad \text{ValCallSeq} \quad ] \\
\end{align*}
\]

Function Calls

\[
\begin{align*}
\text{FunCall} & ::= \text{function} \quad ( \quad \text{SubjectSeq} \quad ) \\
& \quad | \quad \text{function} \quad ( \quad [ \quad \text{KeywdArg} \quad ]^* \quad ) \\
& \quad | \quad + \quad \text{Subject} \quad | \quad - \quad \text{Subject} \\
& \quad | \quad \text{Subject} \quad \text{infixop} \quad \text{Subject} \\
& \quad | \quad \text{AggFunCall} \\
\text{infixop} & ::= - \quad | \quad + \quad | \quad * \quad | \quad / \quad | \quad % \quad | \quad <* \\
\text{KeywdArg} & ::= \text{lower} \\
& \quad | \quad \text{preposition} \quad \text{Subject} \quad | \quad \text{preposition} \quad \text{Aggregate} \\
\text{AggFunCall} & ::= \text{count} \quad ( \quad [ \quad \text{distinct} \quad ]^* \quad \text{over} \quad \text{VblSpec} \quad ]^* \\
& \quad | \quad \text{UnaryAggFun} \quad ( \quad [ \quad \text{distinct} \quad ]^* \quad \text{over} \quad \text{VblSpec} \\
& \quad \quad | \quad \text{each_time} \quad \text{BoundedAsn} \quad ) \\
& \quad | \quad \text{quantile} \quad ( \quad [ \quad \text{distinct} \quad ]^* \quad \text{namely} \quad \text{Subject} \quad \text{over} \quad \text{VblSpec} \\
& \quad \quad | \quad \text{each_time} \quad \text{BoundedAsn} \quad ) \\
& \quad | \quad \text{BinaryAggFun} \quad ( \quad \text{over} \quad [ \quad \text{VblSpec} \quad , \quad \text{VblSpec} \quad ] \\
& \quad \quad | \quad \text{each_time} \quad \text{BoundedAsn} \quad ) \\
& \quad | \quad \text{BinaryAggFun} \quad ( \quad \text{over} \quad \text{VblSpec} \quad \text{over} \quad \text{VblSpec} \\
\end{align*}
\]
each_time BoundedAsn )

UnaryAggFun ::= sum | prod | avg | min | max |
| median | stdev | var

BinaryAggFun ::= covar | corr

PartialAggFunCall ::= count ( l distinct p l over VblSpec p ) |
| UnaryAggFun ( l distinct p l over VblSpec ) |
| BinaryAggFun ( over [ VblSpec , VblSpec ] )

PartialAggFunCallSeq ::= PartialAggFunCall l , PartialAggFunCall l *

Aggregates

Aggregate ::= Bunch | Tuple |
| SetFormer | ListFormer |
| box ( for OpCond l BuildBoxKeywdArg p ) |
| . BoxVbl |
| $[ SqlQuery ]$ 

Tuple ::= [ l AggItemSeq p ]

Bunch ::= { l AggItemSeq p }

AggItemSeq ::= AggItem l , AggItem l *

AggItem ::= Subject | IntervalSpec | Tuple

SetFormer ::= { SomeSubjects : Assertion l : BoxQualifiers p }
| { AggItemSeq : : BoxQualifiers }

ListFormer ::= [ SomeSubjects : Assertion l : BoxQualifiers p ]
| [ AggItemSeq : : BoxQualifiers ]
BoxQualifiers ::= [ BuildBoxKeywdArg ]* \\
; 

Interval ::= [ IntervalSpec ] \\
; 

IntervalSpec ::= Subject -> [ Subject ]? ] by Subject ]? \\
;
Types, Definitions, And Imports

Types

Type ::= class_name \[ ( SubclassSpec ) \] P
| Type ARRAY DimSpecTuple
| TUPLE \[ ArgSlotDefSeq \]
| STRUCT \( \) ArgSlotDefSeq \)
| LIST \[ Type \[ BoxQualifiers \] P \]
| SET \{ Type \[ BoxQualifiers \] P \}
| I Type P VBL
| FppType
| Type | Type
| Type & Type
| Type - Type

FppType ::= Type FUN \( \) ArgSlotDefSeq
| PRED \[ ArgSlotDefSeq \]
| PROC \( \) ArgSlotDefSeq

FUN ::= FUN | FUNCTION

PRED ::= PRED | PREDICATE

PROC ::= PROC | PROCEDURE

class_name ::= upper | FLT | INT | STR | DATE
| BOOL | TEXT | BLOB | RE | CRE
| CHAN | THING | LIT | OBJ
| VBL | FUN | PRED | PROC | ARRAY

SubclassSpec ::= integer | * | = | ?
| _long_ | _short_
| _file_ | _string_ | _text_ | _cmd_line_
| _pipe_ | _bipipe_ | _fifo_
| _popkorn_ | _ccp_
Environment Statements

\[\text{EnvStmt} ::= \text{ClassDef} | \text{HelperFppDefs} | \text{ExportedVblDefs} | \text{LocalVblDefs} | \text{Imports} ;\]

\[\text{FppEnvStmt} ::= \text{LocalVblDefs} | \text{Imports} ;\]

\[\text{LocalVblDefs} ::= \text{local} \{ \text{vbladj} \}^* : \{ \text{MultiVblDefs} \}^* ;\]

\[\text{ExportedVblDefs} ::= \text{export} \{ \text{vbladj} \}^* : \{ \text{MultiVblDefs} \}^* ;\]

\[\text{Imports} ::= \text{import} \{ \text{vbl_or_fppadj} \}^* : \{ \text{MultiImps} \}^* ;\]

\[\text{local} ::= \text{local} | \text{locals} ;\]

\[\text{export} ::= \text{export} | \text{exports} ;\]

\[\text{import} ::= \text{import} | \text{imports} ;\]

Definitions

\[\text{ClassDef} ::= \text{define CLASS upper} = \text{with_symbols Bunch} | \text{define CLASS upper} \{ \text{Type} \}^? \text{subclass_of upper} ;\]

\[\text{define} ::= \text{define} | \text{def} ;\]

\[\text{MultiVblDefs} ::= \{ \text{vbladj} \}^* \text{Type}^? \text{VBL} \{ \text{VblBase} \}^? , \text{VblBase} \}^* ;\]

\[\text{VBL} ::= \text{VBL} | \text{VARIABLE} ;\]

\[\text{vbladj} ::= \text{static} | \text{dynamic} | \text{constant} | \text{alias} | \text{copy} | \text{C_external} | \text{C_const} | \text{manifest} ;\]

Copyright © 2013 AT&T
All Rights Reserved.
September 15, 2013
\textbf{VblDBase} ::= \{ . \} \text{lower} \{ DimSpecTuple \} ? \{ \text{Initializer} \} ? ;

\textbf{DimSpecTuple} ::= [ \text{DimSpec} \{ , \text{DimSpec} \} * \{ : \text{ArrayQualifiers} \} ? ] ;

\textbf{DimSpec} ::= \text{integer} | \text{Interval} | [ \text{SubjectSeq} ] | \text{Type}

\textbf{ArrayQualifiers} ::= \{ \text{ArrayKeywordArg} \} *

\textbf{ArrayKeywordArg} ::= \text{with_init_size_in_K} \text{Subject} | \text{with_init_max_nbr_elts} \text{Subject} | \text{with_growth_factor} \text{integer or float} = 2 | \text{with_no_deletions} | \text{with_deletions_ok} | \text{with_default} @ => \text{Subject} ;

\textbf{VblSpec} ::= \{ \text{vbladj} \} * \{ \text{Type} \} ? \{ . \} * \text{lower} \{ \text{DimSpecTuple} \} ? ;

\textbf{VblSpecSeq} ::= \text{VblSpec} \{ , \text{VblSpec} \} *

\textbf{SomeVblSpecs} ::= \text{VblSpec} | [ \text{VblSpecSeq} ]

\textbf{HelperFppDef} ::= \text{FunDef} | \text{PredDef} | \text{ProcDef} |
| \text{define} \{ \text{fppadj} \} * \text{FppType} \text{fpp} \text{Do} \{ ; \} ?
| \text{DeclFppDef} ;

\textbf{fpp} ::= \text{lower} | \text{uplow} ;

\textbf{FunDef} ::= \text{define} \{ \text{fppadj} \} * \text{Type} \text{FUN} \{ : \} ? \text{FunBase} \text{Do} \{ ; \} ? ;

\textbf{fppadj} ::= \text{C_external} | \text{ignoring_side_effects} | \text{overloaded} |
| \text{otherwise_ok} | \text{path} | \text{sighandler} ;

\textbf{FunBase} ::= \text{lower} \{ \} \text{ArgSlotDefSeq} ? ? ;
\[
\text{ArgSlotDefSeq ::= ArgSlotDef} \quad , \quad \text{ArgSlotDef} \quad \ast \\
\text{ArgSlotDef ::= \{ NbrConstraint \} \quad \text{preposition} \quad \text{PRED} \quad \ast \quad \text{fppad} \quad \ast \quad \text{PROC} \quad \text{Do} \quad \ast \quad \text{ProcBase} \\
\text{NbrConstraint ::= ( Multiplicity )} \\
\text{Multiplicity ::= 0 | 1 | 0-1 | 0-> | 1->} \\
\text{Initializer ::= = constant} \quad | \quad = \text{Tuple} \quad | \quad = \text{ValCall} \quad | \quad = \text{FunCall} \\
\text{PredDef ::= define \{ fppad \} \quad \text{PRED} \quad \ast \quad \text{PredBase} \quad \text{Do} \quad \ast \quad \text{ProcBase} \\
\text{PredBase ::= uplow \{ \text{ArgSlotDefSeq} \} \} \\
\text{ProcDef ::= define \{ fppad \} \quad \text{PROC} \quad \ast \quad \text{ProcBase} \quad \text{Do} \quad \ast \quad \text{ProcBase} \\
\text{ProcBase ::= uplow \{ \text{ArgSlotDefSeq} \} \} \\
\text{DeclFppDef ::= MacroPredDef | ViewDef | PathPredDef | OpcoFunDef} \\
\text{MacroPredDef ::= define \{ fppad \} \quad \text{PRED} \quad \ast \quad \text{PredBase} \quad \text{iff} \quad \text{BoundedAsn} \\
\text{using(} \\
\text{outside \{ VblSpecSeq \}} \ast \quad \ast \quad \text{ProcBase} \quad \text{Do} \quad \ast \quad \text{ProcBase} \\
\text{ViewDef ::= define \{ RECORD_CLASS \quad \text{upper} \quad \text{as a view where(} \text{ViewEquivSeq} \} \} \\
\text{alternate_view_spec \{ ViewEquivSeq \} \ast } \\
\text{using(} \\
\text{outside \{ VblSpecSeq \}} \ast \quad \ast \quad \text{ProcBase} \quad \text{Do} \quad \ast \quad \text{ProcBase} \\
\text{opcond_something \{ VblSpecSeq \}} \ast \quad \ast \quad \text{ProcBase} \quad \text{Do} \quad \ast \quad \text{ProcBase} \\
\text{infer_bounds_for \text{thing} \ast } \\
\text{as_read_only \ast \quad \text{has_variant_fields} \ast } \\
\text{all_keyed_record_existence_tests_pass \ast } \\
\text{)} \quad \ast \quad \ast \quad \text{ProcBase} \quad \text{Do} \quad \ast \quad \text{ProcBase} \
\]
ViewEquivSeq ::= ViewEquiv [ and ViewEquiv ]*
<ViewEquiv ::= for_each SomeVblSpecs conclude( Desc iff( Asn ) )
alternate_view_spec ::= with_this_isas_using | with_from_sections_using
| with_deletes_using | with_this_is_no_deletes_using
| with_adds_using | with_updates_using
PathPredDef ::= define [ fppadj ]* path PRED TransVblSpecs [ : P ]
uplow by_stepping_with BoundedAsn
| using(
| outside [ VblSpecSeq ] P
| backtracking_when BoundedAsn P
| stop_finding_children_when BoundedAsn P
| with_distance_vbl INT VBL P
| with_child_nbr_vbl INT VBL P
| with_outcount_vbl INT VBL P
| with_path_vbl INT VBL P
| with_identity P
| given_acyclic P
| HybridBoxKeywdArg P
)
;}

(Task that the backtracking_when and the stopping_when BoundedAsns can use the Candidate_Selected_Before predicate.)

TransVblSpecs ::= [ VblSpec, VblSpec ]
| [ TUPLE [ VblSpecSeq ] , TUPLE [ VblSpecSeq ] ]

OpcoFunDef ::= define [ fppadj ]* Type FUN [ : P ] FunBase = ValCall
iff BoundedAsn [ ; P ]

TaskDef ::= FunTaskDef | PredTaskDef | ProcTaskDef
| define [ fppadj ]* FppType [ taskadj ]* task fpp TaskDo [ ; P ]

Copyright © 2013 AT&T
All Rights Reserved.
September 15, 2013
FunTaskDef ::= define \ fppad \ | \ Type \ | \ FUN \ | \ taskadj \ | \ task \ | : \ task
FunBase \ TaskDo \ | ;

PredTaskDef ::= define \ fppad \ | \ PRED \ | \ taskadj \ | \ task \ | : \ task
PredBase \ TaskDo \ | ;

ProcTaskDef ::= define \ fppad \ | \ PROC \ | \ taskadj \ | \ task \ | : \ task
ProcBase \ TaskDo \ | ;

taskadj ::= begin \ | \ transaction
| flush_on_return \ | \ close_on_return
| on_abort_return \ | \ Subject
| with_logging \ | \ with_no_logging \ | \ with_logging_optional
| with_logging_flag \ | \ Subject
| free_on_begin_when \ | \ Subject \ | \ free_on_return_when \ | \ Subject
;

TaskDo ::= do \ | \ EnvStmt \ | \ CmdSeq \ | \ EnvStmt
;

Imports

MultiImps ::= MultiVblImps \ | \ MultiFunImps \ | \ MultiPredImps \ | \ MultiProcImps
| \ fpp \ | \ FppType \ | \ taskadj \ | \ task \ | :
FunBase \ TaskDo \ | ;

MultiVblImps ::= \ vblad \ | \ Type \ | \ VBL \ | : \ VblBase \ | , \ VblBase
;

VblBase ::= \ lower \ | \ DimSpecTuple \ | ;

MultiFunImps ::= \ fpp \ | \ Type \ | \ FUN \ | \ taskadj \ | \ task \ | : \ task
FunBase \ | , \ FunBase
;

MultiPredImps ::= \ fpp \ | \ PRED \ | \ taskadj \ | \ task \ | : \ task
PredBase \ | , \ PredBase

\[
\text{MultiProclmps} \ ::= \ \{ \text{fppadj} \}^* \text{PROC} \ \{ \text{taskadj} \}^* \text{task} \ \{ \text{ProcBase} \}^* \ \{, \ \text{ProcBase} \}^* \\
\]
\[
\text{package} \ ::= \ \text{package} \ | \ \text{packages} \\
\]
Daytona Data Dictionary Grammar

Daytona supports the use of a rich variety of data modelling constructs. Cymbal descriptions in a Daytona data dictionary specify how these are applied to create any given database. This section presents a formal exposition of the grammar specifying these descriptions.

Syntactic classes here are indicated by underlined italic strings whereas, with the exception of the grammar symbols, the other characters are to be taken literally. The grammar symbols are " ::= " , " ; " , " | " , " _ " , " _ " , " [ " , " ] " , and directed double quotes. " [a] " is read as meaning a , " [aP] " is read as meaning Λ | a where " Λ " denotes the empty string, " [aR+] " is read as meaning Λ | a | a a | . . . , and " [aRP+] " is read as meaning a | a a | . . . .

Grammar non-terminals are represented by underlined UPLOWS as illustrated by RecClassDesc.

Classes of grammar terminals are represented by underlined LOWERS. Here are some special ones: a lower is a string of characters beginning with a lower-case letter and continuing with only lower-case letters, digits, and underscores. An upper is defined similarly. An uplow consists of a sequence of letters, digits, and underscores for which the first letter is upper-case and some subsequent letter is lower-case. A name is a C identifier. An integer is a string of digits. An anything is just a sequence of ASCII characters. A char is a single ASCII character.
Record Class Descriptions

RecClassDesc ::= #{ RECORD_CLASS ( upper ) l rc_role l2
  l <Multiplicity, multiplicity r l2
  l <Fio_Dir, directory r l2
  l <Version, anything r l2
  l ClassifyingFieldsDesc r l2
  l BinsDesc r l2
  l KeysDesc r l2
FieldsDesc
  l RecClassDesc r l* }#

rc_role ::= Is_A_Partition l Is_A_Subclass

multiplicity ::= 1 l | 0->1 l | 0-> l | 1->

ClassifyingFieldsDesc ::= #{ CLASSIFYING_FIELDS
  l #{ FIELD ( uplow ) <Value, anything> }# l* }#

BinsDesc ::= #{ BINS
  l <Max_Open_Bins, integer r l2
  l <Max_Open_Seq_Acc_Bins, integer r l2
  l <Max_Open_Random_Acc_Bins, integer r l2
  l <Dynamic, yesorno r l2
  l <Indices_Banned_For_Fif, yesorno r l2
  l <Using_Source, anything r l2
  l <Default_Data_File_Source, path r l2
  l <Default_Data_File_Indices_Source, path r l2
  l <Default_Rec_Map_Spec_File, anything r l2
  l <Default_Seq_Acc_Bufsize, integer r l2
  l <Default_Random_Acc_Bufsize, integer r l2
  l <Default_Reuse_Freed_Space, yesorno r l2
  l <Default_Pad_To_Len, integer r l2
  l <Default_Read_Only, yesorno r l2
  l <Enable_Using_Source_Txns, yesorno r l2

Copyright © 2013 AT&T
All Rights Reserved.
September 15, 2013
{ FileDesc 1* 
  { FileInfoFileDesc 1* 
  }# 
  ; 
  yesorno ::= yes | no 
  ; 

FileDesc ::= #{ FILE ( name ) 
  { <Unit_Sep, char> 1? 
  { <Unit_Seps, chars> 1? 
  <Comment_Beg, char> 
  { <Tuple_Delims, chars> 1? 
  <Source, anything> 1? 
  { <Indices_Source, anything> 1? 
  { <Rec_Map_Spec_File, anything> 1? 
  { <Seq_Acc_Bufsize, integer> 1? 
  { <Random_Acc_Bufsize, integer> 1? 
  { <Reuse_Freed_Space, yesorno> 1? 
  { <Pad_To_Len, integer> 1? 
  { <Read_Only, yesorno> 1? 
  { <Data_Interface, upper> 1? 
  <PartitioningFieldsDesc 1? 
  }# 
  ; 

PartitioningFieldsDesc ::= #{ PARTITIONING_FIELDS 
  { #{ FIELD ( uplow ) <Value, anything> }# 1* 
  }# 
  ; 

FileInfoFileDesc ::= #{ FILE_INFO_FILE ( name ) 
  { <Unit_Sep, char> 1? 
  { <Unit_Seps, chars> 1? 
  <Comment_Beg, char> 
  <Tuple_Delims, chars> 
  { <Source, anything> 1? 
  { <Version, anything> 1? 
  { <Pad_To_Len, integer> 1? 
  ; 

Copyright © 2013 AT&T
All Rights Reserved.
September 15, 2013
Can_Grow_Stats_By_Pct, yesorno

Default_Data_File_Source, yesorno

Default_Data_File_Indices_Source, integer

Default_Rec_Map_Spec_File, anything

Default_Seq_Acc_Bufsize, integer

Default_Random_Acc_Bufsize, integer

Default_Reuse_Freed_Space, yesorno

Default_Pad_To.Len, integer

Default_Read_Only, yesorno

#( FILE ( __DUMMY )

Unit_Sep, char

Unit_Seps, chars

Comment_Beg, char

Tuple_Delims, chars

Source, __DUMMY

Indices_Source, __DUMMY

Rec_Map_Spec_File, anything

Seq_Acc_Bufsize, integer

Random_Acc_Bufsize, integer

Reuse_Freed_Space, yesorno

Pad_To.Len, integer

Read_Only, yesorno

Data_Interface, upper

#( PARTITIONING_FIELDS

#( FIELD ( uplow ) <Value, __DUMMY> )

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)
KeyDesc ::= #{ KEY ( name )
  [ Field, name ]| |
  [ Fields, [ name | , name |* ] > | ]
  [ Unique, yesorno ]| |
  [ IndicesDesc | ]
}

IndicesDesc ::= #{ INDICES
  [ # ( INDEX ( name ) <Kind, indexkind>
    [ Unique, yesorno ]| |
    [ Ignore_DFR_Missing_Values, yesorno ]| |
    ) # | ]+
}

indexkind ::= btree | cluster_btree

FieldsDesc ::= #{ FIELDS
  [ More_Fields_Follow, yesorno ]| |
  [ # ( FIELD ( name ) partorclassrole <Type, FieldType>
    [ Value_Expr, anything ]| |
    ) # | ]+
  [ # ( FIELD ( name ) <Type, FieldType>
      [ Value_Expr, anything ]| |
      [ Default_Value, anything ]| |
      [ Validation_RE, anything ]| |
      [ Min_Value, anything ]| |
      [ Max_Value, anything ]| |
      [ Input_Filter_Fun, lower ]| |
      [ Output_Filter_Fun, lower ]| |
    ) # | ]+
}

partorclassrole ::= Is_A_Partitioner | Is_A_Classifier

FieldType ::= [ FieldMultiplicity cymbal_type | ]

Copyright © 2013 AT&T
All Rights Reserved.
September 15, 2013
FieldMultiplicity ::= ( 0 ) | ( 1 ) | ( 0->1 )
Application Descriptions
AppDesc ::= "# { APPLICATION ( name )
  <Version, anything >
  <Default_Data_File_Source, anything >
  <Default_Data_File_Indices_Source, anything >
  <Default_Fio_Dir, directory >
  <Default_Seq_Acc_Bufsize, integer >
  <Default_Random_Acc_Bufsize, integer >
  <Default_Reuse_Freed_Space, yesorno >
  <Default_Read_Only, yesorno >
  <Default_Data_Interface, upper >" } #

MakeGoodsDesc ::= "# { MAKE_GOODS
  <Make_CC, anything >
  <Make_AR, anything >
  <Make_LD, anything >
  <Cpp_Path, anything >" # { FILES
  # { FILE ( name )
    <Source, anything >" } #

  # { LIBRARY ( name )
    <In_Package, anything >
    <Source, anything >" } #

  FileBaseDesc" } #

FileBaseDesc ::= "# { FILE_BASE ( name )
  <Depends_On, [ anything ] , anything ]* }" } #
Project Descriptions

ProjDesc ::= # ( PROJECT ( name )
    <Apps, [ name _ , name ] >
    <Default_Fio_Dir, directory >
    <Txn_Log_Dir, anything >
    <Max_Do_Queue_In_K, integer >
    <Max_Log_File_In_K, integer >
    <Query_Preprocessor_Invocation, anything >
    MakeGoodsDesc )
) #
Daytona Man Pages

This appendix consists of a collection of man pages for selected Daytona commands. A subgroup of these man pages deals with Daytona’s DC data format and various data filters that transform it. Man pages for Archie, Synop, Sizup, Tracy and others are also given. These pages are also available on-line by means of the DS Man command.
NAME
intro - introduction to Daytona commands

DESCRIPTION
This section describes the principal Daytona commands in alphabetic order. This introduction provides general information about Daytona, its commands, and the environment in which they operate.

General Introduction
The Daytona system translates user programs (called "queries", though these programs may manipulate data as well as access it) into object code or standalone executables, which obey certain protocols that enable them to run concurrently without clobbering the data (or each other). For the application programmer or end user, Daytona presents an abstract view of the data as a set of tables (basically the relational model, with certain extensions) and a very-high-level language called Cymbal for writing queries. Cymbal contains the standard query language SQL as a subset. For the database administrator, Daytona presents a simple picture: data resides in ordinary flat UNIX files; data files belong to the application, and can be placed wherever the application finds convenient; there is no DBMS server (Daytona programs use the operating system as a server).

Daytona compiled queries are generally very fast (typically several times faster than equivalent code built with Oracle or Informix). This is primarily due to Daytona's strategy of extracting all information about the data and the query at compile time. If the data is redefined (say by adding additional fields to each record) then all existing queries that refer to the data must be recompiled by the end user or application developer. But for a great many applications this is a worthwhile tradeoff.

At compile time, Daytona examines not the data itself, but the data dictionary, which defines data's file structure and integrity constraints. It also defines what indices are to exist for the data, and supplies implementation information such as the names and locations of the files containing the data. The data dictionary also typically contains statistics gathered from the data, which Daytona sometimes uses at compile time in determining the data access strategy to be used at runtime. The data administration tool Sizup also uses the data dictionary. Compiled queries, however, do not: as mentioned above, all necessary data dictionary information is extracted at compile time, and is reflected in the C code that Daytona generates to implement the query. (A recent exception to this is dynamic horizontal partitioning.)

Locating Data Dictionary Information
Daytona tables (also called record classes) are grouped into applications, which in turn are grouped into projects. The corresponding data dictionary elements are called record class descriptions (rcds), application descriptions (apds) and project descriptions (pjd). There can be several applications active in the user's environment, but at most one project. If there is a project, the active applications are expected to be among those belonging to the project. Certain default information relevant to an individual record class (such as the pathname of the directory that contains the corresponding data) may be specified at the application level. Certain information relevant to processing transactions on the data (such as the pathname of the directory that contains the transaction logs) is only specified at the project level. Therefore, though the most commonly needed information about a table is present in the corresponding rcd, the user may on occasion need to consult the corresponding application or project description as well. The apd for an application app and theapds for its associated record classes are collected in an application archive file app. The pjds for a project proj is embedded (as the sole member) in a project archive file proj. By default, Daytona tools set their current application and project environment from the shell variables APPS, PROJ, and PATH. APPS is a colon-separated list of application names, PROJ is a single project name (or is empty), and PATH is a colon-separated list of directories. These directories are the ones Daytona searches to locate the corresponding application archives and project archives.

Initializing the Environment
Daytona provides a utility called Set for initializing the shell environment. This should be run as a dot script from each user's login shell. Daytona utilities and compiled queries rely on this script having been run. Set sets certain shell variables appropriate for the particular site and platform type, and
User Commands

modifies the user’s PATH variable, if necessary, to include the Daytona installation directory. **DS Set** also checks that the applications listed in **DS APPS** all belong to the project **DS PROJ**, if the latter is nonempty. **DS Set** should be rerun when these variables are changed. **DS Env** will print out the state of Daytona’s shell environment.

**LIST OF COMMANDS**

By just typing in the command **DS**, one can see the current list of Daytona shell commands:

% **DS**

commands to try:

(to get usage help for any command, invoke with a -? or +? option, whichever works)

(commands invoked using DS with no arguments usually offer interactive prompting)

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DS Set</strong></td>
<td>sets the Daytona shell environment</td>
</tr>
<tr>
<td><strong>DS Env [vbl]</strong></td>
<td>displays the Daytona shell environment</td>
</tr>
<tr>
<td><strong>Dice</strong></td>
<td>the EASEL menu/screen interface to Daytona</td>
</tr>
<tr>
<td><strong>Daisy</strong></td>
<td>the Daytona interactive SQL DML and DDL processor</td>
</tr>
<tr>
<td><strong>[DS] QQ</strong></td>
<td>quickly edits &amp; runs queries; few options</td>
</tr>
<tr>
<td><strong>DS Expr</strong></td>
<td>quickly evaluates a Cymbal term</td>
</tr>
<tr>
<td><strong>[DS] Tracy/Stacy</strong></td>
<td>translates Cymbal/SQL into C and calls DS Exec; many options</td>
</tr>
<tr>
<td><strong>DS Exec</strong></td>
<td>makes and/or runs executables from C; many options</td>
</tr>
<tr>
<td><strong>DS Compile</strong></td>
<td>just compiles queries to executables; no options</td>
</tr>
<tr>
<td><strong>DS Mk</strong></td>
<td>&quot;DS Mk &lt;name&gt;&quot; makes executable &lt;name&gt; from DS C code</td>
</tr>
<tr>
<td><strong>DS Edit</strong></td>
<td>supports editing rcd.*, *Q, *.S, etc.</td>
</tr>
<tr>
<td><strong>DS Vi/Vu</strong></td>
<td>supports vi editing/reading rcd.*, *Q, *.S, etc.</td>
</tr>
<tr>
<td><strong>DS Emacs/Emucs</strong></td>
<td>supports emacs editing/reading rcd.*, *Q, *.S, etc.</td>
</tr>
<tr>
<td><strong>DS Joe/Jou</strong></td>
<td>supports joe editing/reading rcd.*, *Q, *.S, etc.</td>
</tr>
<tr>
<td><strong>[DS] DC-rcd</strong></td>
<td>creates a record class description (rcd) for data</td>
</tr>
<tr>
<td><strong>[DS] Synop</strong></td>
<td>provides synopses of application and project archives</td>
</tr>
<tr>
<td><strong>[DS] Synop_fpp</strong></td>
<td>provides synopses of functions/predicates/procedures</td>
</tr>
<tr>
<td><strong>find-it tools:</strong></td>
<td>ar_of, ar_or_env_fl_for, env_fl_for_fpp, ds_whence</td>
</tr>
<tr>
<td><strong>[DS] Archie</strong></td>
<td>performs archive maintenance</td>
</tr>
<tr>
<td><strong>DS Show</strong></td>
<td>displays phys/virt tables, data files or executable output filters:</td>
</tr>
<tr>
<td></td>
<td>DC-pkt DC-prn</td>
</tr>
<tr>
<td><strong>DS Mk_DE</strong></td>
<td>makes data entry programs from rcds</td>
</tr>
<tr>
<td><strong>[DS] Sizup</strong></td>
<td>validates and builds indices for data</td>
</tr>
<tr>
<td><strong>DS Msgmrg</strong></td>
<td>merges Sizup.msgs with data</td>
</tr>
<tr>
<td>file tools:</td>
<td>Get_Lines, Delete_Faults, Trunc, Split</td>
</tr>
<tr>
<td><strong>find lockers:</strong></td>
<td>Lock_Blockers_For_File_Path, Lock_Blockers_For_Shmem</td>
</tr>
<tr>
<td><strong>IPC info:</strong></td>
<td>Show_Shmem, Rm_Shmem, Describe_Shmem, Show_Sem, Rm_Sem, Describe_Semid</td>
</tr>
<tr>
<td><strong>[DS] Check_DC_Lines</strong></td>
<td>does a quick sanity check on a DC data file</td>
</tr>
<tr>
<td><strong>[DS] Check_Indices</strong></td>
<td>cross checks indices against their data for a RECORD_CLASS</td>
</tr>
<tr>
<td><strong>[DS] Checkups</strong></td>
<td>checks that the metadata (pjd/apd/rcd) is consistent with what it describes</td>
</tr>
<tr>
<td><strong>DS Resync</strong></td>
<td>regenerates RECORD_CLASS I/O files and other objects and/or indices</td>
</tr>
<tr>
<td><strong>DS Relocate</strong></td>
<td>relocates System_Generated pjd/apd Source values to new $DS_PATH</td>
</tr>
<tr>
<td><strong>DS Clean</strong></td>
<td>removes RECORD_CLASS I/O files and other objects and/or indices</td>
</tr>
<tr>
<td><strong>DS Clean.Misc</strong></td>
<td>removes miscellaneous DS-generated files</td>
</tr>
<tr>
<td><strong>DS Basics</strong></td>
<td>the Daytona Basics book presented via acroread</td>
</tr>
<tr>
<td><strong>DS All_About</strong></td>
<td>the All About Daytona reference manual presented via acroread</td>
</tr>
<tr>
<td><strong>DS Tutorial</strong></td>
<td>the Hands-On Tutorial presented via acroread</td>
</tr>
<tr>
<td><strong>DS Man</strong></td>
<td>DS Man lists topics; DS Man &lt;topic&gt; provides the associated man page</td>
</tr>
<tr>
<td><strong>DS White_Paper</strong></td>
<td>updated SIGMOD conference paper presented via acroread</td>
</tr>
<tr>
<td><strong>DS Course</strong></td>
<td>the Daytona course presented via acroread</td>
</tr>
<tr>
<td><strong>DS Doc</strong></td>
<td>cats the nroff form of All About Daytona into $PAGER</td>
</tr>
</tbody>
</table>
DS M4 runs DS-modified m4 preprocessor on Cymbal/SQL queries

[DS] Reducyr/Squirrel parses Cymbal/SQL queries

recovery: Recover, Clean_Logs, dump_log

compression: Find_Dict.1, Cmpl_Dict.1, Eval_Dicts, Dict_Map_Rec

Census Census [ <siz_file_paths_RE> ]* computes statistics for all given data files

Stat_Proc Stat_Proc <pid> displays process status info

Distribute_Cmds parallelize a file of commands by job distribution

psme shows all processes for this user

Sleep_Time e.g., Sleep_Time 1.5s

licmanu shows manufacturing date for the license file in use

DS Logos displays the Daytona, Cymbal, and backtalk logos using xv

Man pages for many of these follow.
NAME
CSV – Comma Separated Value file format

DESCRIPTION
This file format is used as a common language by a family of filters that translate to and from various more specialized formats (Daytona’s DC, Lotus, dBase II and III, etc) as well as other common UNIX formats (name-value pairs, etc). It is a variation of the UNIX flat file. Additional commands exist to manipulate data in this format (e.g. swap rows and columns, subset or reposition blocks of data, etc).

The format consists of newline-terminated records, with data values separated into fields by a delimiter (by default a comma ','). Field names may be indicated a header record. Programs that read or write CSV have a command line switch that indicates whether such a header is present.

Standard CSV uses the field delimiter as a separator (between fields only). A variant form of CSV uses the field delimiter as a terminator (after every field, including the last field). Programs that read or write CSV have a command line switch that indicates which variant to assume.

Leading and trailing spaces and tabs are ignored in fields. String fields are enclosed in double quotes. The enclosing quotes are stripped when the field is read. Quotes and the delimiter character may occur inside the enclosing quotes, and do not need to be escaped; they are read as ordinary text. Dates are considered strings, and need to be quoted.

EXAMPLES

"Name" | "Age" | "Spouse" | "Child" | "Cars"
"Jed" | 55 | "" | "{Billie Jo|Betty Jo|Bobbie Jo}" | "{Rambler}" 
"Lucas" | 22 | "" | "" | "{Yugo}
"Jock" | 65 | "Miss Ellie" | "{JR|Gary|Bobby}" | "{Mercedes|Mercedes}"

SEE ALSO
Csv-csv(1), csv-Csv(1), Csv-prn(1), Csv-pkt(1), Csv-rcd(1).
NAME
DC – Daytona data file format

DESCRIPTION
This is the file format used by Daytona to store data. It is also used by Daytona Display and SQL queries for output. It is a variation of the UNIX flat file.

The actual data items (field values) are embedded in delimited records. The record and field delimiters and six other syntactic characters are reserved for indicating the structure of the records, and are restricted partially or entirely from appearing in the field values.

<table>
<thead>
<tr>
<th>name</th>
<th>function</th>
<th>default value</th>
<th>redefinable</th>
</tr>
</thead>
<tbody>
<tr>
<td>CM</td>
<td>comment</td>
<td>%</td>
<td>y</td>
</tr>
<tr>
<td>FS</td>
<td>field separator</td>
<td>/</td>
<td>y</td>
</tr>
<tr>
<td>ES</td>
<td>element separator</td>
<td>FS</td>
<td>y</td>
</tr>
<tr>
<td>BB</td>
<td>tuple begin</td>
<td>[</td>
<td>y</td>
</tr>
<tr>
<td>BE</td>
<td>tuple end</td>
<td>]</td>
<td>y</td>
</tr>
<tr>
<td>DL</td>
<td>delete marker</td>
<td>^^</td>
<td>n</td>
</tr>
<tr>
<td>CN</td>
<td>continuation</td>
<td>\</td>
<td>n</td>
</tr>
<tr>
<td>RS</td>
<td>record separator</td>
<td>NEWLINE</td>
<td>n</td>
</tr>
</tbody>
</table>

CN cannot be the last byte of the last field. DL cannot be the first byte of the first field. RS, FS, and ES cannot appear in the data fields at all. BB, and BE cannot appear in data fields if there are TUPLE-valued FIELDS defined.

Data is grouped into records, which are terminated by the NEWLINE character RS. A line ending with CN however causes the record to continue to the next line.

Records are of three types: comment records, freed records, and active records. Records consisting of a single new-line character, and records beginning with CM, are comment records; records beginning with DL are freed records; all other records are active records.

Comment records and freed records are not regarded as containing data. Comment records are used for annotating the data or improving readability, or for passing information to the Daytona filters. Freed records arise in practice from delete or update operations. Daytona insert or update queries by default regard freed records as available space for new or relocated data.

Active records consist of one or more fields, separated by FS. All active records in a data file are expected to have the same number of fields.

Active records may contain embedded comments. A comment within an active record begins with CM and extends to the next FS, ES, BE, or RS. Active records may, like other records, contain newlines escaped by CN. However, escaped newlines are only allowed immediately before FS, ES, CM, or RS. The escaped newline is transparent to Daytona, and in particular is not considered to be part of the field value.

Fields can store one value or a sequence of values. A value sequence begins with BB, contains zero or more values separated by ES, and ends with BE. Daytona uses value sequences to store LIST-valued fields. The filter DC-csv(1) has an option to convert data containing value sequences to normalized form.

EXAMPLES
\%msg)f1ds)Name|Age|Spouse|Child|Cars
Jed|55|[Billie Jo|Betty Jo|Bobbie Jo]|(Rambler|
Lucas|22%maybe|[]|[Yugo]
Jock|65|Miss Ellie|[JR|Gary|Bobby]|(Mercedes|Mercedes|

SEE ALSO
DC-csv(1), csv-DC(1), DC-prn(1), DC-pkt(1), DC-rcd(1).
NAME
csv-DC – convert a CSV (Comma-Separated Value) format file to DC (Daytona data format)

SYNOPSIS
csv-DC [ -ddelim ] [ -Ccom ] [ -h ] [ -n ] [ -t ] [ files ]

DESCRIPTION
csv-DC converts files (or standard input) in the CSV format (see CSV(5)) into Daytona DC data format
(see DC(5)) which is written on the standard output.

Leading and trailing whitespace and enclosing quotes are stripped from fields. The field name header, if
present, is converted to a DC msg)flds) comment record. The trailing delimiter, if present, is
removed.

The output can be redirected into a file for use by another program. If multiple input files are specified,
all are translated into a single PRN stream.

OPTIONS
-ddelim
Specify the field delimiter in the input as delim; the default is a pipe ‘\|’. A tab may be
specified either by a literal tab character or by the C escape sequence ‘\t’. (This applies to the
following options as well.) If the multivalued field separator VS in the input is different from
FS, it should be specified by appending it to the argument to the –d option, as in the second
example below.

-Ccom Specify the comment character CM in the output as com; the default is ‘%’.
-h Interpret the first record as a field name header, and convert it to a msg)flds) comment record in the output.
-n Assume the input does not have a trailing delimiter after the last field in each record. This is
the default.
-t Assume the input has a trailing delimiter after the last field in each record. This delimiter will
be discarded on output.

EXAMPLES
$ DC-csv -d'\t' -C'#' -t'[]' $ORDERS_HOME/SUPP > SUPP.csv
$ csv-DC -d'\t' -C'#' -t $ORDERS_HOME/SUPP.csv > SUPP.DC

SEE ALSO
DC-red(1), DC-prn(1), DC-pkt(1), DC-csv(1).
NAME
DC-csv – convert a DC (Daytona data format) file to CSV (Comma-Separated Value) format

SYNOPSIS
DC-csv [ -ddelim ] [ -Ccom ] [ -ttuple_delims ] [ -h ] [ -N ] [ -n ] [ -T ] [ files ]

DESCRIPTION
DC-csv converts files (or standard input) in the Daytona DC data format (see DC(5)) into CSV format (see CSV(5)) which is written on the standard output.

Embedded comments are stripped from data records, and continued newlines are removed. Comment records are ignored, except for a msg flds comment record when the -h option is used. String fields, date fields, and TUPLE-valued fields are quoted on output. Free data records (representing deleted or relocated data) are removed.

The output can be redirected into a file for use by another program. If multiple input files are specified, all are translated into a single PRN stream.

OPTIONS
-ddelim
Specify the field delimiter FS in the input as delim; the default is a pipe ‘|’. A tab may be specified either by a literal tab character or by the C escape sequence ‘\t’. (This applies to the following options as well.) If the TUPLE-valued field separator ES in the input is different from FS, it should be specified by appending it to the argument to the -d option, as in the second example below.

-Ccom
Specify the comment character CM in the input as com; the default is ‘%’.

-tuple_delims
Specify the TUPLE delimiter characters BB, BE in the input as tuple_delims; the default is ‘[]’.

-h
Print a field name header as the first output record. The field names are taken from a msg flds comment record in the input, which must have the same number of fields as the data records and must precede the data records.

-N
Normalize records containing TUPLE-valued fields (i.e. emit multiple output records, one for each choice of one value for each TUPLE-valued field).

-n
Do not print a delimiter after the last field in each output record. This is the default.

-T
Print a trailing delimiter after the last field in each output record.

EXAMPLES
Pretty-print the file SUPP from the Daytona sample database:
$ DC-csv -d’\t’ -C’#’ -t’[]’ $ORDERS_HOME/d/SUPP > SUPP.prn

Normalize the file PERSONI from the Daytona sample database:
$ DC-csv -d’:\t’ -C’#’ -t’[]’ -N $ORDERS_HOME/d/PERSOINI > PERSONI.norm

SEE ALSO
DC-red(1), DC-prn(1), DC-pkt(1), csv-DC(1), DC(5), CSV(5).
NAME
DC-rcd – generate a record class description from a DC (Daytona data format) file

SYNOPSIS
DC-rcd [ -d delim ] [ -C com ] [ -t tuple_delims ] [ -S Source ] [ -I Indices_Source ] [ -r recls ] [ -f ] [ -p ] [ -H ] [ -T ] [ -i y | n ] [ -V ver ] [ file ]

DESCRIPTION
DC-rcd reads a file (or standard input) in the Daytona DC data format (see DC(5)) and writes a corresponding record class description (rcd) to a file called rcd.<record_class_name> (or stdout with the -p option). Typically, this record class description is just a first approximation to the final one the user will want to use and so, the user will frequently wish to edit this rcd to add new information (such as additional key information and validation checks).

A comment record beginning with CM(msg)flds) is interpreted as defining the field names for the data. Such a record, if present, must contain the same number of fields as the data records, and must precede the data records.

All other comment records, including empty lines, are ignored. (see -T for an exception)

OPTIONS
-d delim
Specify the field delimiter FS in the input as delim; the default is a pipe ‘|’. A tab may be specified either by a literal tab character or by the C escape sequence ‘\t’. (This applies to the following options as well.) If the TUPLE-valued field separator ES in the input is different from FS, it should be specified by appending it to the argument to the -d option, as in the second example below.

-C com
Specify the comment character CM in the input as com; the default is ‘%’.

-t tuple_delims
Specify the TUPLE delimiter characters BB, BE in the input as tuple_delims; the default is ‘[]’.

-S Source
Use the string Source to define a shell expression for a directory containing the data file. Compiled queries will evaluate this expression at runtime to locate the data file. For pipe _ FILES, the Source is the shell command creating the pipe.

-I Indices_Source
Use the string Indices_Source to define a shell expression for a directory containing the indices for the data file. Compiled queries will evaluate this expression at runtime to locate the indices for the data file.

-f
Define upper bounds for lengths of string fields, based on the sample data.

-p
Print the rcd to stdout instead of the file rcd.<record_class_name> . The output can then be redirected into a file for use by another program.

-H
Insert rcd sub-templates for horizontal partitioning in the output rcd.

-r recls
Use recls as the name for the record class in the output rcd.

-T
Use the comment record beginning with CM(msg)types) in the input file to determine the field types in the output rcd instead of scanning the entire data file.

-i y | n
Create a btree INDEX specification in the output rcd if and only if argument is ‘y’, which is the default.

-V ver
Use ver as the version for the record class in the output rcd.

EXAMPLES
Generate an rcd to stdout using the file SUPP from the Daytona sample database and save it in rcd.SUPPLIER: $ DC-rcd -d \t -C # -p -r SUPPLIER $ORDERS_HOME/SUPP > rcd.SUPPLIER
Generate an rcd in the current directory using the file PERSONI from the Daytona sample database:

$ DC-rcd -d:\t-\t-C\#-t[ ]$ORDERS_HOME/PERSONI

Generate an rcd in the current directory that includes sub-templates for horizontal partitioning:

$ DC-rcd -H -r HP_TABLE data1

SEE ALSO

DC-prn(1), DC-pkt(1), DC-csv(1), csv-DC(1), DC(5).
NAME
DC-pkt – convert a DC (Daytona data format) file to PKT (PacKeT) format

SYNOPSIS
DC-pkt [ -d delim ] [ -C com ] [ -tuple delims ] [ -s sopen ] [ -S ] [ -T ] [ -X ] [ -r rec_hdr ] [ -z ] [ -q ] [ files ]

DESCRIPTION
DC-pkt converts files (or standard input) in the Daytona DC data format (see DC(5)) into PKT format (name-value pairs separated by a separator string), which is written on the standard output. The output may be redirected into a file or pipe for use by another program. If multiple input files are specified, all are translated into a single PKT stream.

Let CM indicate the chosen comment character. A comment record beginning with CM msg)flds) is interpreted as defining the field names for the data. Such a record, if present, must contain the same number of fields as the data records, and must precede the data records.

A comment record beginning with CM msg)recls) is interpreted as a record header to use instead of the default ‘PACKET %d’. However, the command line -rrec_hdr (see below) in turn overrides this comment record.

All other comment records, including blank lines, are ignored.

OPTIONS
- d delim
Specify the field delimiter FS in the input as delim; the default is a pipe ‘|’. A tab may be specified either by a literal tab character or by the C escape sequence ‘\t’. (This applies to the following options as well.) If the TUPLE-valued field separator ES in the input is different from FS, it should be specified by appending it to the argument to the -d option, as in the second example below.

- C com
Specify the comment character CM in the input as com; the default is ‘%’.

- tuple delims
Specify the TUPLE delimiter characters BB, BE in the input as tuple delims; the default is ‘[‘].

- s sopen
Use the string sopen to separate name from value in the name-value pair section of the output. The default separator string is ‘:’.

- S
Use the delimiters and comment character produced by ‘_safe_’ format. This overrides any use of the other delimiter options (‘-d’, ‘-C’, ‘-T’).

- T
Use Cymbal description format for the output, instead of packets.

- X
Use XML format for the output, instead of packets.

- rrec_hdr
Group the name-value pairs into ‘‘packets’’ in the PKT file using the string rec_hdr. The default label or record header is ‘PACKET %d’, where %d expands into a packet number, (i.e., 1, 2, 3, etc.).

- z
Fill blank and null fields with zero on output.

- q
Put double quotes around strings in the output packets.

DC-pkt will handle files using the multi-valued field format employed by Daytona. A TUPLE-valued field consists of the TUPLE start character, by default ‘[‘, followed by zero or more values separated by the TUPLE-valued field separator, and ends with the TUPLE end character, by default ‘]’.

EXAMPLES
Print the file PART from the Daytona sample database in Cymbal description format:

$ DC-pkt -d’ ’ -C’ ’ -T $ORDERS_HOME/d/PART
Print the file PERSONI with TUPLE-valued FIELDS from the Daytona sample database to the file PERSONI.pkt in packet format:

```
$ DC-pkt -d:\t -C'#' -t[']' ORDERS_HOME/d/PERSONI > PERSONI.pkt
```

Convert the DC file \textit{xyz} to the packet file \textit{xyz.pkt}, with packets defined by `"##### "' followed by the packet number, and the name-value pairs separated by tabs:

```
$ DC-pkt -r '##### %d' -s '\t' xyz > xyz.pkt
```

SEE ALSO

DC-red(1), DC-prn(1), DC-csv(1), csv-DC(1), DC(5).
NAME

DC-prn – convert a DC (Daytona data format) file to PRN (PRiNt) format

SYNOPSIS

DC-prn [ -ddelim ] [ -Ccomch ] [ -ttuple_delims ] [ -sosep ] [ -q ] [ -c ] [ -L ] [ -S ] [ -T ] [ files ]

DESCRIPTION

DC-prn converts files (or standard input) in the Daytona DC data format (see DC(5)) into PRN format (equally-spaced fixed-width fields), which is written on the standard output. The width of each “column” is determined by the longest field entry for that column. Input fields must be either numbers, dates, or ordinary strings. Numeric fields are aligned with respect to the (possibly implicit) decimal point.

A comment record beginning (after the CM) with msg)flds) is interpreted as defining the field names for the data. Such a record, if present, must contain the same number of fields as the data records, and must precede the data records.

Other comment records beginning (after the CM) with msg) are passed through to the output, after stripping off the characters up through the first ‘)’. Such records will be split into fields for output using the FS delimiter. They are allowed to have as many or fewer fields than the data records, and comments with fewer fields may have oversize fields, which are printed “as is,” and the output column separator is changed to blank for the rest of the line. Only records having the maximum number of fields are considered in determining the output field widths. This feature can be used to put titles on reports, as in the third example below.

All other comment records, including empty lines, are ignored.

The output can be redirected into a file for use by another program. If multiple input files are specified, all are translated into a single PRN stream.

OPTIONS

-ddelim

Specify the field delimiter FS in the input as delim; the default is a pipe ‘|’. A tab may be specified either by a literal tab character or by the C escape sequence ‘\t’. (This applies to the following options as well.) If the TUPLE-valued field separator ES in the input is different from FS, it should be specified by appending it to the argument to the -d option, as in the second example below.

-Ccomchar

Specify the comment character CM in the input as comchar; the default is ‘%’.

-ttuple_delims

Specify the TUPLE delimiter characters BB, BE in the input as tuple_delims; the default is ‘[]’.

-sosep

Use the string sosep to separate the fixed-width output columns; the default is two spaces. A null separator may be entered as NULL. This is useful to construct fixed width (IBM style) data files of minimum width.

-q

Put double quotes around strings in the output.

-c

Write “#columns” information as an output header. This option automatically creates a file of columnated data, and includes the information needed for later operations using col-csv(1L), for example. The -c option is frequently used with -s NULL. See csv-col(1L) for more information.

-L

Put dashed lines in the output, before and after the column header and at the end of the data.

-S

Use the delimiters and comment character produced by ‘_safe_’ format. This overrides any use of the other delimiter options (‘-d’, ‘-C’, ‘-t’).

-T

Print the column separator (or trailer) after every output column, including the last one. The
default is to print the column separator only between columns.

EXAMPLES
Pretty-print the file SUPP from the Daytona sample database:
$ DC-prn -d'\t' -C'#' -t'[]' $ORDERS_HOME/d/SUPP > SUPP.prn

Pretty-print the file PERSONI from the Daytona sample database:
$ DC-prn -d':\t' -C'#' -t'[]' $ORDERS_HOME/d/PERSONI > PERSONI.prn

Generate a quick report by adding a title:
$ { echo "msg)|Profitability Report"; cat xyz ;} | DC-prn
When xyz is a file like
|msg|flds)Product|Sales|Cost|Profit
Paper Goods|4569.00|2312.00|2257.00
Glass|8568.00|5734.00|2834.00

this yields
<table>
<thead>
<tr>
<th>Product</th>
<th>Sales</th>
<th>Cost</th>
<th>Profit</th>
</tr>
</thead>
</table>
|Paper Goods| 4569.00| 2312.00| 2257.00
|Glass    | 8568.00| 5734.00| 2834.00

SEE ALSO
DC-red(1), DC-pkt(1), DC-csv(1), csv-DC(1), DC(5).
User Commands

NAME
Check_DC_Lines – Checks to see if a file is in DC (Daytona data) format

SYNOPSIS
Check_DC_Lines [ -d delims ] [ -C com ] [ -n nbr_fields ] [ -t tuple_delims ] [ -E max_errors ] [ -variants_ok ] [ -save_faults ] [ -save_nonfaults ] [ -v ] [ file ]

DESCRIPTION
Check_DC_Lines is used to quickly examine a DC data file to look for corruption and check for proper syntax. It does not make any use of rcds. Use "Sizup validate_only" to go further and check data for conformance to the data dictionary (i.e., the rcd).

OPTIONS
- d delims
  Specify the field delimiter FS in the input as delims; the default is a pipe |. A tab may be specified either by a literal tab character or by the C escape sequence \t. (This applies to the - C and - t options as well.) If the TUPLE-valued field separator ES in the input is different from FS, it should be specified by appending it to the argument to the - d option, as in the second example below.

- C com
  Specify the comment character CM in the input as com; the default is %.

- n num_fields
  Specify the expected number of fields. If not specified, that number will be determined from the first non-comment line of the input file.

- t tuple_delims
  Specify the TUPLE delimiter characters BB, BE in the input as tuple_delims. If not specified, the default is no tuple delimeters.

- E max_errors
  Specify the maximum number of errors to report before aborting.

- variants_ok
  Specify that it is acceptable to have lines with differing numbers of fields.

- save_faults
  Saves the bad records in base_file_name.faults in the current directory. If the input is from standard input, the file check_dc_lines_output.faults is used.

- save_nonfaults
  Saves the valid records in base_file_name.nonfaults in the current directory. If the input is from standard input, the file check_dc_lines_output.nonfaults is used.

- v
  Produce verbose statistics about the file.

EXAMPLES
Check the file SUPP from the Daytona sample database:
  $ Check_DC_Lines -d \t -C '#' $ORDERS_HOME/SUPP

Check the file MANIFOLD from the Daytona sample database:
  $ Check_DC_Lines -d ',' -t '[]' $ORDERS_HOME/MANIFOLD

SEE ALSO
DC-rcd(1), DC-prn(1), DC-pkt(1), csv-DC(1), DC(5), CSV(5).
NAME
Check_Indices – veri®es that indices are consistent with their data and rcds for the given record classes

SYNOPSIS
Check_Indices [ –n ] record_classes

DESCRIPTION
Check_Indices crosschecks indices against their data and rcd speci®cation for the given record classes, reporting any issues that are found.

Note that this could take a long time for record classes with a large number of partitions, records, and/or indices, since Check_Indices will identify and compare all key values in the data against the existing index®les.

A custom check can be created by using the –n argument to generate a script called ck.record_class.sh and then modifying that script to perform the desired check.

OPTIONS
–n Only generate a check script named ck.record_class.sh.

EXAMPLES
$ Check_Indices –n ORDER
Check_Indices: ORDER: using archive /home/user1/d/aar.orders

$ Check_Indices ORDER
Check_Indices: ORDER: using archive /home/user1/d/aar.orders

Now starting to work on /home/user1/d/ORDER.siz

Now starting to work on /home/user1/d/ORDER.+T
btdiag: /home/user1/d/ORDER.+T looks OK

Now starting to work on /home/user1/d/ORDER.n.T
btdiag: /home/user1/d/ORDER.n.T looks OK

Now starting to work on /home/user1/d/ORDER.sp.T
btdiag: /home/user1/d/ORDER.sp.T looks OK

Now starting to work on /home/user1/d/ORDER.p.T
btdiag: /home/user1/d/ORDER.p.T looks OK

Now starting to work on /home/user1/d/ORDER.dr.T
btdiag: /home/user1/d/ORDER.dr.T looks OK

Now starting to work on /home/user1/d/ORDER.dp.T
btdiag: /home/user1/d/ORDER.dp.T looks OK

Now starting to work on /home/user1/d/ORDER.ln.T
btdiag: /home/user1/d/ORDER.ln.T looks OK
ORDER: indices OK

SEE ALSO
intro(1), Checkup(1), Sizup(1).
NAME
Checkup – Metadata and Environment Consistency Checking Tool

SYNOPSIS
Checkup [ -proj ] [ -apps / alist ] [ -recls / rlist ] [ -as_cron ]

DESCRIPTION
For the given project, applications, and/or record classes (including views), Checkup validates any meta-
data references (to directories and files) against the current Daytona database environment, thus provid-
ing a convenient method for identifying gaps and inconsistencies in a project/application environment, which inconsistencies are particularly possible when the application is running with the +T, or +trustme, flag. The output format of Checkup is intended to be convenient for human perusal, but is also structured to support parsing by downstream tools.

Options control the particular metadata that is being checked and the level of reporting. When invoked with no arguments, Checkup prompts the user with a series of questions about what to do.

The -proj, -apps and -recls options indicate that detailed results should be provided for the project, for applications, and for record classes (including views), respectively. The lists provided to the -apps and -recls options are used to limit the output to the specified applications and record classes (including views), respectively.

The -as_cron option indicates that Checkup is being run as part of a cron job and will therefore limit its output appropriately.

The current Daytona database environment is defined by the contents of the environment variables DS_PROJ, DS_APPS, and DS_PATH (where these are most often defined by a project or application setup script which calls DS_Set).

OPTIONS
- proj Print details for the current project.
- apps / alist Print detailed results for applications. If an application list is given, then details will be pro-
vided for each application in alist. The list members may be separated by colons or spaces. When no application list is given, results will be provided for all applications (or just those applications that contain any record classes specified, if any).
- recls / rlist Print detailed results for record classes (including views). If a record class list is given, then
details will be provided for each record class in rlist. The list members may be separated by
colons or spaces. When no record class list is given, results will be provided for all record
classes (or just those record classes contained in the applications specified, if any).
- as_cron This indicates that Checkup is being run as part of a cron job. This means that no output
should be generated (and therefore no cron email) unless a potential problem has been
identified. When Checkup is run as part of a cron job, the environment must be set up first
(either through a setup script or through an invocation of DS_Set with the appropriate values
for DS_PROJ, DS_APPS, and DS_PATH).

MESSAGES
Currently, Checkup produces a heading for each type of object (PROJECT or APPLICATION or
RECORD_CLASS or VIEW) encountered. This heading is of the form:

========> <obj_type>: <obj_name>

For example:

========> PROJECT: daytona
Note that, in some cases, an object name may be [NONE].

**Checkup** also produces messages with four different formats. The first format possible is:

```
<msg_type>[!] <name>
```

where `<name>` is the (optional) entity associated with the given message, and the colon or exclamation point indicates whether the message is informational or a potential problem, respectively. This message format is currently used for DS_PROJ, FOUND_IN, MISSING_AR_FILE, MISSING_DESC_FILE, and ILLEGAL_USE_FOR.

The second format possible is:

```
<msg_type>[!] <list>
```

where `<list>` is the (optional) colon-separated list of entities associated with the given message. This message format is currently used for DS_APPS, DS_PATH, RECLS USED, APP_NOT_IN_DS_APP, and RECLS NOT_IN_APP_LIST.

The third format possible is:

```
<msg_type>[!] <note>=<full_path> [(<original_path>)]
```

which is currently only used for MISSING_DIRECTORY. In this case, `<full_path>` indicates the directory that could not be found and `<note>` indicates the description note that specified the directory. If the original directory path given in the metadata is not identical to the fully expanded path name, the original path will also be printed.

The final format possible is:

```
<msg_type>[!] <full_fpath> [(<orig_fpath>)]
```

which applies to all remaining messages. `<full_fpath>` is the full path for a file that has caused an issue. If the original file path given in the metadata is not identical to the fully expanded path name, the original path will also be printed.

In some cases, the `<msg_type>` contains additional error information, as it does for OTHER_ERROR_FOR_DATA_FILE(<err>). For clarification, this `<err>` provides the literal code for the specific errno encountered. More information on these codes can be found in `errno(3)` or `fopen(3)`.

An exhaustive list of potential messages is given below. Of these messages, a few are considered purely informational. They are:

**DS_PROJ:**  
This indicates the definition, if any, for DS_PROJ.

**DS_APPS:**  
This indicates the definition, if any, for DS_APPS.

**DS_PATH:**  
This indicates the definition, if any, for DS_PATH.

**FOUND_IN:**  
This indicates the definition, if any, for DS_PROJ.
For an associated view, this message indicates the `env.cy` file containing its definition.

```
RECLS_USED: <list>
```

For an associated view, this message indicates the list of record classes referenced within the view.

All of the remaining messages identify a potential problem with metadata and environment consistency. These messages are:

```
APP_NOT_IN_DS_APPS! <list>
```

A given application was not found in the list of applications defined by `DS_APPS`. **Checkup** will still attempt to provide information for it.

```
ILLEGAL_USE_FOR! <name>
```

An illegal use was found while validating a record class description.

The full error can be found with the command:

```
DS Tracy [-app <app>] -val_rcd_for <recl>
```

```
MISSING_AR_FILE! <name>
```

```
INVALID_AR_FILE! <full_fpath> []((<orig_fpath>))
```

An archive file (‘par’ or ‘aar’) either could not be found or was found but could not be read. The archive could truly be missing, permissions on the file could be wrong, there may be an error in `DS_APPS` or `DS_PATH`, or the archive may have been corrupted.

```
MISSING_DATA_FILE! <full_fpath> []((<orig_fpath>))
```

```
FILE_PATH_NOT_A_DATA_FILE!
```

```
NO_PERMISSIONS_FOR_DATA_FILE!
```

```
INVALID_FILE_PATH_FOR_DATA_FILE(<err>)! <full_fpath> []((<orig_fpath>))
```

```
OTHER_ERROR_FOR_DATA_FILE(<err>)! <full_fpath> []((<orig_fpath>))
```

A data file could not be found or is inaccessible. The file could truly be missing, permissions on the file could be wrong, its path could be incorrectly specified, or some other error may have occurred.

```
MISSING_DEPENDENCY!
```

```
EMPTY_DEPENDENCY!
```

```
FILE_PATH_NOT_A_DEPENDENCY!
```

```
NO_PERMISSIONS_FOR_DEPENDENCY!
```

```
INVALID_FILE_PATH_FOR_DEPENDENCY(<err>)! <full_fpath> []((<orig_fpath>))
```

```
OTHER_ERROR_FOR_DEPENDENCY(<err>)! <full_fpath> []((<orig_fpath>))
```

A file mentioned by a Depends On note (in the MAKE GOODS section of the associated ‘pjd’ or ‘apd’ file) is missing, empty, or inaccessible. The file could truly be missing/empty, permissions on the file could be wrong, its path could be incorrectly specified, or some other error may have occurred.

```
MISSING_DESC_FILE! <name>
```

The description file for a given project or application (i.e., a ‘pjd’ or ‘apd’ file) could not be
found. The associated archive file (‘par’ or ‘aar’) should be checked.

MISSING_DICT_FILE!  
EMPTY_DICT_FILE!  
FILE_PATH_NOT_A_DICT_FILE!  
NO_PERMISSIONS_FOR_DICT_FILE!  
INVALID_FILE_PATH_FOR_DICT_FILE(<err>)!  
OTHER_ERROR_FOR_DICT_FILE(<err>)!

The dictionary file specified in a description note ((Default)_Rec_Map_Spec_File) is missing, empty, or inaccessible. The dictionary may be truly missing/empty, permissions on the file could be wrong, its path could be incorrectly specified, or some other error may have occurred.

MISSING_DIRECTORY!  

The directory specified in a description note ((Default)_Fio_Dir, (Default_Data_File)_Source, (Default_Data_File)_Indices_Source, or Txn_Log_Dir) could not be found. The directory may be truly missing, permissions on the directory could be wrong, or there may be an issue with an undefined or incorrect environment variable.

MISSING_FIF_FILE!  
FILE_PATH_NOT_A_FIF_FILE!  
NO_PERMISSIONS_FOR_FIF_FILE!  
INVALID_FILE_PATH_FOR_FIF_FILE(<err>)!  
OTHER_ERROR_FOR_FIF_FILE(<err>)!

A file_info file could not be found or is inaccessible. The file could truly be missing, permissions on the file could be wrong, its path could be incorrectly specified, or some other error may have occurred.

MISSING_FIFO!  
FILE_PATH_NOT_A_FIFO!  
NO_PERMISSIONS_FOR_FIFO!  
INVALID_FILE_PATH_FOR_FIFO(<err>)!  
OTHER_ERROR_FOR_FIFO(<err>)!

A fifo (or named pipe) could not be found or is inaccessible. The file could truly be missing, permissions on the file could be wrong, its path could be incorrectly specified, or some other error may have occurred.

MISSING_FILE_BASE!  
EMPTY_FILE_BASE!  
FILE_PATH_NOT_A_FILE_BASE!  
NO_PERMISSIONS_FOR_FILE_BASE!  
INVALID_FILE_PATH_FOR_FILE_BASE(<err>)!  
OTHER_ERROR_FOR_FILE_BASE(<err>)!

A file (derived from a FILE_BASE or LIBRARY in the MAKE GOODS section of the associated ’pj’ or ’apd’ file) is missing, empty, or inaccessible. The file could truly be missing/empty (in which case a new object file or archive file may need to be built), permissions on the file could be wrong, its path could be incorrectly specified, or some other error may have occurred.
MISSING_FIO_FILE!
EMPTY_FIO_FILE!
FILE_PATH_NOT_A_FIO_FILE!
NO_PERMISSIONS_FOR_FIO_FILE!
INVALID_FILE_PATH_FOR_FIO_FILE(<err>)!
OTHER_ERROR_FOR_FIO_FILE(<err>)!

MISSING_INDEX_FILE!
EMPTY_INDEX_FILE!
OUT_OF_DATE_INDEX_FILE!
FILE_PATH_NOT_A_INDEX_FILE!
NO_PERMISSIONS_FOR_INDEX_FILE!
INVALID_FILE_PATH_FOR_INDEX_FILE(<err>)!
OTHER_ERROR_FOR_INDEX_FILE(<err>)!

MISSING_LIBRARY!
EMPTY_LIBRARY!
FILE_PATH_NOT_A_LIBRARY!
NO_PERMISSIONS_FOR_LIBRARY!
INVALID_FILE_PATH_FOR_LIBRARY(<err>)!
OTHER_ERROR_FOR_LIBRARY(<err>)!

MISSING_INDEX_FILE!
EMPTY_INDEX_FILE!
FILE_PATH_NOT_A_INDEX_FILE!
NO_PERMISSIONS_FOR_INDEX_FILE!
INVALID_FILE_PATH_FOR_INDEX_FILE(<err>)!
OTHER_ERROR_FOR_INDEX_FILE(<err>)!

MISSING_LIBRARY!
EMPTY_LIBRARY!
FILE_PATH_NOT_A_LIBRARY!
NO_PERMISSIONS_FOR_LIBRARY!
INVALID_FILE_PATH_FOR_LIBRARY(<err>)!
OTHER_ERROR_FOR_LIBRARY(<err>)!

NEWER_SRC_THAN_OBJ!

RECL_NOT_IN_APP_LIST!

An I/O file (e.g., <recl>.o or <recl>_fio_pkg.a) for a given record class is missing, empty, or inaccessible. The file could truly be missing/empty, permissions on the file could be wrong, its path could be incorrectly specified, or some other error may have occurred.

I/O files can be built using:

DS Tracy [-app <app>] -gen_fio_for_recls <recl>

The KEYS section of a record class description specifies the index files that should be found for each data file. One of these index files is missing, empty, out-of-date, or inaccessible. The file could truly be missing/empty, permissions on the file could be wrong, its path could be incorrectly specified, or some other error may have occurred. A Sizup may be needed for the associated data file (even if this data file does not have a subsequent SIZUP_NEEDED_FOR message).

A file (derived from LIBRARY in the MAKE_GOODS section of the associated 'pjd' or 'apd' file) is missing, empty, or inaccessible. The file could truly be missing/empty (in which case a new object file or archive file may need to be built), permissions on the file could be wrong, its path could be incorrectly specified, or some other error may have occurred.

The specified C source file is newer than its associated object file. A recompilation is likely needed. A "gen_fio_for_recls" might also be needed if the message is for an I/O file (see above).

A given record class was not found in the list of applications being searched. No additional information can be provided for it.
SIZUP_NEEDED_FOR! <full_fpath> [(<orig_fpath>)]

Sizup needs to be run for the given data file, possibly because of missing or out-of-date index files.

More information can be found using:

```
DS Sizup -jt [-app <app>] -recls <recl>
```

The index files can be re-built using one of:

```
DS Sizup -m [-app <app>] -recls <recl>
DS Sizup -m [-app <app>] -fls @<recl>=<file>
```

For more information about a project, use one of:

```
DS Archie -proj <proj> -t
DS Archie -proj <proj> -p pjd.<proj>
```

For more information about an application, use one of:

```
DS Archie -app <app> -t
DS Archie -app <app> -p apd.<app>
```

For more information about a record class, use one of:

```
DS Archie -proj <proj> -p rcd.<recl>
DS Archie -app <app> -p rcd.<recl>
```

Or to see in an editor, use the likes of `DS Vi` or `DS Vu`.

**EXAMPLES**

```
$ Checkup -recls SUPPLIER NETFLOWED
```

```
DS_PROJ: daytona
DS_APPS: orders:misc
DS_PATH: .:/test/d
```

```
======> PROJECT: daytona
```

```
======> APPLICATION: orders
```

```
======> RECORD_CLASS: SUPPLIER
MISSING_INDEX_FILE! /test/d/SUPP.siz
MISSING_INDEX_FILE! /test/d/SUPP.1.T
SIZUP_NEEDED_FOR! /test/d/SUPP
```

```
======> APPLICATION: misc
```

```
======> RECORD_CLASS: NETFLOWED
MISSING_FIO_FILE! /test/d/NETFLOWED.o
MISSING_DATA_FILE! /test/d/NETF.00 (${ORDERS_DATA:-$ORDERS_HOME}/NETF.00)
```

( note that more info is obtained by adding `-proj` and `-app` )

```
$ Checkup -proj -apps -recls SUPPLIER
```

```
DS_PROJ: daytona
DS_APPS: orders:misc
DS_PATH: .:/test/d
```
======> PROJECT: daytona
MISSING_DIRECTORY! /missingdir/d
MISSING_LIBRARY! /missingdir/d/bothapps.a

======> APPLICATION: orders
MISSING_FILE_BASE! /test/d/PERSON.o

======> RECORD_CLASS: SUPPLIER
MISSING_INDEX_FILE! /test/d/SUPP.siz
MISSING_INDEX_FILE! /test/d/SUPP.1.T
SIZUP_NEEDED_FOR! /test/d/SUPP

ENVIRONMENT

DS_PATH
Colon-separated list of directories to search for project and application archives.

DS_PROJ
Specification of the current project, if any; used to locate the appropriate project description file.

DS_APPS
Colon-separated list of current applications.

SEE ALSO
intro(1), errno(3), fopen(3), Archie(1), Edit(1), Sizup(1), Tracy(1).
NAME
Sizup – maintain Daytona data files and indices

SYNOPSIS
Sizup [ -app app ] [ -recs classes ] [ -fls files ] [ -fls_via_stdin ] [ -m ] [ -jt ] [ -add_new_keys ]
[ -no_validate ] [ -validate ] [ -validate_only ] [ -nullsock ] [ -delete_faults ] [ -save_faults ] [ -dmrcd ]
[ -lock_patience <patience> ] [ -create ] [ -truncate ] [ -clean_slate ] [ -padding ] [ -packing ]
[ -source dir ] [ -indices_source dir ] [ -adds_indices_source_dir ] [ -adds_fl addfile ]
- rec_map_spec_file file ] [ -max_doable_fls file_count ] [ -nonempty_flss_only ] [ -parallel_for
clone_count ] [ -do_section sec_nbr/hot_secs ]

DESCRIPTION
Sizup reads one or more data files in the Daytona DC format (see DC(5)), checks the data for validity
against the local data dictionary, and builds derived B-tree and siz indices. Optional additional actions
include rewriting the data file to eliminate freed data records, appending a file of additional records, or
padding the records in the file to a stated minimum length.

By default, Sizup checks the time stamps of each data file against those of the associated siz file and
rcd. If the siz file exists and is newer than the data file, and is newer than the rcd, then Sizup assumes
all indices associated with the file exist and are up-to-date. If the indices are not up-to-date, then, unless
directed otherwise, Sizup will validate the data and rebuild the indices. The -jt and -m options modify
this behavior, as described below.

Ordinarily, Sizup performs validity checks upon the data, and computes reachability statistics for the
indices when building them, and writes this information back to the rcd. Each activity can be optionally
suppressed, as described below. Each data file visited must be readable and writable by the user. How-
ever, the contents of the data files are not altered unless a data-altering option such as -packing or
-adds_fl is supplied. Even if Sizup changes the contents of the rcd (such as by inserting new or
updated statistics), it will, by default, preserve the previous modification time of the rcd. The -m
option modifies this behavior, as described below.

OPTIONS
-E or -ERROR
Print error messages, but suppress warning and informational messages.

-W or -WARNING
Print error and warning messages, but suppress informational messages. This is the default.

-F or -FYI
Print all messages.

-recs classes
As needed or as else mandated by -m, validate data and build indices for all the data files for
each record class that appears in the space-separated list classes. The use of the -recs keyword
and arguments is completely independent of any use of the -fls keyword and arguments.

-fls files
As needed or as else mandated by -m, validate data and build indices for all data files that
appear in the space-separated list files. If a file path is prefixed with @<record_class>= (as in
@SUPPLIER=SUPP), then Sizup will only search for that file path in the rcd for the indicated
record class, instead of searching the rcds one-by-one in sequence in an effort to find the first
record class description that refers to the given file path. Not only is the @-syntax faster for
Sizup, it also reduces lock contention on the rcds. The use of the -fls keyword and arguments
is completely independent of any use of the -recs keyword and arguments.

-fls_via_stdin
With this option, instead of reading file paths to work on from the command line, Sizup will
read them from stdin, where they should appear one per line.

-app app
Look for all rcd’s in the file aar.app. This option overrides the DS_APPS environment variable. Whether or not this option is given, all requested rcd’s must be contained in a single aar, as Sizup only works with one application per invocation.

- `--jt` or `--just_testing`
  Test the timestamps of data file, siz index and rcd against each other for currency, but do not build indices even if the file or indices are out-of-date. Print a message saying whether the indices are up-to-date and why they are not if they are not; return exit status 0 if the indices are up-to-date, otherwise return exit status 1.

- `--m` or `--mandatory`
  Build indices even if the timestamps of data file, siz index and rcd suggest that the indices exist and are up-to-date. Update the rcd modification time, unless the `--dmrcd` or `--adds fl` option is supplied.

- `--add_new_keys`
  Build just those indices which do not already exist. With this option it is an error if out-of-date indices exist.

- `--validate` or `--no_validate`
  Enables (by default) or disables data validation checks.

- `--validate_only`
  Causes Sizup to validate the data but disables the creation of any indices, including the .siz.

- `--nullsok`
  Allow missing values anywhere in the data.

- `--delete_faults`
  With this option, Sizup will automatically delete faulty records as it finds them. That particular Sizup run will end with error messages and with unusable indices; however, just rerunning that particular Sizup invocation again will be successful because the previously faulty records will have been deleted by the first run.

- `--save_faults`
  With this option, Sizup creates a file containing information pinpointing the exact location of each faulty record as well as displaying the record itself. The fault file is placed in the same directory as the data and has a name consisting of the data file name suffixed by .faults, as in ORDER.faults. Each fault generates an entry that consists of a comment record (consisting of the line number, the record number, the record offset, and the record length) followed by the faulty record itself. Please note that in the case of duplicate unique keys only, information on the bad record is put into Sizup.msgs and not into the .faults file.

- `--dmrcd`
  Do not compute index statistics and do not modify the rcd; since it does not need to modify the rcd, it will also refrain from getting exclusive locks (both file lock and lock file) on the rcd.

- `--lock_patience`
  When, at the start of a run, Sizup is inventorying the status of the files it has been assigned to work with, unless otherwise instructed, it obtains an exclusive file lock on each file in turn. In part, this is to determine whether or not the file is currently being worked on by a transaction or by another Sizup. In order to keep this inventory process from taking an indefinite period of time due to blocking indefinitely on pre-existing locks, Sizup by default waits for 15 seconds and if it has not been able to obtain the lock by then, it exits with an error message. The `-lock_patience` option allows the user to specify a different time interval, if desired. This can be either `--wait_on_block`, `--fail_on_block`, or a non-negative number of seconds. An argument of 0 (i.e., no patience) is equivalent to `--fail_on_block`.

- `--create`
  Create the data files, and create matching empty indices, if the data files do not already exist. Existing data files are treated in the usual way.
User Commands

–trunc
e Truncate the data files and create matching empty indices.

–clean_slate
 Equivalent to –create –trunc.

–packing
 Compactify the data file by removing the freed records.

–padding
 Ensure each data record is of at least the pad length, as defined in the rcd, by supplying extra comment bytes at the ends of undersize records.

–source dir
 Assume all data files are located in dir. This overrides any rcd Source notes.

–indices_source dir
 Assume all index files are located in dir. This overrides any rcd Indices_Source notes.

–adds_fl addfile
 Validate and build indices for addfile, then append addfile to the data file and merge its indices with the previously existing indices for the data file. The data file and its existing indices must be up to date. The rcd modification time is not changed.

–adds_indices_source dir
 Assume all index files for the adds files are located in dir. This is only for _recursive_ batch adds, now deprecated.

–rec_map_spec_file file
 file is the rec_map_spec_file (frequently, compression dictionary) to use when mapping the raw data to the DC format that Sizup understands. This option overrides any Rec_Map_Spec_File in the rcd.

–max_doable_fl file_count
 Sizup will do at most file_count files this invocation. This is useful when specifying that a large horizontally-partitioned record class should be processed where it is too large to be done in one invocation.

–nonempty_fl only
 Sizup will only consider nonempty files for further processing. A nonempty file exists but has size 0.

–parallel_for clone_count
 After ascertaining all the files that need work, Sizup will clone itself into a total of clone_count processes and work on its assignment in parallel.

–do_section section_nbr/total_sections
 After ascertaining the total number of files that need work, Sizup will divide them up into a total of total_sections pieces and do the section with number section_nbr.

EXAMPLES
 Build indices for all data files associated with the record classes SUPPLIER and PART:

$ Sizup -app orders -recs SUPPLIER PART

Build indices for some specific data files:

$ Sizup -app orders -fls $ORDERS_HOME/d/SUPP PART

ENVIRONMENT

DS_APPS
 Colon-separated list of application names, from which to generate names of application archives.
DS_PATH
Colon-separated list of directories to search for application archives.

BUGS
Pathnames for files supplied with the \texttt{-fls} option may only include `.` or `..` as the first component of the pathname.

SEE ALSO
\texttt{DC(5)}, \texttt{Tracy(1)}. 
NAME
Census – computes statistics for a collection of data files

SYNOPSIS
Census [ -c{ommas} ] [ -i{ndices} ] [ siz_file_paths_RE ]*

DESCRIPTION
Census computes statistics for a collection of data files identified by siz_file_paths_RE. Note that it may be necessary to quote shell expressions in writing these paths. If no path expressions are given, the default of ./* .siz will be used.

Statistics for the associated indices will be included when the argument -i{ndices} is given.

OPTIONS
- c{ommas}
  Use commas when writing out large numbers (for better readability).
- i{ndices}
  Output statistics for associated indices.

EXAMPLES
$ Census '$ORDERS_HOME/HOPPER.*.siz'

=============================================================================
Starting to work on siz_filepaths = HOPPER.*.siz
=============================================================================
Total_Records = 10000
Total_Data_In_Bytes = 466709
Total_Data_In_GB = 0.000434656627476215
Min_Data_File_Bytes = 75675
Avg_Data_File_Bytes = 77784.83
Max_Data_File_Bytes = 80213
File_With_Max_Data_File_Bytes = HOPPER.6
Min_Nbr_Recs = 1622
Avg_Nbr_Recs = 1666.67
Max_Nbr_Recs = 1719
File_With_Max_Nbr_Recs = HOPPER.6
Min_Rec_Len = 35
Avg_Rec_Len = 46.67
Max_Rec_Len = 103
Total_Readable_Siz_Files = 6
First_Readable_Siz_File = HOPPER.1.siz
Last_Readable_Siz_File = HOPPER.6.siz
Status GOOD: 6

======== timing info @ Sun Nov 06 20:30:01 EST 2011 ==========
elapsed time = .229584s
own_user_time = 0s
own_sys_time = .01s
own_user_sys_time = .01s

nbr_of_kids_waited_for = 1
kids_user_time = .01s
kids_sys_time = .01s
kids_user_sys_time = .02s

ratio of cpu to elapsed time = 0.131

8:30pm up 34 day(s), 12:13, 53 users, load average: 0.29, 0.48, 0.75

==========================================

SEE ALSO
intro(1), Checkup(1), Sizup(1).
NAME
ds_m4 – a Daytona macro facility extending the functionality of m4

SYNOPSIS
ds_m4 [ -e ] [ -B bytes ] [ -Hsize ] [ -Ssize ] [ -Tsize ]
[ [ -s ] [ -D name[=val] ] [ -U name ] [ -I pathlist ]* files ]*

DESCRIPTION
ds_m4 is a macro facility, used by Daytona, which extends the functionality of m4(1) (see "man m4").
Macro expansion is applied to the given files, if any. stdin is used if no files are given.

Daytona customizes its use of ds_m4 by redefining all m4 macro names to begin and end with an underscore, by changing comment and quote characters, and by adding additional macros. To use ds_m4 as Daytona uses it, use the command "DS M4" (for more information, see "DS Man M4").

OPTIONS

-e Operate interactively. Interrupts are ignored and the output is unbuffered.

-B bytes Specify the size in bytes of the internal pushback and argument text buffer. Error messages announcing exhaustion of this resource can be forestalled by providing a larger amount for this argument. Note that the ds_m4 default is 4096 bytes, but "DS M4" will use 40960 bytes.

-H size Change the size of the symbol table hash array from the default of 199. The size should be prime.

-S size Change the size of the call stack from the default of 100 slots (macros take three slots and non-macro arguments take one).

-T size Change the size of the token buffer from the default of 512 bytes.

-s Enable line sync output for the C preprocessor (#line ...). This argument can be interspersed among the input file names to provide file-specific behavior.

-D name[=val] Define name (optionally with the value val). This argument can be interspersed among the input file names to provide file-specific behavior.

-U name Undefine name. This argument can be interspersed among the input file names to provide file-specific behavior.

-I pathlist Specify additional directories to search for included files. pathlist is a colon-separated list of directory paths. This argument can be interspersed among the input file names to provide file-specific behavior.

DETAILS
As an extension to m4, ds_m4 has the same macros as m4 does, which are described in the m4 man page. ds_m4 also has two additional macros that m4 does not: _m4_file_ expands into a double-quoted string containing the current file name (which string is "." if stdin) and _m4_line_ expands into a double-quoted string containing the current line number in the file.

When it comes to finding include, not input, files, first, if the file argument to include contains a dollar sign or begins with a tilde, then it is expanded by echoing it into the (Korn) shell. Then if the possibly expanded file begins with a slash, then that entire path is opened. Otherwise, the current directory is searched for the include file. If not found, then the directories that are searched first are the ones, if any, specified with the -I option, in the order found on the command line. (A -I argument consists of one or more directory paths separated by colons.) Finally, as needed, if the M4PATH environment variable is set, it is expected to contain a colon-separated list of directories, which will be searched in order. -I arguments and the M4PATH value may contain unexpanded shell expressions using dollar; a tilde can
only be used if it starts the concatenation of the \-I arguments and the M4PATH value. Basically, the only expansions that occur are the ones that would occur if the concatenation of the \-I arguments and the M4PATH value were echoed by ksh. Obviously, the tilde is a little finicky here -- safer to use the dollar sign.

**EXAMPLES**

```bash
$ ds_m4 <<!
#define_(FOOBAR, this is the replacement text)
hello
FOOBAR
goodbye
!

hello
this is the replacement text
goodbye

More examples can be found in $DS_DIR/EXAMPLES/sys/Archie.cy.
```

**SEE ALSO**

`intro(1)`, `M4(1)`. 
**NAME**

M4 – a macro facility for Daytona Cymbal and DSQL queries

**SYNOPSIS**

\texttt{DS M4 ( - | files )}

**DESCRIPTION**

\texttt{DS M4} supports macro expansion for Cymbal and DSQL queries in the Daytona environment. \texttt{DS M4} is currently equivalent to invoking \texttt{ds_m4} directly with the following options:

\texttt{ds_m4 -B40960 \$DS_DIR/rename.m \$DS_DIR/sys.macros.m [proj.env.m][usr.env.m]}

where the \texttt{-B} argument increases the push-back buffer by a factor of 10 over the default, and the \texttt{* .m} files provide context for expansion in the Daytona environment (and where the project-specific and user-specific files are used if they exist).

Additional arguments to \texttt{DS M4} can be specified by modifying the definition of the shell variable \texttt{DS_M4FLAGS} (which is the empty string by default).

Deviating from m4/ds\_m4, quoting within macros uses the digraphs \texttt{@<} and \texttt{> @} (known as \texttt{squawks}). These are also useful for delaying or preventing further expansion (sometimes this is required to avoid an infinite recursion). Furthermore, double-quotes serve as the comment characters.

Since it uses rename.m, all of m4’s standard macros have been renamed to start and end with an underscore.

Some common macros are:

\begin{verbatim}
#include_
Include, and expand, the given file.

#define_
Create a new macro with the given name and value.

#define_
The same as \texttt{#define} except that it does not include the new-line that follows its final parenthesis.

_BC
Begin a comment block.

_EC
End a comment block.

_m4_file_
Expands into a double-quoted string containing the current file name (which string is "." if stdin).

_m4_line_
Expands into a double-quoted string containing the current line number in the file.
\end{verbatim}

**EXAMPLES**

\texttt{$ DS M4 - <= !
#include_\_DEFINE_ FOO, @<@<BAR>@ NONE>@
#include_\_DEFINE_ BAR, @<this is line 1 for FUN
this is line 2 for FOO
this is line 3 for FEE>@)
hello
FOO
BAR
goodbye}
User Commands

M4(1)

!

hello
BAR NONE
this is line 1 for FUN
this is line 2 for BAR NONE
this is line 3 for FEE
goodbye

More examples can be found in $DS_DIR/EXAMPLES/sys/Archie.cy.

SEE ALSO
intro(1), ds_m4(1), Tracy(1).
NAME
Tracy – process Daytona query files

SYNOPSIS

Tracy [-apps apps] -gen_fio_for_recs class

DESCRIPTION
Tracy processes a query file consisting of Cymbal and/or DSQL statements, and generates C code that implements the requests, together with a makefile for make(1).

For each record class name (i.e. table name) class encountered in the query, the corresponding data dictionary rcd.class is consulted in order to determine the layout of the table, and a table-specific, query-independent C file class.c is created in the corresponding directory, if it does not already exist or is out of date. This file is called the IO file for the table, and can be created ahead of time, if desired, by using the -gen_fio_for_recs option.

Data dictionaries are found by searching a list of directories (specified in the environment variable DS_PATH) for application archives, each of which has a name of the form aar.appname, where appname appears in the application list (specified in the environment variable DS_APPS, or via the -apps option). Each aar.appname is an ar(1) or ds_ar(1) archive whose members (all text files) are the application description file apd.appname and data dictionaries rcd.class for the various tables. If more than one application archive contains a member rcd.class, only the first one encountered is used. The search strategy works through the list of application names, and for each application name works through the directory list DS_PATH. If a data dictionary cannot be found, or if there is an application name for which no application archive can be found, Tracy announces an error and terminates processing.

Tracy has two modes to support the two query languages it accepts. In mixed mode, Tracy accepts arbitrary Cymbal statements and most DSQL statements. The DSQL constructs that may not occur in mixed mode are:

- ‘--’ as a comment starter
- begin ... end
- statements beginning with a parenthesis.

In mixed mode, the semicolon character may be used as a statement separator or statement terminator. In SQL mode, Tracy accepts arbitrary DSQL statements, but no Cymbal statements. The semicolon may only be used as a statement separator in SQL mode.

By default, Tracy processes all text between the mode delimiters ‘$[’ and ‘$’ in SQL mode, and everything else in mixed mode. Alternatively, there are several ways of directing Tracy to process the entire file in SQL mode, as described below. This is useful for handling arbitrary DSQL queries without having to nest the whole query, or certain statements, in the mode delimiters.

OPTIONS
- E or -ERROR
  Print error messages, but suppress warning and informational messages.
- W or -WARNING
  Print error and warning messages, but suppress informational messages. This is the default.
- F or -FYI
  Print all messages.
- r file
  Process file as the query file. If ‘-’, is used instead of -r file, read the query from the standard input. If file ends in ‘.S’, process the whole query file in SQL mode.
- apps apps
  Use the colon-separated list apps as the list of application names when searching for data
User Commands

dictions. This option overrides the DS_APPS environment variable.

-SQL  Process the entire query file in SQL mode.

-ABC  Generate code that includes checks for array bound overflow.

-SOC  Generate code that includes checks for string overflow.

-ZDC  Generate code that includes checks for division by zero.

-COU  Generate code that includes checks for any transaction that assumes it can see its own updates.

-VTC  Generate code that includes variable trace messages. Tracing can be turned on or off from
within the query by setting the variable vbl_tracing to _true_ or _false_. When the
query includes a begin task (i.e. when Tracy generates C code that includes main()), tracing
is turned on automatically when the program starts. If the user supplies main(), tracing must
be turned on explicitly.

-IVC  Generate code that includes (implicit variable) checks for user variables that have no explicit
scoping appearances.

-DUV  causes Tracy to print out such descriptive information as the type and scope on each variable
defined or imported into each task.

-gen_fio_for_recls class
   Just generate the IO file class.c for the record class class.

EXAMPLES
   Process the query file prog.Q, and print all messages:
      $ Tracy -apps orders:misc -FYI -r prog.Q
   Process an all-DSQL query, printing only warning & error messages:
      $ Tracy -W -r prog.S
   Process an all-DSQL query:
      $ cat prog.S | Tracy -SQL -

ENVIRONMENT

DS_APPS
   Colon-separated list of application names, from which to generate names of application
   archives.

DS_PATH
   Colon-separated list of directories to search for application archives.

DS_SQLONLY
   If value is ‘y’, process query file in SQL mode.

SEE ALSO
   Mk(1), Sizup(1).
NAME
Mk – makes an executable from the C code generated for a Daytona query

SYNOPSIS
DS Mk [ executable ]

DESCRIPTION
Once Tracy has been used to generate C code for a Daytona query, DS Mk is used to make an executable (with the given name) from that C code. When no executable name is given, DS Mk will create an executable named R.

DS Mk expects to find a makefile named R.mk and corresponding C code with names matching R*.[ch].

EXAMPLES
$ DS Tracy -r qname.Q
Tracy finished for qname.Q

$ ls
R.h R.mk R_0.c R_0.h

$ DS Mk qname.X
...
R moved to qname.X
compilation completed successfully

SEE ALSO
Tracy(1).
Name

DS Compile

SYNOPSIS

DS Compile   [ <request_file>   [ <executable>   ] ]
             stdin is used for - as <request_file>
             if <executable> is -, then
             <file>[qQsS] -> <file>.X, <file>.cy -> <file>,
             else <file>   -> <file>.X
             Also, Q/<file>   -> bin/<corresponding-file>

DESCRIPTION

Daytona processes Cymbal and SQL queries by translating them into C code complete with make file
(by using Tracy), using a C compiler to compile that C code into an executable, and then invoking the
executable perhaps with runtime arguments and formatting filter processes into order to produce the out-
put of the query. DS Compile takes care of processing a Cymbal or SQL query into an operating system
executable.

When invoked with no arguments, DS Compile will prompt the user for a wide variety of options -- and
then actually, will offer to run the resulting executable possibly with arguments.

If the first argument is -, stdin is assumed. If the second argument is missing, then R is assumed; if it
is -, then the name of the executable is formed in the following way:

-) If the query file name ends with .[qQsS], then the executable name is formed from the query file
name by replacing that suffix with .X .

-) If the query file name ends with .cy, then the executable name is formed from the query file name
by removing that suffix.

-) Otherwise, the executable name is formed from the query file name by appending .X .

Independently of the above, if the query file path begins with Q/, then the executable file path will
begin with bin/ .

DS Compile uses a lock-file-based locking mechanism to ensure that there are never two DS Compiles
working at the same time in the same directory.
NAME

DS Edit – supports editing of rcds, apds, pjds, queries, and view and fpp definitions

DS Vi/Vu – DS Edit using vi
DS Emacs/Emucs – DS Edit using emacs
DS Joe/Jou – DS Edit using joe

SYNOPSIS

DS Edit [ -Readonly ] [ -EK ] [ -dmtm ] [ -env ] files/names [ -apps apps ] [ -proj proj ] [ -ar[ch] ar_file ]

DESCRIPTION

DS Edit can be used to edit any record class description (rcd), view definition, function/predicate/procedure (fpp) definition, application description (apd), project description (pjd), or query or other file that can be found in the current environment. A project name, application name, and/or archive file can be provided to restrict the search for the desired object.

In the case of views and fpps, the '-env' argument must be given to indicate that '*.env.cy' files should be searched (as well as any packages that the env.cy files import).

In the case of rcds, apds, and pjds, the file will be checked out of the appropriate archive file and will be checked back in at the completion of the editing session. Edit automatically locks and unlocks various files to ensure that Edits do not interfere with other processes including other Edits.

By default, DS Edit will use the vi editor unless this has been overridden by either the VISUAL or EDITOR shell environment variable. DS Vi, DS Emacs, or DS Joe can be used instead of DS Edit to invoke a specific editor. DS Vu, DS Emucs, or DS Jou can be used to invoke these same editors but in read-only mode (-Readonly).

OPTIONS

-Readonly
Start the edit session in read-only mode.

-EK
When editing a description (rcd, apd, or pjd), use the English Keyword form of the file.

-dmtm
When returning an edited description (rcd, apd, or pjd) to an archive, ensure that the file modification time for the description is unchanged.

-env
When editing a view or fpp (function or predicate or procedure) definition, this indicates that '*.env.cy' files should be searched (as well as any packages that those env.cy files import).

-apps apps
Search in the specified applications.

-proj proj
Search in the specified project.

-ar[ch] ar_file
Search in the specified archive file.

ENVIRONMENT

DS_PATH
Colon-separated list of directories to search for project and application archives.

DS_PROJ
Specification of the current project, if any.

DS_APPS
Colon-separated list of current applications.
SEE ALSO
   intro(1), Synop(1), Synop_fpp(1).
Name
    DS Show

SYNOPSIS
    DS Show [ <recls> [ <there_isa_where> ] [ <p|t|d|x|s|u = t> ] ]

DESCRIPTION
    The DS Show command is used to print out record classes, data files, or executable output. Simply invoke it and answer the questions:

        % DS Show
            enter:  r to display a record class
                    e to display executable output;
                            else display a data file

When a record class is chosen, DS Show will use DS QQ to compile and execute a query to print out the associated records in any of the table, packet, Cymbal description, XML, or unfiltered formats. Obviously, the data dictionary is used here to locate and work with the data files.

If DS Show is asked to work with a single data file, then the same output formats are available but the user will have to provide the path to the file as well as the unit separator(s), comment character, and other related syntactical details. When asked to work with data files (as opposed to record classes), DS Show will demonstrate that it does not know how to display plaintext values of FIELDS having compressed types like HEKA, nor does it know how to work with files written using record-level compression. Such capabilities are available only when DS Show works with a record class. DS Show will also directly display the output of executables: just give it the name of the executable and various other syntactical specifications.

Note that DS Show can also be called with arguments so as to display the contents of record classes, whether conventional disk tables or views.

        DS Show <recls> [ <p|t|d|x|s|u = t> ]

The optional second argument provides the contents of a Cymbal there_isa_where clause as illustrated by:

        DS Show SUPPLIER 'Number <= 415 and City Matches "o"'

The optional third argument with values p|t|d|x|s|u and default t specifies one of the formats _packet_, _table_, _desc_, _xml_, _safe_ and unformatted.
SYNOPSIS

DS Expr [ -v{erbose} ]

DESCRIPTION

DS Expr is a convenient command utility that takes a constant Cymbal term or expression from stdin, evaluates it, and prints it to stdout. Recall that a constant Cymbal term is a constant or a function call with constant term arguments. Here is a sample use:

```
% echo "4*5" | DS Expr
20
```

Here is another example using the -verbose option:

```
% echo "today()+10" | DS Expr -v

today()+10 = 2005-09-07
```

The stdin input to DS Expr may alternatively be terminated with a `D, as is the UNIX convention. Here is DS Expr doing a little helpful casting:

```
% DS Expr
(TIME(_dhmsf_))^50000s^TIME
^D
13h53m20s
```

Caveat: DS Expr is implemented by means of DS QQ and the _Show_Exp_To macro: thus, unfortunately, it is subject to the limitations of m4. In particular, any trouble it may have with double-quotes can be avoided by using LITERAL constants instead of STRING constants.

If DS Expr creates temporary files, it creates them in /tmp by default, else in $DS_TMPDIR if DS_TMPDIR is set.
**Name**

QQ – Quick Query

**SYNOPSIS**

```
DS QQ [ -t | -p | -d | -x | -u ] [ -v(erbos) ] <request_file> [
<query_args> ]
```

`/ *` stdin if <request_file> is ` -` (implies no prompting and no
messages) */

**DESCRIPTION**

QQ provides a simple, quick way to compile and execute Cymbal and SQL request programs.

Here is a typical interaction:

```
% DS QQ stpaul.1.S
```

enter y to edit request; else begin processing immediately:

```
enter p for packet, t for table, d for description, x for xml, \nelse unfiltered output: t
```

is this comment/field delimiter spec (` -C’%’ -d’|’`) acceptable? (yes):

```
Working
```

to see output via $PAGER, type <return>:

```
Query File: ./Q/stpaul.1.S
```

<table>
<thead>
<tr>
<th>Supplier</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acme Shipping</td>
<td>612-149-5678</td>
</tr>
<tr>
<td>Bouzouki Receiving</td>
<td>612-943-7416</td>
</tr>
<tr>
<td>Julius Receiving</td>
<td>612-309-3492</td>
</tr>
<tr>
<td>Sunshine Warehouse</td>
<td>612-303-7074</td>
</tr>
</tbody>
</table>

If the file name ends with .Q (for Cymbal or DSQL) or .S (for pure standard SQL), then the path
`./Q/../Q:/..Q:/q` is searched for the file; else the current directory alone is searched. If no
file by the given name is found, one will be created.

The `-t`, `-p`, `-d`, `-x` options inform QQ that the desired output format is tabular, packet, Cymbal
description, or XML respectively; the default (i.e., `-u`) is unfiltered. The filtering options assume that
the data is being presented in a form compatible with what the corresponding filters like DC-prn require.

If `DS QQ` is invoked with the `-v` verbose option, then not only will messages indicate each step of the
query evaluation but also the output will be saved in a temporary file that QQ will identify. If the
(query) input to `DS QQ` is from `stdin`, then the output will be to `stdout`; otherwise, it will be sent to
`$PAGER` if it exists, else to `stdout`.

If `DS QQ` creates temporary files, it creates them in `/tmp` by default, else in `$DS_TMPDIR` if
`DS_TMPDIR` is set. `DS QQ` cleans up after itself by removing temporary file artifacts, with the exception
of the executable that it makes, which is named R.QQ.
If the executable requires arguments, those arguments can be placed after the `<request_file>` on the command line or given interactively.

Here is how to just run a query and get the answers formatted with no interactive fuss:

```
DS QQ -t - <Q/stpaul.1.S
```

or even:

```
DS QQ -t - "St. Paul" <Q/stpaul.3.IQ
```
NAME
Daisy – Daytona interactive SQL processor

SYNOPSIS
Daisy [-b [batch_file]]

DESCRIPTION
Daisy is an interactive SQL processor for Daytona. Users can enter SQL commands and obtain the
results displayed at their terminals.

One key feature of Daisy is an SQL Data Definition Language (DDL) interface. This allows users to
manipulate the Daytona Record Class Descriptions (rcd’s) in an SQL-like language instead of editing the
rcd’s by hand.

Daisy also has various internal commands to allow you to modify and view your environment and per-
form various DS commands in a simple manner.

USAGE
Daisy allows you to enter and run Daytona’s SQL (DSQL) commands interactively. The DSQL com-
mands can be on a single line or on multiple lines, and can consist of a single query or a series of
queries separated (and optionally terminated) by semicolons (;). Such SQL queries include select,
delete, insert, and update SQL statements. Each request to Daisy must end with the string /<enter>. The
<enter> must immediately follow the /!. The terminator string can appear on the same line as the
request or on a separate line.

Daisy will initially use the query output format specified in the Daytona environment variable
DS_SFLAGS with the -sql_out_fmt keyword. If there is no output format specified in DS_SFLAGS, Daisy
will use _packet_. You can also change the output format dynamically within Daisy (see the
set format and set shell commands below).

Daisy also has SQL Data Definition Language (DDL) commands to manipulate Daytona’s rcd’s. Daisy
has commands to create and drop tables, indices and rcd BINS descriptions.

Daisy commands, DSQL statements, and DDL commands can be mixed together into scripts as long as
some rules are followed. You cannot create a table and use it for data manipulation in the same request.
Rather, you must terminate the request for the table creation with a /<enter> before using it for data.
And obviously, you cannot delete a table and then try to use it in the same request. This will generate a
run time error.

Daisy can be run with the -b option to run in a batch mode. In batch mode, Daisy will read a series of
requests from the batch file (or standard input if no batch file is given), with each request terminated by
/<enter>. Daisy will process each request in turn until there are no more requests. If an error occurs
while processing the script, Daisy will exit with a non-zero return code.

By way of terminology, columns of tables are synonymous with fields of record classes.

Note that Daisy will silently call DS Clean_Misc -f upon exiting. This will clean out Daytona’s
temporary files. If you wish to save any files (such as Sizup.msgs), you must copy it before exiting
Daisy.

DAISY COMMANDS
Daisy has several commands in addition to DSQL statements and DDL commands. Daisy commands
are case insensitive.

help  A quick synopsis of Daisy commands, DSQL, and Daisy DDL commands is available from the
help command. The output is paginated based on the value of the shell variable $PAGER, using less if $PAGER is undefined.

set format
To change the output format of a select statement, use the command set format = <format>,
where <format> is one of _table_, _packet_, _desc_, _xml_, or _data_.

Daytona  Last change: 19 January 2010
When the format is modified, the DS_SFLAGS environment variable changes as well. Old formats are removed from the environment variable and the new format is appended.

**set file**

To redirect the output of a select statement from the terminal to a file, you can enter `set file = <filename>`. The `<filename>` is the name of the file where output should be appended. The `<filename>` can contain shell variables, or begin with `"`, which has the same meaning as in ksh. The `<filename>` file must be writable at the time of the `set file` command, as well as at the time of the execution of the select. If a shell variable is used as the file name, the shell variable must be defined and point to a valid file before Daisy processes the `set file` command, and must be defined and point to a valid file (even if it is different) when the select command is entered. To turn output direction off so that the output comes to the terminal, enter the string `set file = .`.

**set shell**

Daisy lets you change shell variables during execution. To do this, enter `set shell <variable> = <value>`. The `<variable>` can be any valid shell variable. The `<value>` can be any value and need not be in quotes. Leading and trailing spaces are removed, but other spaces in the `<value>` remain. If you want the leading or trailing spaces, or want to enter quotes, you can enclose the `<value>` in either single (`) or double (") quotation marks. To unset the `<variable>`, omit the `<value>`. Note that setting the DS_SFLAGS environment variable may affect the output format. Also note that the format information will always be included in the DS_SFLAGS environment variable.

**set echo**

Daisy allows you to turn on or off the echoing of the input when not running interactively. To turn off the echoing of the input, enter `set echo off`. To turn it back on, enter `set echo on`. By default, echo is set to on.

**see format**

**see file**

**see shell**

**see echo**

For each `set` command, there is a corresponding `see` command. You can see the current format with `see format`. You can see the current output file with `see file`. You can see the current value of shell variables with `see shell <variable>`. Or you can see the current value of the input echoing with `see echo`.

**show table**

To see the contents of a table, enter `show table <table> [as <format>] [into <file>]`. The `<format>` is one of `table`, `packet`, `desc`, `xml`, or `data`, with `packet` being the default. The optional `into` clause allows you to append the results to a file.

**list tables**

**list fields**

To see the list of tables in an application list, enter `list tables in <application_list>`. To see the list of tables in an all applications in `$DS_APPS`, enter `list tables`. To see the list of fields in a table, enter `list fields in <table>`.

**describe project**

**describe application**

**describe table**

To describe a project (i.e. a synopsis of its contents), enter `describe project <project>`. To describe an application, enter `describe application <application>`. To describe a table, enter `describe table <table>`.

**create project**

**create application**

To create a project, enter `create project <project>`. To create an application, enter
create application <application>.

drop project
drop application
To drop an application and all the tables (both data and indices) within the application, enter drop application <application>. To drop a project, all the applications in the project, and all the tables (data and indices) in these applications, enter drop project <project>.

resync project
resync application
To rebuild all indices or the objects (i.e., record class I/O files and other environment files) within an application, enter resync application <application> [indices | objects]. To rebuild all indices or all objects within all applications in a project, enter resync project <project> [indices | objects].

clean project
clean application
To erase the indices or objects of an application, enter clean application <application> [indices | objects]. To erase the indices or objects of all applications in a project, enter clean project <project> [indices | objects].

exec
The exec command must be terminated by the /! command terminator (not a semicolon) or unpredictable results will occur. The script file must end with the Daisy /! command terminator. Otherwise, all commands after the final /! terminator (which may be the whole file if there are none) will be ignored. Nesting of the exec commands is allowed. A script may exec another script. (Do not confuse this command with execute, described later.)

The script file is run in the current environment, so any changes to format, output files, or environment variables in the script files will remain in effect when the script ends. If you wish to run a script of Daisy commands in a separate environment, enter shell Daisy -b <script_file>. (See the next paragraph for explanation of the shell command.)

shell
To run a script of ksh shell commands, enter shell <shell_command>. To run multiple commands through ksh (or a single command containing a semicolon), enter shell #(<shell_command1>; <shell_command2> ... #).

prepare query
To prepare a single sql command by precompiling it for later execution, enter prepare query <query_name> from <query> [working in <directory>]. To prepare a series of sql commands by precompiling them, enter prepare query <query_name> from #(<query1>; <query2> ... #) [working in <directory>].

By using the DSQL READ command, parameterized queries can be created. The parameters can be cast on the READ statement or when they are used in the query itself. Note that Daisy allows you to read from the command line (with or without the optional FROM " _cmd line " clause) or from a file (with the FROM "file" clause). Daisy does not support READ FROM " _stdin "; this will cause the read to fail with a read call status of _instant_eoc_.

If the optional working in <directory> is not present, the current directory (.) is assumed. The prepare command will fail if <directory> (or .) does not have a Q subdirectory for the query source and a bin subdirectory for the query executable. The <query_name> (more specifically <directory>/QI<query_name>.S and <directory>/bin/<query_name>.X) must not already exist.

execute query
To run the prepared queries, enter execute query <query_name> [using <parameter1> [<parameter2> ...]] [working in <directory>]. If the query does not have any parameterized values then the using clause can be left out. If you do not provide
enough parameters then the execute command will fail. (Extra parameters are ignored.) If you have a string parameter that contains spaces you can enclose it in single or double quotes (as you would for ksh) so that it is treated as a single parameter. (Do not confuse this command with exec, described earlier.)

To see what queries you have, you can enter `shell ls <directory>/bin/* .X.`

**drop query**

To delete a prepared query source and its executable, enter `drop query <query_name> [working in <directory>]`.

Note that although the commands are named prepare query, execute query, and drop query, all series of insert, update, delete, and select statements can be included in the query.

**edit**

When working interactively, Daisy gives you the opportunity to edit the command currently being entered or the previous command if no new command has been started. This is accomplished by entering `edit /!` on a line by itself. Daisy will put you in your favorite editor (as defined by the VISUAL environment variable, or the EDITOR environment variable if VISUAL is undefined, or else vi if neither VISUAL nor EDITOR is defined). You can then edit the commands in progress.

When saving the edited request, the terminating /! is optional. The request is immediately processed by Daisy when you exit the editor. If you change your mind, you can empty the editor buffer and save the now empty file. Daisy will ignore the empty file and allow you to enter your next command.

**exit**

To exit Daisy, enter exit or quit. Daisy will also exit on an end-of-file (usually `ctrl-D` when interactive).

Daisy will ignore input from `whitespace` to the end of the line, or lines beginning with `/`. These are treated as Daisy comments. If the `/` is not at the beginning of a line or preceded with `whitespace`, the `/` is ignored. This is done to allow such statements as

```
insert into web_addr values("Yahoo", "http://www.yahoo.com")
```

since the `/` is preceded by a colon(:).

Sometimes Daisy will get confused as to what is a command for it to process and what is a valid DSQL statement. You can force Daisy to treat a string as DSQL by enclosing it between `$` and `]`.

If you hit the interrupt key while interactively typing in a request, Daisy will ignore the request you are typing and prompt you to begin again. The current request in progress will not be available for editing.

If you hit the interrupt key while Daisy is processing a request, the current request is aborted.

If you hit the interrupt key while Daisy is in batch mode, Daisy will exit.

When running interactive DML commands that modify the data, Daisy will report the number of rows inserted, updated and/or deleted. Note that prepared queries do not display this information.

**DDL COMMANDS**

Daisy’s DDL commands allow you to create and drop tables, indices, and rcd FILE and FILE_INFO FILE BINS.

```
CREATE TABLE Table_Name [VERSION 'Version'] [APPLICATION Application] ( 
   Field_Name Field_Type <Optional_Parameters>
   [, Field_Name Field_Type <Optional_Parameters>]... 
)
```

where Optional_Parameters are:

```
[[NOT] NULLABLE] [DEFAULT Default_Value] [MIN Min_Value] [MAX Max_Value]
[VALIDATION_RE Validation_Re] [PARTITIONER]
```
This is the basic create table statement which serves to create a basic stub Daytona rcd with only the **FIELDS** section of the rcd filled in. The **CREATE FILE BIN** and **CREATE INDEX** commands are used to flesh out the **BINS** and **KEYS** portion of the rcd.

The **Table_Name** will be converted to an Uppercase. The **VERSION** option, if present, must immediately follow the **Table_Name**. The **Version** string must be enclosed in single quotes. The **APPLICATION** option must be present if the **DS_APPS** environment variable contains more than one application (as in *orders:misc*) and must be one of the applications in the **DS_APPS** environment variable.

The records in the table consist of one or more fields. Each **Field_Name** will be converted into an Uppercase. The **Field_Type** is any valid Daytona type and is case sensitive. This includes subtypes (such as **INT(long())**) and **LIST** or **SET** values (such as **LIST[ (0-) STR(25) ]**). (**Field_Type** does not contain the Multiplicity prefix that appears in the Type note in Daytona rcd’s since that is specified here by using the **NULLABLE** option.)

Each field can have zero or more optional parameters in any order. The field can be defined to be **NOT NULLABLE** (no missing values allowed) or allowed to have missing values. (If desired, the word **NULL** can be used instead of **NULLABLE**, which while more common for other DBMS, nonetheless makes less sense.) The field can be defined to be a **PARTITIONER** field which will be used to split the table into different files depending on the values of all the **PARTITIONER** fields. A **MINimum**, **MAXimum**, and/or **DEFAULT** value can be specified for the field. In addition, a **VALIDATION_RE** (regular expression) may be entered. Remember that since an RE is a string, it must be entered as such using double quotes.

**CREATE TABLE** **Table_Name** **[VERSION 'Version_To']** **[APPLICATION Application]** **LIKE** **Table_Name** **[VERSION 'Version_From']**

The **CREATE TABLE LIKE** command will create a new rcd for table **Table_Name** (with optional version **Version_To** in single quotes) by copying the **KEYS** and **FIELDS** from the rcd for Daytona table **Table_Name** (with optional version **Version_From** in single quotes). Both the **Table_Name_From** and **Table_Name_To** parameters will be converted to an Uppercase.

The **APPLICATION** option must be present if the **DS_APPS** environment variable contains more than one application (as in *orders:misc*) and must be one of the applications in the **DS_APPS** environment variable.

The **CREATE TABLE LIKE** command only creates the **KEYS** and **FIELDS** section of the rcd. Before using the table, you must first create one or more **BINS**.

**CREATE TABLE** **Table_Name** **[VERSION 'Version']** **[APPLICATION Application]** **[FILE 'File_Name']** **[DELIM 'd' | DELIMS 'dd']** **[TUPLE_DELIM 't']** **[COMMENT 'c']** **AS** **Sql_Query**

The **CREATE TABLE AS SELECT** command will create a table based on an arbitrary SQL query by creating both the rcd as well as the data file itself, that being the result of the query.

The **Table_Name** will be converted to an Uppercase. The **VERSION** option, if present, must be the first option and must be enclosed in single quotes. The **APPLICATION** option must be present if the **DS_APPS** environment variable contains more than one application (as in *orders:misc*) and must be one of the applications in the **DS_APPS** environment variable. The **FILE** option specifies the file name where the data will be stored. It can be any valid shell expression and must be enclosed in single quotes (such as `'$(ORDERS_DATA:-$ORDERS_HOME)/MYTAB')`.
You can supply your own field separators with the \texttt{DELIM 'd'} or \texttt{DELIMS 'dd'} options. The \texttt{DELIM} option is used as the field separator in the flat DC file used by Daytona and must be exactly one character and enclosed in single quotes. The \texttt{DELIMS} option is used to specify both the field separator and the separator for tuple elements in the flat DC file used by Daytona and must be exactly two characters in single quotes. The \texttt{TUPLE_DELIM} option is used as the tuple field delimiters and must be exactly two characters in single quotes. The \texttt{COMMENT} option is used to specify the comment delimiter character and must be enclosed in single quotes.

The \textit{Sql.Query} can be any valid Daytona DSQL \texttt{SELECT} statement.

\begin{verbatim}
CREATE [UNIQUE] [CLUSTER] INDEX Index_Name ON Table_Name [VERSION 'Version'] ( Field_Name [, Field_Name] ... )
\end{verbatim}

The \texttt{CREATE INDEX} command is used to create an index on a Daytona table \texttt{Table_Name} (with optional \texttt{VERSION Version} enclosed in single quotes).

The index can be optionally unique, cluster, or both. The \texttt{Index_Name} can be either a number or a character string and is case sensitive. The \texttt{Index_Name} must be unique to the \texttt{Table_Name}. The index can be on one or more fields.

Note that this command updates the rcd to add the index and then validates the syntax of the rcd by automatically running

\begin{verbatim}
DS Tracy -gen_fio_for_recs Table_Name
\end{verbatim}

This does not actually build the index files. The index files will be built the next time the table is queried. If you wish to save the first user of the table the time (particularly on a large table or one with many indices), or else wish to validate the index (e.g., you did not accidentally put a unique index on a non-unique field), you should run

\begin{verbatim}
shell DS Sizup -recs Table_Name -ank
\end{verbatim}

from within \texttt{Daisy}.

\begin{verbatim}
CREATE FILE BIN 'Bin_Name' ON Table_Name [VERSION 'Version'] [PARTITION VALUES (value [value] ...)] [SOURCE 'Source'] [DELIM 'd' | DELIMS 'dd'] [TUPLE_DELIM 'tt'] [COMMENT 'c']
\end{verbatim}

The \texttt{CREATE FILE} command is used to add a new \texttt{FILE} description to the rcd for \texttt{Table_Name} (with optional \texttt{VERSION Version} enclosed in single quotes).

If you are adding a \texttt{FILE} bin to a partitioned table, you must specify the \texttt{PARTITION VALUES} option with values for each partioner field in the rcd. The values must be specified in the same order as the partitioning fields in the rcd.

The \texttt{SOURCE} option specifies the directory where the file \texttt{Bin_Name} resides. It can be any valid shell expression and must be enclosed in single quotes (such as \texttt{'${ORDERS_DATA:-ORDERS_HOME}'}).

You can supply your own field separators with the \texttt{DELIM 'd'} or \texttt{DELIMS 'dd'} options. The \texttt{DELIM} option is used as the field separator in the \texttt{FILE} bin \texttt{Bin_Name} and must be exactly one character and enclosed in single quotes. The \texttt{DELIMS} option is used to specify the field separator and separator for tuple elements and must be exactly two characters in single quotes. The \texttt{TUPLE_DELIM} option is used as the tuple field delimiters and must be exactly two
characters in single quotes. The **COMMENT** option is used as the comment delimiter and must be enclosed in single quotes.

All the options after **VERSION** can appear in any order.

After updating the rcd, the fio files are regenerated (since the number of BINS has changed) and recompiled. This also serves to validate the change.

**CREATE** **FILE_INFO** **FILE BIN** 'Bin_Name' **ON** Table_Name [**VERSION** 'Version'] [**SOURCE** 'Source'] [**DELIM** 'd'] [**DELIMS** 'dd'] [**TUPLE_DELIM** 'tt'] [**COMMENT** 'c'] [**FILE_MODEL** [**DELIM** 'd'] [**DELIMS** 'dd'] [**TUPLE_DELIM** 'tt'] [**COMMENT** 'c']]

The **CREATE** **FILE_INFO** **FILE BIN** command is used to add a new **FILE_INFO** **FILE** description to the rcd for Table_Name (with optional **VERSION** Version enclosed in single quotes). Daytona will currently only allow you to add a **FILE_INFO** **FILE** bin if the partitioning data is greater than the last entry in the last **FILE** or **FILE_INFO** **FILE** bin (according to Cymbal data type sorting, not C character sorting).

The **SOURCE** option specifies the directory where the **FILE_INFO** **FILE** Bin_Name resides. It can be any valid shell expression and must be enclosed in single quotes (such as '${ORDERS_DATA:-$ORDERS_HOME}').

You can supply your own field separators for the **FILE_INFO** **FILE** with the **DELIM** 'd' or **DELIMS** 'dd' options. The **DELIM** option is used as the field separator in the **FILE_INFO** **FILE** Bin_Name and must be exactly one character and enclosed in single quotes. The **DELIMS** option is used to specify the field separator and separator for tuple elements and must be exactly two characters in single quotes. The **TUPLE_DELIM** option is used as the tuple field delimiters and must be exactly two characters in single quotes. The **COMMENT** option is used as the comment delimiter and must be enclosed in single quotes.

You can optionally specify a **FILE_MODEL**, a typical description of each of the files listed in the **FILE_INFO** **FILE**. After the **FILE_MODEL**, you can supply your own field separators for each file with the **DELIM** 'd' or **DELIMS** 'dd' options. The **DELIM** option is used as the field separator in each file named in the **FILE_INFO** **FILE** and must be exactly one character and enclosed in single quotes. The **DELIMS** option is used to specify the field separator and separator for tuple elements and must be exactly two characters in single quotes. The **TUPLE_DELIM** option is used as the tuple field delimiters and must be exactly two characters in single quotes. The **COMMENT** option is used as the comment delimiter and must be enclosed in single quotes.

Each of the **FILE_MODEL** parameters, namely the **DELIM**, **DELIMS**, **TUPLE_DELIM**, and **COMMENT** can be different than the similarly named separators in the **FILE_INFO** **FILE**. The **FILE_INFO** **FILE** separators specify the separators in the **FILE_INFO** **FILE**; the **FILE_MODEL** separators specify the separators in each data file named in the **FILE_INFO** **FILE**.

All the options after **VERSION** can appear in any order.

After updating the rcd, the fio files are regenerated (since the number of BINS has changed) and recompiled.

**DROP** **TABLE** Table_Name [**VERSION** 'Version'] [**ERASE**]

The **DROP** **TABLE** command will remove the rcd for the table Table_Name from Daytona (with optional **VERSION** Version enclosed in single quotes). All data files, index files, and
FILE INFO FILES remain intact unless the ERASE option is used, at which point, all files associated with the table are removed.

DROP INDEX Index_Name FROM Table_Name [VERSION 'Version']

The DROP INDEX command will drop the index Index_Name from the rcd for Table_Name (with optional VERSION Version enclosed in single quotes). It will delete all of the associated index files.

DROP FILE BIN 'Bin_Name' FROM Table_Name [VERSION 'Version'] [ERASE]

The DROP FILE BIN command will drop a FILE type BIN from the rcd. The data and index files associated with the FILE bin will not be deleted, unless the ERASE option is given.

If there are no remaining BINS, the fio files are deleted. Otherwise the fio files are regenerated (since the number of BINS has changed) and recompiled.

DROP FILE_INFO_FILE BIN 'Bin_Name' FROM Table_Name [VERSION 'Version'] [ERASE]

The DROP FILE_INFO_FILE BIN command will drop a FILE_INFO_FILE type BIN from the rcd. The data, index, and FILE_INFO_FILE files associated with the FILE_INFO_FILE bin will not be deleted, unless the ERASE option is given.

If there are no remaining BINS, the fio files are deleted. Otherwise the fio files are regenerated (since the number of BINS has changed) and recompiled.

EXAMPLES
Create a new table in aar.orders by partitioning the PART table on Color.

```
cREATE table cpart_part application orders (  
   color STR(25) partitioner,  
   number INT not null,  
   name STR,  
   weight FLT,  
   intro_date DATE(_yyyymmdd_)  
) ;
```

Create a file bin for pink parts

```
cREATE file bin 'CPART_PART.PINK' on cpart_part  
source '$ORDERS_HOME' comment '#' delim ':'  
partition values ('pink');
```

Create a file bin for blue parts. Note that the separators can be different for different partitions.

```
cREATE file bin 'CPART_PART.BLUE' on cpart_part  
source '$ORDERS_HOME' comment '$' delim ',',  
partition values ('blue');
```

Create a unique index on the color partitioned part number.

```
cREATE unique index 1 on cpart_part(number);`
```

Create a (non-unique) index on the color partitioned part name.

```
cREATE index 2 on cpart_part(name);
```
Populate the new CPART_PART table by selecting from the PART table.

```
insert into cpart_part
    (color, name, number, weight, intro_date)
select color, name, number, weight, intro_date
from part
    where color in ("pink", "blue") !/
```

Create a temporary histogram table of order delivery time in aar.orders. Note the use of *order* to distinguish the table name from the SQL keyword.

```
create table hist application orders as
    select (date_recd-date_placed)/10 period, count(*) cnt
    from `order`
    group by date_recd, date_placed !/
```

Build an index on the period column.

```
create index per on hist(period) !/
```

Display the histogram as a graph. Note that Daisy supports the usage of DSQL constructs such as casting, string concatenation via the + operator, and the string multiplier sum(cnt) * "#". Note also the mixture of Daisy and DSQL into a single request.

```
set format = _table_;
select period,
    (:STR:)(period*10) + "-" + (:STR:)(period*10+9) range,
    sum(cnt) * "#" graph
from hist
    group by period
    order by period !/
```

Drop and delete the temporary hist table.

```
drop table hist erase !/
```

Select all data from PART into a file in table format, then return output to screen.

```
set file = /tmp/my_part_table;
set format = _table_;
select * from part;
set file = !/
```

Force the following update command to be processed as DSQL by enclosing it between $[ and ]$. (If it were not forced to be DSQL, Daisy sees the string ; set in the double-quoted string and thinks it is a set command. An incomplete string would then be passed on for further processing generating incorrect error message.)

```
$[ update instructions
    set notes = "check power switch; set dial to minimum"
    where number = 1 ]$ !/
```

Create and run a parameterized query.

```
shell mkdir -p ./Q ./bin;
set shell DS_FLAGS=-E;
set format = _table_;
prepare query limited_part from #{
```
read into INT .m;
select * from part limit to .m #) /!
execute query limited_part using 3/!
exit /!
The results of the following queries are all equivalent.

read into .number;
select * from part where number = (:INT:) .number

read into .number;
select * from part where number = (:FIELD:) .number

read into .number;
select * from part where number = cast(.number as INT)

read into .number;
select * from part where number = cast(.number as FIELD)

read into INT .number;
select * from part where number = .number

SEE ALSO
Archie(1), Sizup(1), Show(1), Synop(1), Tracy(1).
NAME
Archie – Archive manipulation utility

SYNOPSIS
Archie [options]

DESCRIPTION
The archive maintenance tool Archie is a front end for both the UNIX archiver ar(1), and for the Dayton a archiver ds_ar; these are used to maintain groups of application and record class descriptions as a single-file application archive or project descriptions in project archives. Unlike UNIX archive files, Dayton application archives are portable across different platforms and architectures, and they allow long member names (up to 80 characters). ds_ar has the same command syntax and general behavior as ar. Archie provides the same batch services as ar and ds_ar, plus some additional services described below. When run without options or arguments, Archie also leads the user through a series of menus for the available services.

Archives whose names start with `aar.' or `par.' are treated specially by Archie. They are assumed to be Dayton application or project archives, respectively, and it is an error for them to contain any member whose name does not begin with `rcd.', `apd.', or `pjd.'. Application or project archives, if not in the current directory and not specified with a full pathname, will be searched for in the directories listed in the environment variable DS_PATH. By default, application or project archives are assumed to be ds_ar archives, but a different archiver can be specified by setting the variable DS_AR to have the desired archiver name as its value. Users should never change DS_AR. Otherwise, Archie assumes that archives whose names do not start with `aar.' or `par.' are ordinary UNIX ar archives and uses ar to work with them.

Most options have a long and a short form. For example the delete option can be specified by -d or -delete. This is shown below by the notation -d(delete). Most options may take an argument list of archive members. It is an error for such a list to contain an item that is not the name of an archive member. Archie will print an error message and exit with value 101 in such cases.

OPTIONS
-ar(ch) archive_file
Use the specified archive file.

-app application
Use the archive file aar.application.

-proj project
Use the archive file par.project.

-apps for proj applist
When needing to create an initial par.project, include an Apps note in the pjd that lists the given apps.

-U
Print a usage message and exit.

-t{oc} [memberlist]
Print a table of contents of an archive. If memberlist is supplied, only the names of members in memberlist will be printed.

-T{oc} [memberlist]
Print a table of contents of an archive, showing owner, permissions, and modification times. If memberlist is supplied, only information on members in memberlist will be printed.

-d{elete} memberlist
Delete the members in memberlist from the archive.

-p[rint] [memberlist]
Print the contents of each archive member on the standard output. If memberlist is supplied, only members in memberlist will be printed.
User Commands

ARCHIE (1)

± x{tract} [ memberlist ]
Extract a copy of each archive member and place it in the current directory. The archive is not modified. If memberlist is supplied, only members in memberlist will be extracted.

± r{eplace} list [-a{fter} member | -b{efore} member ]
The files named in list are placed in the archive, replacing any existing archive members with the same name. If the file is a new member, it will be placed at the end, or immediately before or after member if a -a or -b suboption is specified. The archive is created if it does not already exist. A warning will be issued when this happens. If the archive is an application archive and no apd.application member is supplied, a rudimentary one will be created automatically. The files named in list are not removed.

± G{et} [ memberlist ]
While temporarily locking the archive, extract a copy of each archive member in the list and place it in the current directory, if the member is not prefixed with a file path component; else place it where the file path component says. The members are then deleted from the archive. If memberlist is NOT supplied, then extract a copy of each archive member into the current directory; in this case, the archive is not modified.

± P{ut} list [-a{fter} member | -b{efore} member ]
While temporarily locking the archive, the files named in list are placed in the archive, replacing any existing archive members with the same name. If the file is a new member, it will be placed at the end, or immediately before or after member if a -a or -b suboption is specified. The archive is created if it does not already exist. A warning will be issued when this happens. If the archive is an application archive and no apd.application member is supplied, a rudimentary one will be created automatically. The files named in list are then removed.

± m{ove} memberlist [-a{fter} member | -b{efore} member ]
Rearrange the archive, moving the members named in memberlist after (if -a is specified) or before (if -b is specified) the member member file. If no -a or -b suboption is provided, the members in memberlist are moved to the end of the archive.

± q{uickly_append} list
The files in list are placed at the end of the archive. It is the responsibility of the user not to use this option when an item in list is the name of an already existing archive member.

EXTENDED OPTIONS
The following options are intended for application archives or project archives only.

± C{heckout} [ memberlist ]
Copies of the members named in memberlist are extracted from either a par or aar and placed by default in the directory containing that archive or else in $DS_ARTMPDIR, if that exists. Each item in memberlist must begin with `rcd.', `apd.', or `pjd.' A lock file is also created in the local directory for each file extracted to prevent any other Archie -C from performing the same operation. This option is intended to provide a protected way to extract a member, modify it, and return it to the archive (with the -R option) without possibility of interference from other users also using -C and -R. If memberlist is not specified, all archive members are extracted into the directory containing the aar.

± R{eturn} memberlist
The indicated memberlist files are returned to the archive and their checked-out copies outside of the archive are removed. This option is intended to restore members that were extracted with the -C option, and it is an error for memberlist to list a member not previously extracted with -C.

± I{ntmodtime} member
Displays time of last modification of the specified member. Time is expressed in UNIX time(2) format, typically an integer count in seconds since 00:00:00 GMT, January 1, 1970.
-S{etmodtime} member [-Time time]
Change the time of last modification of the specified member. If -Time time is not supplied, the modification time will be set to current UNIX time. Time is expressed in UNIX time(2) format. That is, an integer usually count in seconds since 00:00:00 GMT, January 1, 1970.

-N{ewerthan} file –Member member
Compare the time of last modification of the specified file and archive member and return an exit code of 0 if file is newer than member, otherwise return 1.

-V{erify} member
Verify that the archive contains a member named member, and return an exit code of 0 if so, or else 1. No error message is printed if member is not the name of an archive member.

-W{ritable} member
If the archive contains a member named member, then return an exit code of 0 if it is writable by someone, or else 1. An error message is printed if member is not the name of an archive member.

If Archie creates temporary files for commands other than -C and -R, it creates them in /tmp by default, else in $DS_ARTMPDIR if $DS_ARTMPDIR is set and that directory exists.

EXAMPLES
Create an aar for an application named demo using all the rcd files in the current directory:
$ Archie -app demo -r rcd.*

Print a table of contents of what is in an archive:
$ Archie -app demo -t

Display the "application description" file apd.demo for the archive aar.demo:
$ Archie -ar aar.demo -p apd.demo

ENVIRONMENT
DS_PATH
Colon-separated list of directories to search for application or project archives.

DS_AR Archiver to use for application or project archives. Users should never change this.

DS_ARTMPDIR
If set, then informs Archie as to the directory to use for all temporary files.

SEE ALSO
ar(1), DC-rcd(1), Sizup(1).
NAME

Synop – Data dictionary reporting tool

SYNOPSIS

Synop [ -recls [ rlist ] ] [ -pjd ] [ -apd ] [ -toc ] [ -stats ] [ -fields+ ] [ -bins+ ] [ -bins- ] [ -all ] [ -phys_only ] [ -virt_only ] [ -views_only ] [ -proj_only ] [ -usr_only ] [ -proj project ] [ -apps [ alist ] ]

DESCRIPTION

Synop prints synopses of Daytona data dictionaries, displaying the principal information in the form of a concise report. Options control the amount of information printed and the search environment. When invoked with no arguments, Synop prompts the user with a series of questions about what to do. The output format of Synop is intended to be convenient for human perusal, and is subject to revision in later releases as necessary to further this goal. Users wanting to extract data dictionary information for use as input to other tools should write their own backtalk scripts for this purpose, or contact the Daytona support organization.

Selecting Information to Display

Most often, Synop is used to display information about one or more specified record classes by using the -recls option. This can include physical record classes, as defined by rcds in aar files, and virtual record classes, or views, as defined in *.env.cy files. The associated application or project description information (see intro(1)) can be displayed in addition, or instead by using the -apd and -pjd options. The information provided includes the names and types of the fields for each selected table, as well as basic information about the associated data files. For horizontally partitioned tables, the output includes the subclass fields and their corresponding values for each data file. If requested, Synop will include additional information such as key definitions; statistics; additional field information such as integrity constraints and default values; and additional bin information such as the delimiters and the buffer size.

Modifying the Search Environment

By default, Synop uses the environment variables DS_PROJ, DS_APPS, and DS_PATH to determine where to look for rcds, apds, and pjd (see intro(1)). The current project and application list can also be set directly from the command line, overriding the DS_PROJ and DS_APPS variables. Users may specify a project name using the -proj argument, and/or a list of applications using the -apps argument. Any project or application named this way must have a corresponding archive located somewhere in the DS_PATH search path. If just a project name is specified and it is different from the value provided by DS_PROJ, then the application list is taken to be the full list of applications associated with that project. Otherwise, the list of applications will be taken from the DS_APPS variable. If both a project name and a list of applications is given, the applications must be among those associated with the project. If no project is named, but the user requests a project description (via the -pjd argument, described below) then DS_PROJ must be set to a valid project name, and any applications specified using the -apps argument must belong to that project. Finally, if no project is named and the user does not request any project description, the user may supply an arbitrary application list (subject of course to the DS_PATH restriction above).

OPTIONS

-recls [ rlist ]
Print a synopsis of the record class description (rcd) for each record class in rlist. The list members may be separated by colons or spaces. If rlist is omitted, print a synopsis of the rcd for every record class in each current application.

-pjd
Print a synopsis of the project description for the current project.

-apd
Print a synopsis of the application description for each application in the current search environment.

-toc
Print only the names of the record classes found. No detailed information will be returned. If -apd is also specified, then the application name will be printed at the appropriate time; likewise for -pjd.
-keys Include information about what keys and indices are defined, in the synopsis of each record class description.

-stats Include index statistics if key information is being printed. If no key information is being printed, then this option is ignored.

-fields+ Print additional information about the record class fields including allowable minimum and maximum values, default values, and other information if it is included in the rcd file(s).

-bins+ Print all available information about the record class BINS including data buffer sizes and max open bins if specified in the rcd as well as information on every FILE described in associated FILE_INFO_FILES.

-bins- Print no record class BINS information.

-all Print all RECORD_CLASS information. This is the same as specifying all of -bin+, -field+, -keys, -stats.

-phys_only Print information only for physical record classes (i.e., those defined by rcds in aar files).

-virt_only | views_only Print information only for views, which are virtual record classes.

-proj_only Print information only for the current project, excluding any information that is application-specific.

-usr_only Print information only from usr.env.cy.

-proj project Use project as the current project name. Only applications belonging to project will be searched.

-apps [alist] Use alist as the list of current applications, in which to search for information. The list members may be separated by colons or spaces. If alist is provided and a -proj project argument is also given, the list members must be among the applications associated to project.

EXAMPLES
Print a synopsis of rcd.PART_ from the orders application with the understanding that SDS_APPS contains orders.

$ Synop -recls PART_
The output is:

-----------------------------------------------------
RECORD_CLASS: PART_
Desc Found In: /home/user1/d/aar.orders
-----------------------------------------------------

FILE: PART_
Source (from apd): ${ORDERS_DATA:-$ORDERS_HOME}
FIELDS:

  Field 1:   INT       Number
  2:   STR(25)   Name
  3:   STR(25)   Color
  4:   FLT       Weight

Daytona Last change: 16 June 2009
Print a synopsis of the $rcd$ files for the record classes $SUPPLIER$, $PART$, and $ORDER$ for the application $orders$:

$ Synop -apps orders -recls SUPPLIER PART ORDER$

Print a synopsis of the $apd$ file and each $rcd$ file for the application $orders$, providing all available key and index information and statistics:

$ Synop -apps orders -apd -recls -keys -stats$

Print a synopsis of the application descriptions for the applications $orders$ and $misc$:

$ Synop -apps orders misc -apd$

Print out a list of view names interspersed with their appropriate app/proj:

$ Synop -recls -virt_only -apps orders:misc -toc -apd -pjd$

ENVIRONMENT

$DS\_PATH$

Colon-separated list of directories to search for project and application archives.

$DS\_PROJ$

Specification of the current project, if any; used to locate the appropriate project description file.

$DS\_APPS$

Colon-separated list of current applications.

SEE ALSO

intro(1), Archie(1), DC-rcd(1), Sizup(1).
NAME

Synop_fpp – Function/Procedure/Predicate (FPP) reporting tool

SYNOPSIS

Synop_fpp

[ -proj project ]
[ -app(s) applist ]
[ -name(s) fpplist ]
[ -toc | -compact | -full ]
[ -fun_only | -proc_only | -pred_only ]
[ -sys_only | -proj_only | -usr_only ]
[ -add_imports ]

DESCRIPTION

Synop_fpp prints synopses of functions/procedures/predicates (FPPs) found in the environment, displaying the principal information in the form of a concise report. Options control the amount of information printed and the extent of the search. When invoked with no arguments, Synop_fpp prompts the user with a series of questions about what to do. The output format of Synop_fpp is intended for both third-party tools and human perusal.

Selecting Information to Display

Most often, Synop_fpp is used to display compact (or prototype-like) information about one or more FPP definitions as determined by the -name(s) argument. When this argument is not given, then no name restriction will apply and all FPP names will be searched. In addition to the compact style, given by default or with the -compact argument, output can be limited to a list of names using the -toc argument or to a more complete, human-readable form using the -full argument. Information provided includes modifiers such as the C_external and overloaded keywords, argument types and names for all FPPs, and the return type if applicable.

Modifying the Search Environment

By default, Synop_fpp uses the environment variables DS_PROJ, DS_APPS, and DS_PATH to determine where to look for FPPs. The current project and application list can also be set directly from the command line, overriding the DS_PROJ and DS_APPS variables. Users may specify a project name using the -proj argument, and/or a list of applications using the -app(s) argument. Environment and package files for a project or applications named this way must be located somewhere in the DS_PATH search path. If just a project name is specified, the application list is taken to be the full list of applications associated with that project. If both a project name and a list of applications is given, the applications must be among those associated with the project.

OPTIONS

-proj project
Use project as the current project name. Only applications belonging to project will be searched.

-app(s) applist
Use applist as the list of current applications, in which to search for information. The list members may be separated by colons or spaces. If a -proj project argument is also given, the list members must be among the applications associated with project.

-name(s) fpplist
Print a synopsis of FPPs that match any of the patterns (Cymbal-style regular expressions) given in fpplist. The list members may be separated by colons or spaces. Note that the pattern "sum_two_nbrs" will match any FPP containing that substring, such as "sum_two_nbrs_int" and "sum_two_nbrs_flt". For an exact match, use the pattern "sum_two_nbrs". If this argument is omitted, print a synopsis for all FPPs found.

-toc
Print only the names of the FPPs found. No detailed information will be returned.

-compact
Print a compact form of the FPPs found, including modifiers, argument types and names, and return types if applicable.

- **full**  
  Print a full, human-readable report for each FPP found.

- **fun_only**  
  Print information only for functions.

- **proc_only**  
  Print information only for procedures.

- **pred_only**  
  Print information only for predicates.

- **sys_only**  
  Print information only for FPPs from `sys.env.cy`.

- **proj_only**  
  Print information only for FPPs from the project environment file.

- **usr_only**  
  Print information only for FPPs from `usr.env.cy`.

- **add_imports**  
  Print information on imported FPPs as well.

**EXAMPLES**

Print a synopsis of `Is_A_Descendant_Of` from the `orders` application with the understanding that `$DS_APPS` contains `orders`.

```bash
$ Synop_fpp -name Is_A_Descendant_Of
```

The output is:

```bash
# FILE orders.env.cy (/home/loginid/d)
path PRED: Is_A_Descendant_Of[TUPLE[STR .x], TUPLE[STR .y]]
```

A full listing can also be given for the previous example.

```bash
$ Synop_fpp -name Is_A_Descendant_Of -full
```

The output is:

```
PRED: Is_A_Descendant_Of
Defined In: /home/loginid/d/orders.env.cy
```

**MODIFIERS:** path declarative

**POSITIONAL ARGS:**

```
TUPLE[STR(*) .x]
TUPLE[STR(*) .y]
```

Print a synopsis of the FPP definitions matching `Ancestor` and `Descendant` for the application `orders`:

```bash
$ Synop_fpp -app orders -names Ancestor Descendant
```

Print the names of the FPP definitions for functions in the applications `orders` and `misc`:

```bash
$ Synop_fpp -apps orders misc -fun_only -toc
```

**ENVIRONMENT**

**DS_PATH**

Colon-separated list of directories to search for project and application archives.

**DS_PROJ**
Speciation of the current project, if any.

**DS_APPS**
Colon-separated list of current applications.

SEE ALSO
intro(1), Synop(1).
Name
ar_of, ar_or_env_fl_for, env_fl_for_fpp, ds_whence

SYNOPSIS
ar_of record_class [ version ]
ar_or_env_fl_for record_class [ version ]
env_fl_for_fpp fpp_name
ds_whence file search_path=${DS_PATH}

DESCRIPTION
The set of find-it tools are utilities that provide a search capability for a given Daytona environment, as defined by the contents of the DS_DIR, DS_PATH, DS_PROJ, and DS_APPS shell variables.

The ar_of command will find the aar or par file that defines the provided record class. An optional version argument may also be given to refine the search.

The ar_or_env_fl_for command is similar to ar_of, in that it can be used to locate a record class definition. It can additionally be used to locate view definitions since it also searches Cymbal environment files, *.env.cy, as well as aar and par files. An optional version argument may also be given to refine the search.

The env_fl_for_fpp command will find the *.env.cy file that defines the provided fpp (function/procedure/predicate).

The commands ar_of, ar_or_env_fl_for, and env_fl_for_fpp return a rooted, expanded file path when the provided argument is found. If it cannot be found, an error is generated.

The ds_whence command will find the directory in the search_path argument ($DS_PATH by default) that contains the provided file name.
Name

Show_Sem / Rm_Sem

SYNOPSIS

Show_Sem { -nt DATE_CLOCK | -ot DATE_CLOCK | -u user }*
Show_Sem { hex_key | int_semid }*

Rm_Sem { -nt DATE_CLOCK | -ot DATE_CLOCK | -u user }+
Rm_Sem { hex_key | int_semid }+

DESCRIPTION

The Show_Sem command is used to print information about semaphores. When no arguments are given, the command will identify the:

- SEM KEY
- SEMID
- OWNER
- CREATE/CHANGE TIME
- #SEMS (number of semaphores)
- LAST SEMOP TIME

of all known semaphores (excluding those owned by root). Information is collected from the ipcs command and from semctl(2). Any unavailable information will be displayed as N/A.

Arguments can be provided to constrain these results. These arguments can be either dash options (such as -u user) or explicit semaphore identifiers (such as key or semid), but not both.

Rm_Sem accepts the same arguments as Show_Sem but requires at least one constraining argument. Rm_Sem will (attempt to) remove the corresponding semaphores.

OPTIONS

-nt DATE_CLOCK
  Display only those semaphores whose create/change times are newer than or equal to the given DATE_CLOCK.

-ot DATE_CLOCK
  Display only those semaphores whose create/change times are older than or equal to the given DATE_CLOCK. This option also recognizes "now" as an alternative to a DATE_CLOCK.

-u user
  Display only those semaphores belonging to user.

hex_key
  Display semaphores with the given hex_key (e.g., 0x0).

int_semid
  Display semaphores with the given int_semid (e.g., 553648130).

EXAMPLES

$ Show_Sem -nt 2011-06-01@12:00:00
SEM KEY SEMID OWNER CREATE/CHANGE TIME #SEMS LAST SEMOP TIME
------- ----- ----- ------------------ ----- ---------------
0x0 553648130 rxga 2011-06-06@16:02:21 1 N/A

$ Rm_Shmem 553648130
removing key=0x0, id=553648130
User Commands

SHOW

SYNOPSIS

Show_Shmem { ± nt DATE_CLOCK | ± ot DATE_CLOCK | ± u user }=
Show_Shmem { name | hex_key | int_shmid }=

Rm_Shmem [-f] { ± nt DATE_CLOCK | ± ot DATE_CLOCK | ± u user }+
Rm_Shmem [-f] { name | hex_key | int_shmid }+

DESCRIPTION

The Show_Shmem command is used to print information about shared memory segments. When no arguments are given, the command will identify the:

- SHMEM KEY
- SHMID
- OWNER
- SEGSZ(K)
- CREATE/CHANGE TIME
- #ATT (number of attached)
- LAST ATTACH TIME
- LPID (pid of last shmat/shmdt)
- NAME

of all known shared memory segments (excluding those owned by root). Information is collected from the ipcs command, from shmctl(2) and from the SHMEM_SEG table. Any unavailable information will be displayed as N/A (or as an empty entry in the case of NAME).

Arguments can be provided to constrain these results. These arguments can be either dash options (such as ± u user) or explicit shared memory segment identifiers (such as name, key, or shmid), but not both.

Rm_Shmem accepts the same arguments as Show_Shmem but requires at least one constraining argument. Rm_Shmem will (attempt to) remove the corresponding shared memory segments (and their entries in the SHMEM_SEG table). Furthermore, if any of the dynara in the named region are concurrent, then it will also remove an associated semaphore as well as kill the associated Legate.

OPTIONS

- ± f Force option: Remove shared memory segment(s) even when number of attached is non-zero.
- ± nt DATE_CLOCK
  Display only those shared memory segments whose create/change times are newer than or equal to the given DATE_CLOCK.
- ± ot DATE_CLOCK
  Display only those shared memory segments whose create/change times are older than or equal to the given DATE_CLOCK. This option also recognizes "now" as an alternative to a DATE_CLOCK.
- ± u user
  Display only those shared memory segments belonging to user.
- name
  Display shared memory segments with the given name (as defined in the SHMEM_SEG table).
- hex_key
  Display shared memory segments with the given hex_key (e.g., 0x1006d92).
- int_shmid
  Display shared memory segments with the given int_shmid (e.g., 50331711).

EXAMPLES

$ Show_Shmem ±nt 2011-06-01@12:00:00
### User Commands

**Show_Shmem(1)**

<table>
<thead>
<tr>
<th>SHMEM KEY</th>
<th>SHMID</th>
<th>OWNER</th>
<th>SEGSZ(K)</th>
<th>CREATE/CHANGE TIME</th>
<th>#ATT</th>
<th>LAST ATTACH TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1006d92</td>
<td>50331711</td>
<td>rxga</td>
<td>2052</td>
<td>2011-06-02@10:43:36</td>
<td>0</td>
<td>2011-06-02@10:43:36</td>
</tr>
</tbody>
</table>

$ Rm_Shmem 50331711
removing key=0x1006d92, id=50331711
NAME
Distribute_Cmds – parallelizes a file of commands via job distribution

SYNOPSIS
Distribute_Cmds {nbr_clones} [ -d ] [ -as_daemon ]

Add_Cmds_To [ -d ] file_of_cmds

DESCRIPTION
Distribute_Cmds will take a list of commands from a file (or from stdin if not running as a daemon) and will parallelize those commands across the specified number of subprocesses (or clones). If there are more commands than clones, then a command will be distributed to each clone and, as each command is completed, the next command will be passed to that clone (until no commands are left).

If Distribute_Cmds is not run as a daemon and there are fewer commands than clones, then the total number of clones will be reduced. When run as a daemon (using -as_daemon), Distribute_Cmds will continue to read and run commands from the specified file_of_cmds as they become available. The most recent list of commands read from the input file will be stored as file_of_cmds.prev. Don’t worry, when running as a daemon, Distribute_Cmds will not allow the input file of commands to grow indefinitely simply because when it reads a bunch of commands to work on, it empties out the file to allow for new ones to be added. Add_Cmds_To is used to add new commands to the file_of_cmds.

Putting ./Pause_Distribute_Cmds in the directory where Distribute_Cmds is running will cause it to cease issuing work temporarily (resuming only when that file is removed).

Putting ./Stop_Distribute_Cmds in the directory where Distribute_Cmds is running will cause it to cease issuing work altogether. Any commands read but not yet issued will be stored as file_of_cmds.todo and will be the first commands processed when Distribute_Cmds is restarted (with the same file_of_cmds).

You can also just kill the lot by using psme to find the process group id <pgid> and then issuing "kill -TERM -<pgid>".

The Cymbal source for Distribute_Cmds.cy and Add_Cmds_To.cy is included in $DS_DIR.

EXAMPLES
$ Distribute_Cmds 3 - <<!
(du -s -k /bin; exit 0)
(du -s -k /etc; exit 0)
(du -s -k /lib; exit 0)
(du -s -k /opt; exit 0)
(du -s -k /sbin; exit 0)
(du -s -k /tmp; exit 0)
(du -s -k /usr; exit 0)
(du -s -k /var; exit 0)
!
1 /bin
32153 /lib
1488 /sbin
76740 /etc
180224 /tmp
7123611 /var
4235945 /opt
9157251 /usr
$ Distribute_Cmds 3 commands_file -as_daemon &
$ Add_Cmds_To - commands_file <<!
  (du -s -k /bin; exit 0)
  (du -s -k /etc; exit 0)
!
fyi: Distribute_Cmds: just loaded 2 command(s)
  1  /bin
  76740  /etc
$ touch Stop_Distribute_Cmds
fyi: Distribute_Cmds: stopping for ./Stop_Distribute_Cmds

##########
// Cymbal illustrating how to add a command from within a Cymbal program
set .ochan =
  new_channel( for "my_cmd_file" with_mode _append_share_ with_locking )
otherwise
  with_msg "error: cannot append commands to my_cmd_file" do Exit(3);

to .ochan do Write_Line( "Sizup -rec SUPPLIER -m" );
do Close( .ochan );

SEE ALSO
  intro(1), Checkup(1), Sizup(1).
NAME

pdq - A server for Polyclient Daytona Queries

SYNOPSIS


DESCRIPTION

pdq is a Dayton DBMS Server for communicating over a TCP network with a variety of clients via an IP address and port number. The classes of clients include:

- The Java JDBC API (via package com.att.research.jdbc.daytona)
- The Perl DBI DataBase Interface (via perl module DBD::Daytona)
- Cymbal client(s) for querying in a networked environment with one or more Daytona DBMS servers or for testing pdq (e.g., see pdc)

Each pdq process is a lightweight server for a particular Daytona project, as defined by the Daytona environment in which the pdq server is running (see environment variables DS_PROJ and DS_APPS and the command DS Env).

Each pdq server for a particular Daytona project must listen for client requests on a unique port. The executable for pdq is in directory $DS_SERVERDIR/pdq/bin. The Daytona installation directory $DS_DIR contains an original $DS_SERVERDIR. Let’s say that dba0 is the person who will be installing pdq. Since dba0 may or may not be the person/login who installed Daytona on the machine and since dba0 will probably not want their pdq installation to be destroyed when a new version of Daytona is installed, dba0 should copy the original $DS_SERVERDIR to a different directory which is called, say, $MY_SERVERDIR. Likewise, since pdq gets its authentication/authorization capability by calling Arthur as a co-process, dba0 should create their own $MY_ARTHURDIR by copying the original $DS_ARTHURDIR

Here are ksh commands that would create these directories and further, initialize the shell environment:

rm -fr $MY_SERVERDIR $MY_ARTHURDIR
cp -r $(getdsenv DS_SERVERDIR) $MY_SERVERDIR
cp -r $(getdsenv DS_ARTHURDIR) $MY_ARTHURDIR
. $DS_DIR/DS_Set "DS_SERVERDIR=$MY_SERVERDIR DS_ARTHURDIR=$MY_ARTHURDIR"

Alternatively, and probably better, one can use the shell command scripts install.pdq and install.arthur to create $MY_SERVERDIR and $MY_ARTHURDIR. The distinction and advantage is that the scripts will automatically link the relevant executables to those under $DS_DIR; that way when Daytona is upgraded, the executables under $MY_SERVERDIR and $MY_ARTHURDIR will be too. Otherwise, someone needs to remember to do the cp’s again after the upgrade, which could be disruptive to ongoing operations unless handled carefully.

The indices for Arthur tables need to be created using this script:

( cd $MY_ARTHURDIR
  . bin/setup.arthur
  DS Resync -i - ; )

Customizing Arthur for pdq’s use and so as to support JDBC can be done using a script like:

( cd $MY_ARTHURDIR ; . bin/setup.arthur ;

Daytona        Last change: 21 February 2012
bin/Arthur \!
CLEAR_TABLE AUTHORIZATION
  
CLEAR_TABLE ROLE_SPEC
  
DELETE TARGET PDQ_CMD
  
DELETE TARGET -- pdq_exec
  
DELETE TARGET_KIND pdq_cmd
  
DELETE TARGET_KIND pdq_exec
  
INSERT TARGET_KIND pdq_cmd DS_ECHO|DS_PING|DS_FPP|DS_QQ|DS_COMPILE
  
AUGMENT TARGET_KIND pdq_cmd DS_TABLE_INFO|DS_DDL|DS_ARTHUR_REQUEST
  
INSERT TARGET_KIND pdq_exec DS_EXECUTE|DS_CAT|DS_SHOW
  
INSERT TARGET PDQ_CMD _arthur_ pdq_cmd -
  
REQUEST _arthur_
define role pdq_user to include actions all on PDQ_CMD
  
DELETE AGENT a_jdbc_client
  
INSERT AGENT a_jdbc_client a_passwd jdbc 0 localhost|mypc.att.com @
  
REQUEST _arthur_
grant role pdq_user to a_jdbc_client
  
!)

Most of the above customization can be taken to be boilerplate, i.e., constant; dba0 may however wish
to change the AGENT's name 'a_jdbc_client' and of course the password 'a_passwd' should be
changed; furthermore, other details of the authorization such as the authorized client machines and times
of access are available for modification. Keep in mind that Arthur never grants any permissions by
default. Invoke "DS Man Arthur" for more information.

A pdq client may run on the same machine as the pdq server it wishes to work with. In this case, from
the client's perspective, pdq is listening on localhost, a synonym for internet host IP address
127.0.0.1. The convention is to run the executable pdq in directory
$DS_SERVERDIR/pdq/local for localhost 127.0.0.1 requests (see -host argument below).
Temporary subdirectories and files for client requests are created there under qworkdir.

Here for example is how to start up pdq listening on localhost:

cd $MY_SERVERDIR/pdq/local
rm -f pdq.log nohup.out
../bin/start.pdq -trace 3 -port 16776

The corresponding pdq.log contains pdq's running log. pdq can be terminated at will just by issuing a
'kill -TERM <pid>' command: the log is one place to find that <pid>; pgrep(1) is another.
For external network requests, run the executable `pdq` in directory `$DS_SERVERDIR/pdq/net` (see the `-host` and `-port` arguments below). Temporary subdirectories and files for network client requests are created there under (another) `qworkdir`.

**OPTIONS**

`-host host_ip`

Specifies the internet hostname or IP address (in octet form) at which `pdq` is to listen for client requests. As a default, `pdq` uses the IP address `127.0.0.1` with the Daytona sample project `daytona` database.

The server would also allow `localhost` for `127.0.0.1` (or `trntwn.acme.org`, say, for `63.251.83.40`).

Another option is to use the `0.0.0.0` wildcard address which will cause the server to listen on every network interface on a multi-homed machine.

`-port port_nbr`

Specifies the port on which `pdq` is to listen for client requests. `pdq` requests that the port number be over `12500`. As a default, `pdq` uses the port number `14015` with the Daytona sample project `daytona` database.

Pick a unique port for your own database.

`-secure`

Specifies that `pdq` should operate in secure communication mode, where interactions with a client will be encrypted. Connections will only be accepted from clients that are also running in secure communication mode.

`-auth_query`

Specifies that `pdq` should validate table and field access privileges (authorized via Arthur) when compiling user queries.

`-max_conns conn_nbr`

Specifies the maximum number of connections `pdq` can handle at one time. Default is `10`.

`-max_cpu cpu_secs`

Specifies the maximum CPU time in seconds for a user executable. Default is system-dependent (but typically unlimited).

`-max_outfile outfile_kbytes`

Specifies the maximum output file size in K for a user executable. Default is system-dependent (but typically unlimited).

`-max_vmem vmem_kbytes`

Specifies the maximum virtual memory usage in K for a user executable. Default is system-dependent (but typically unlimited).

`-save_faults`

Specifies that `pdq` should save a copy of the session directory for each failed user request. By default, no failure information is preserved. This option can be toggled for future sessions by sending `pdq` a `USR1` signal.

`-trace level`

Generate trace or log information on client database request. `0` means no trace information whereas `3` generates the most. Handy when implementing a new client for `pdq` to see the messages that are being exchanged. See the `pdq` tester Poly Daytona Client (`pdc`).

`-ver{sion}`

Output the version information for `pdq` and exit.

`+?`

Output usage (help) information for `pdq` and exit.
PROTOCOL

pdq uses a simple message-based protocol to communicate over a TCP channel with a client. Client command messages begin with the characters DS_command .... Both client and server messages are terminated by a line consisting of the 5-character string #####.

The pdq server reads an incoming DS_command (with its optional arguments and any argument text up to the ##### line) and passes it on to Daytona for processing. Output is sent back to the client (as ASCII characters) line-by-line and terminated with a ##### line.

A brief (but polite) client communication with the pdq server consists of three client requests:

1. A DS_OPEN command,
2. A normal DS_command request, and
3. A DS_CLOSE command.

See the first example in the EXAMPLES below. The command DS_OPEN tells the pdq server that this client (with the client_class and user_id given) wishes to start a session. Possible client classes include jdbc, pdbi, odbc, and cym. The command DS_CLOSE tells the pdq server that this client has no more requests on this connection; the server is free to close the current connection and to listen for and handle new connections from new clients with new requests. pdq sends no reply in response to a DS_CLOSE command.

Polite clients remember to issue:

```
DS_CLOSE
#####
```
as soon as possible.

Details on opening a communications channel to the pdq server vary from client class to client class. For the JDBC, ODBC, and Perl DBI clients, please consult their respective Daytona and/or third-party documentation and look at their Daytona test suites (under $DS_DIR):

- Java JDBC: $DS_JAVADIR/jdbc
- MS ODBC: $DS_ODBCDIR/odbc
- Perl DBI: $DS_PERLDIR/pdbi

COMMANDS

The syntax of Daytona Server commands for pdq is:

```
DS_command [ arguments ... ]
[ text ... ]
#####
```

The arguments are optional, depending on the specific DS_command. One or more lines of text may also be present, for example, the text of a query. The line ##### denotes the end of the DS_command and is required in all cases.

- DS_OPEN client_class user_id
password
#####

Opens a client session (for the given client_class and user_id). Recognized client_classes include: cym (Cymbal), jdbc (Java JDBC), odbc (Microsoft ODBC), and pdbi (Perl DBI); the latter three use their own particular client request syntax for data-dictionary information to conform to their own expectations and standards. Each of client_class, user_id, and password are required. User authentication is provided by Arthur.

Upon successful completion of the request, pdq informs the client by sending the message:

```
SUCCESS
#####
```

Daytona Last change: 21 February 2012
- **DS_ECHO**  
  arg  
  ######  
  Echo the value of the *arg* back to the client. *arg* may be any shell variable in the *pdq* server’s environment, such as `$DS_PATH` or `$DS_APPS`. ‘back quotes’ are not allowed.

- **DS_PING**  
  ######  
  Echo a 1-line ping response from the *pdq* server back to the client of the form:  
  `PDQ on host (ip_addr:port_nbr) alive at date`  
  ######  
  An example of a message is:  
  `PDQ on trntwn001 (63.251.83.40:14016) alive at Sat Feb 2 14:02:02 EST 2002`  
  ######  
  See also the first example in EXAMPLES below.

- **DS_FPP**  
  [ type=category ]  
  ######  
  Echo a comma (jdbc), colon (odbc), or blank (pdbi) -separated list of generally available Cymbal functions, predicates, and procedures (fpps) in one of the following type categories: common, clock, date, date_clock, ip, misc, num, str, sys, time, and time_date.

  For example, the command  
  `DS_FPP type=sys`  
  ######  
  for client class jdbc, instructs *pdq* to send back a comma-separated list of available system fpps.

- **DS_QQ**  
  [ fmt=val [ arg ... ] ]  
  **Cymbal/DSQL Query Text**  
  ######  
  The **Cymbal/DSQL Query Text** is sent to the *pdq* host machine for quick compilation and execution with the query output delivered back to the client.

  Note that the optional arguments for **DS_QQ** correspond to those for **DS_SHOW**, below; they do not pertain to run-time query arguments (see **DS_EXECUTE** below).

  Handy for simple queries as shorthand for the three command **DS_COMPILE**, **DS_EXECUTE**, and **DS_SHOW**. See EXAMPLES below.

- **DS_COMPILE**  
  query_name [ hold | release ]  
  [ **Cymbal/DSQL Query Text** ]  
  ######  
  If provided, the **Cymbal/DSQL Query Text** is stored on the *pdq* host machine in the file *query_name*.Q (if not provided, the server will attempt to use an existing file of the same name). The executable is saved as *query_name*. The query and executable can be made persistent, or kept across sessions, by adding the *hold* keyword to the **DS_COMPILE**. The *release* keyword, indicating that the query should only exist for the current session, is the default.

  Note that *query_name* is required. Clients may generate simple default query names by prepending R_ to *user_id*.

  Upon successful completion of the compilation, *pdq* informs the client by sending the message:  
  `SUCCESS`  
  ######  
  To execute the query, use the command **DS_EXECUTE**; to concatenate (send) the query output to the client, use the command **DS_CAT**.
− DS_EXECUTE  \\
     query_name  [ arg ... ] \\
     ######
Execute the compiled query named query_name, passing in any optional command-line argument(s) specified in [ arg ... ]. Use 'single quotes' to protect white space or other special characters in any argument(s).

Upon successful completion of the execution, pdq informs the client by sending the message: SUCCESS
     ######

Output generated by executing the query is stored in the file query_name.out. To concatenate (send) the query output to the client, use the command DS_CAT.

Note that, if the query_name argument is of the form user_id2:query_name2, then DS_EXECUTE will execute the compiled query named query_name2 belonging to the user user_id2. Privileges to run the executable must have been granted by user_id2 (see DS_ARTHUR_REQUEST below) and will be validated using Arthur.

− DS_CAT  query_name  \\
     ######
Concatenate (send) the query output in the file query_name.out to the client; a synonym for DS_SHOW with the default unformatted data output; typically used to communicate with client database packages. See EXAMPLES below.

− DS_SHOW  query_name  [ fmt=val  [ arg ... ] ]  \\
     ######
Show (send) the query output in the file query_name.out to the client.

Recognized output fmt format values are:

− table: t/table/  
  PRN fixed-width field format; see DC-prn man page.
− description: d/description/  
  use the Cymbal description format; see DC-pkt -T man page.
− packet: p/packet/  
  PKT packet format of name=value pairs; see DC-pkt and DC(5) man pages.
− xml: x/xml/  
  XML Extensible Markup Language tag format; see DC-pkt -X man page.
− unformatted: u/unformatted/  
  DC Daytona data format; a variation of the UNIX flat file format used by Daytona to store data; may contain %msg... comment records to annotate the data; see DC(5) man page. The default is equivalent to DS_CAT.

For the optional argument(s) [ arg ... ], see the respective DC-prn or DC-pkt man page. See also EXAMPLES below.

− DS_TABLE_INFO  type  [ attr=val ... ]  \\
     ######
This command provides metadata (information about the database or from the data dictionary) in a tabular format resembling the output of a Daytona query. It supports the metadata gathering functions embedded in Daytona drivers for Java JDBC, Microsoft ODBC, and Perl DBI clients. backtalk scripts are used to tune the output to the expectations and needs of the aforementioned clients. Other clients may have their own uses for this command.

Recognized types of information include:

− environment: env
output a 2-column table of pdq’s DS_ environmental shell variables and their values.

– application: `app`
  output a 1-column table of application names taken from $DS_APPS.

– table or record class:
  `table [app=apat] [rec=rpat] [type=tpat]`
  `apat` and `rpat` may be application or record class names or patterns that use backtalk (see UNIX® `egrep(1)`) regular expressions or the SQL wildcard characters `%` (matches any string of 0 or more characters) or `_` (matches any single character);
  `tpat` describes a table type, such as a normal TABLE, STDIN, STDOUT, PIPE, PIPEIN, or PIPEOUT, and may also contain regular expression or wildcard characters.

– key or index:
  `key [app=apat] [rec=rpat] [type=index|primary] [uniq=true|false]`
  `apat` and `rpat` may be application or record class names or patterns that use backtalk (see `egrep(1)`) regular expressions or the SQL wildcard characters `%` (matches any string of 0 or more characters) or `_` (matches any single character);
  `type` may be set to either `index` or `primary`;
  `uniq` may be set to either `true` or `false`.

– field:
  `field [app=apat] [rec=rpat] [col=cpat]`
  `apat`, `rpat`, and `cpat` may be application, record class, or column names or patterns that use backtalk (see `egrep(1)`) regular expressions or the SQL wildcard characters `%` (matches any string of 0 or more characters) or `_` (matches any single character).

– executable:
  `exec [exec=epat]`
  `epat` may be an executable name or pattern that uses backtalk (see `egrep(1)`) regular expressions or the SQL wildcard characters `%` (matches any string of 0 or more characters) or `_` (matches any single character).

– SQL input/output parameter types for executables:
  `exio [exec=epat] [col=cpat]`
  `epat` and `cpat` may be executable or column names or patterns that use backtalk (see `egrep(1)`) regular expressions or the SQL wildcard characters `%` (matches any string of 0 or more characters) or `_` (matches any single character); for more about SQL IN, OUT, and INOUT parameters, see Java JDBC API, p. 410, or SQL in a Nutshell.

– `DS_ARThUR_REQUEST`

  **Arthur Query Text**

  The *Arthur Query Text* is passed through to Arthur, with the query output delivered back to the client.

  A typical use of `DS_ARThUR_REQUEST` is for a user to grant/revoke privileges for other users to access its query executables. See EXAMPLES below.

– `DS_DDL`

  **SQL DDL Text**

  The *SQL DDL Text* (e.g., CREATE TABLE, DROP TABLE) is sent to the `pdq` host machine where it will be compiled and executed by the Daisy SQL DDL processor.

  Upon successful completion of the execution, `pdq` informs the client by sending the message: SUCCESS

  ####
DS_CLOSE

Tells the pdq server that this client has no more requests on this connection; the server is free to close the current connection and to listen for and handle new connections from new clients with new requests. pdq does not send any response to this command back to the client. Polite clients remember to issue DS_CLOSE as soon as possible.

ERRORS

pdq logs errors in the file pdq.log. It reports errors to clients by sending error messages of the form: 

Error Message (1 or more lines)

Errors or warnings concerning query or table output are sent to the client in an output message header of the form:

Warning Message

or

Warning Message

Warnings may indicate that no information was found, i.e., there are no rows in an empty (result) table.

STARTING PDQ

To run some Daytona-specific examples, the client needs to communicate with a (possibly remote) Daytona Server pdq on a known port. Sometimes for testing there is a Daytona Server pdq running on localhost IP address 127.0.0.1, on port 14015; typically it is using the default ‘daytona’ sample database that ships with Daytona. To check if it is running, use the command:

myhost:> $(getdsenv DS_SERVERDIR)/pdq/bin/ps.pdq
meme 323232 323131 .... pdq -host 127.0.0.1 -port 14015

If not, see the Daytona pdq man page documentation to start up a pdq server for your Daytona database data of interest, e.g., for the sample project ‘daytona’ or for your own database using a unique port number. Basically you start pdq in a Daytona environment that knows about your database according to the settings described in the output of the command DS Env. Create a convenient working directory for pdq, perhaps one for local service and another for network service, cd to the appropriate directory, and start pdq using:

myhost:> cd myPDQ/local
myhost:> PATH=$(getdsenv DS_SERVERDIR)/pdq/bin:$PATH
myhost:> rm -f nohup.out pdq.log
myhost:> nohup pdq -trace 3 -host 127.0.0.1 -port 16776 &
myhost:> cat nohup.out

PDQ 2.0:2003-04-01 START: 2003-07-04@12:00:00 PID: 12345

...
On Linux, it may be necessary to open up certain ports on the machine with the pdq server using a command like:

```
iptables -I INPUT -p tcp --jump ACCEPT --tcp-flags SYN,RST,ACK SYN -s 192.168.1.20 --dport 14015:14025
```

**EXAMPLES**

Note the use of ‘single quotes’ to protect white space when a client passes arguments to the server. *Polite clients remember to issue DS_CLOSE as soon as possible.*

Here is a brief polite conversation between client and server:

- **Open** connection request to server:
  ```
  DS_OPEN jdbc jtest
  anotyad
  ####
  ```

- **Response** from server:
  ```
  SUCCESS
  ####
  ```

- **Request** to server:
  ```
  DS_PING
  ####
  ```

- **Response** from server:
  ```
  PDQ on trntwn001 (127.0.0.1:14015) alive at Sat 2 14:02:02 EST 2002
  ####
  ```

- **Close** connection request to server:
  ```
  DS_CLOSE
  ####
  ```

- **Response** from server:
  ```
  None.
  ```

As seen below, there are three steps to running a Daytona query: DS_COMPILE, DS_EXECUTE, and DS_CAT.

- **Open** connection request to server:
  ```
  DS_OPEN jdbc jtest
  anotyad
  ####
  ```

- **Response** from server:
  ```
  SUCCESS
  ####
  ```

- **Request** to server:
  ```
  DS_COMPILE R_Test1 hold
  // read 2 arguments from command line: `Area_Code' 'City'
  set [ .cli_arg1, .cli_arg2 ] = read( from _cmd_line_ );
  do Write_Line( "%msg00)title)Suppliers in '.cli_arg1' and Area Code '.cli_arg2' "ISTR );
  $[
  SELECT Supplier_Name AS Supplier, Supplier.Telephone AS Phone
  FROM Supplier
  WHERE Supplier.City Matches .cli_arg1
  AND Supplier.Telephone Matches .cli_arg2
  ORDER BY Supplier.Name
  ```
Response from server:
SUCCESS
###

Request to server:
DS_EXECUTE R_Test1 'St. Paul' '612'
###

Response from server:
SUCCESS
###

Request to server:
DS_CAT R_Test1
###

Response from server:

Response from server:

Response from server:

Request to server:

Response from server:

DS_QQ (for Quick Query) combines the three steps DS_COMPILE, DS_EXECUTE, and DS_CAT into one command.

Open connection request to server:
DS_OPEN cym ctest
anotyad
###

Response from server:
SUCCESS
###

Request to server:
DS_QQ fmt=t -L
do Write_Line( "St. Paul Suppliers in Area Code 612" );
$

Close connection request to server:
DS_CLOSE
###

Response from server:
None.
SELECT SUPPLIER.Name AS Supplier, SUPPLIER.Telephone AS Phone
FROM SUPPLIER
WHERE SUPPLIER.City Matches "St. Paul"
    AND SUPPLIER.Telephone Matches "^612"
ORDER BY SUPPLIER.Name

Response from server:
St. Paul Suppliers in Area Code 612

Query File: /tmp/DS.Q.12345

--------------------------------
Supplier            Phone
--------------------------------
Acme Shipping  612-149-5678
Bouzouki Receiving 612-943-7416
Julius Receiving  612-309-3492
Sunshine Warehouse 612-303-7074
--------------------------------

Close connection request to server:
DS_CLOSE

Response from server:
None.

Query executables can be made to persist from one session to the next (by using the hold keyword). For example, an example above chose to hold the R_Test1 query for user jtest. Running a DS_EXECUTE and DS_SHOW on that previously defined query produces the following:

Open connection request to server:
DS_OPEN jdbc jtest

Response from server:
SUCCESS

Request to server:
DS_EXECUTE R_Test1 'St. Paul' '^612'

Response from server:
SUCCESS

Request to server:
DS_SHOW R_Test1 fmt=t -L
Response from server:

title) Suppliers in 'St. Paul' and Area Code ^612

Query File: R_Test1.Q

--------------------------------
Supplier Phone
--------------------------------
Acme Shipping  612-149-5678
Bouzouki Receiving  612-943-7416
Julius Receiving  612-309-3492
Sunshine Warehouse  612-303-7074
--------------------------------

#####

Close connection request to server:
DS_CLOSE

#####

Response from server:
None.

Query executables can also be shared across users, assuming the appropriate granting of privileges. For example, a query executable for Share_Query can be created by user ctest:

Open connection request to server:
DS_OPEN cym ctest

anotyad

Response from server:
SUCCESS

Request to server:
DS_COMPILE Share_Query hold

select count(*) from SUPPLIER

Response from server:
SUCCESS

Request to server:
DS_ARTHUR_REQUEST

grant actions all on ctest^Share_Query to jtest

#####
Response from server:
OK
#####

Close connection request to server:
DS_CLOSE
#####

Response from server:
None.

The user jtest can then use that executable by referring to ctest~Share_Query:

Open connection request to server:
DS_OPEN jdbc jtest
anotyad
#####

Response from server:
SUCCESS
#####

Request to server:
DS_EXECUTE ctest~Share_Query
#####

Response from server:
SUCCESS
#####

Request to server:
DS_CAT ctest~Share_Query
#####

Response from server:
%msg01)delim) |
%msg02)
%msg03)Query File: Share_Query.Q
%msg04)
%msg05)
%msg06)recls)SHARE_QUERY
%msg07)flds)Col_1
%msg08)types)INT(_long_)
%msg09)
103
#####

Close connection request to server:
DS_CLOSE
#####

Response from server:
None.

**ADMINISTRATIVE COMMANDS**

pdq also provides a command-line interface to facilitate the administration of a pdq instance and the Arthur database instance it will be using (e.g., users, passwords, access constraints, roles, and stored procedures). The command scripts are found in $DS_SERVERDIR/bin. Since these commands work directly with the Arthur command-line interface as needed (see Arthur(8)) while adding any additional
semantics specific to the pdq instance (such as the location and naming of stored procedures), there should be no need to invoke the Arthur administrative commands directly in order to completely configure a pdq instance.

pdq_check_env
- Validate the environment for a pdq instance.
- This check is done in all of the pdq command scripts.

pdq_init
- Initialize a pdq instance and the Arthur instance it will be using.
- This command should only be called once per pdq instance.

pdq_add_user <user> <apis> [<hostnames> [<networks> [<password>]]]
pdq_drop_user <user>
pdq_show_user <user>
- Add/Drop/Show a given user, e.g.:
  $ pdq_add_user juser1 jdbc att.com,sbc.com 10.2.0.0/16 passwd1
- When adding a user, a comma-separated list may be given for <apis>, (using any of 'jdbc', 'pdbi', and 'pydbapi'). Comma-separated lists may also be given for <hostnames>, and <networks>. The defaults for the optional <hostnames> and <networks> are '∗' and '0', respectively, which indicate that all values are accepted. Note that when a constraint list is given for both <hostnames> and <networks>, both constraints must be met. Also, a user password must always be provided. If it is not given on the command line, then the script will prompt for it interactively.
- When showing a user, a '-' argument will show results for all users.

pdq_update_user_password <user> [<password>]
- Update the password for a given pdq user, e.g.:
  $ pdq_update_user_password juser1 newpasswd
- A user password must always be provided. If it is not given on the command line, then the script will prompt for it interactively.

pdq_add_user_api <user> <apis>
pdq_drop_user_api <user> <apis>
pdq_update_user_api <user> <apis>
- Add/Drop/Update the apis for a given pdq user, e.g.:
  $ pdq_add_user_api juser1 pdbi
- A comma-separated list may be given for <apis> (using any of 'jdbc', 'pdbi', and 'pydbapi').

pdq_add_user_host <user> <hostnames>
pdq_drop_user_host <user> <hostnames>
pdq_update_user_host <user> <hostnames>
- Add/Drop/Update the hostnames for a given pdq user, e.g.:
  $ pdq_add_user_host juser1 research.att.com
- A comma-separated list may be given for <hostnames>.

pdq_add_user_network <user> <networks>
pdq_drop_user_network  <user>  <networks>
pdq_update_user_network  <user>  <networks>

- Add/Drop/Update the networks for a given pdq user, e.g.:
  $ pdq_add_user_network  juser1  192.168.1.0/8
- A comma-separated list may be given for  <networks>.

pdq_add_user_stored_proc  <user>  <stored_proc>  [ <qworkdir> ]
pdq_drop_user_stored_proc  <user>  <stored_proc>  [ <qworkdir> ]
pdq_update_user_stored_proc  <user>  <stored_proc>  [ <qworkdir> ]
pdq_show_user_stored_proc  <user>

- Add/Drop/Update/Show stored procedures (or executables) for a given pdq user, e.g.:
  $ pdq_add_user_stored_proc  juser1  $HOME/appbin/Query1
- When adding/updating a user’s stored procedures, the  <stored_proc>  given must be a
  query executable that the user wishes to access via pdq (an absolute or relative path to the
  executable may be included). The  <qworkdir>  (where the user’s pdq directory can be
  found and where the stored procedure will be installed/updated) may also be given, otherwise
  a default of $DS_SERVERDIR/pdq/net/qworkdir will be used. Note that the
  executable will be copied to the user’s pdq directory.
- When dropping a user’s stored procedures, the  <stored_proc>  is just the query execut-
  able name (no path to the executable should be provided). The  <qworkdir>  (where the
  user’s pdq directory can be found and where the stored procedure will be removed) may also be given, otherwise a default of $DS_SERVERDIR/pdq/net/qworkdir will be used.
- When showing a user’s stored procedures, a ‘-’ for the user will show results for all users.

pdq_grant_user_stored_proc_to_user  <owner>  <stored_proc>  <user>
pdq_revoke_user_stored_proc_from_user  <owner>  <stored_proc>  <user>
pdq_show_user_stored_proc_grants  <user>  [ <stored_proc> ]

- Grant/Revoke stored procedure access to/from a given pdq user, or
  Show stored procedure access granted to a given pdq user, e.g.:
  $ pdq_grant_user_stored_proc_to_user  juser1  Query1  juser2
- When showing user grants, a ‘-’ for the user will show results for all users. A specific
  <stored_proc> may be given to constrain the results.

pdq_grant_user_role_to_user  <owner>  <role>  <user>
pdq_revoke_user_role_from_user  <owner>  <role>  <user>
pdq_show_user_role_grants  <user>  [ <role> ]

- Grant/Revoke role access to/from a given pdq user, or
  Show role access granted to a given pdq user, e.g.:
  $ pdq_grant_user_role_to_user  juser1  qrole  juser2
- When showing user grants, a ‘-’ for the user will show results for all users. A specific
  <role> may be given to constrain the results.

pdq_add_user_stored_proc_to_role  <user>  <stored_proc>  <role>
pdq_drop_user_stored_proc_from_role  <user>  <stored_proc>  <role>

- Add/Drop stored procedures for a given pdq role, e.g.:
  $ pdq_add_user_stored_proc_to_role  juser1  Query1  qrole

pdq_drop_user_role  <user>  <role>
pdq_show_role <role>
    - Drop/Show a given pdq role, e.g.:
      $ pdq_drop_user_role juser1 qrole

pdq_show_all_stored_procs_for_user <user>
pdq_show_all_roles_for_user <user>
    - Show all stored procedures/roles for a given pdq user, e.g.:
      $ pdq_show_all_stored_procs_for_user juser1
      - The format of the stored procedure result is owner"stored_proc.

pdq_show_all_stored_procs_for_role <role>
    - Show all stored procedures for a given pdq role, e.g.:
      $ pdq_show_all_stored_procs_for_role qrole
      - The format of the result is owner"stored_proc.

ADMINISTRATIVE EXAMPLES

# Go to my Daytona app
$ cd $MY_APP_DIR

# Set my environment; calls DS_Set, sets DS_SERVERDIR
$ . setup_my_app.sh

# Create a new user
$ pdq_add_user juser1 jdbc mach1.research.att.com 0 passwd1
 pdq_add_user completed: \ 
    user juser1 added

# Oops, thought of another host
$ pdq_add_user_host juser1 mach2.research.att.com
 pdq_add_user_host completed: \ 
    host mach2.research.att.com added for user juser1

# Compile a query to use as a stored procedure
$ DS Compile Q/Query1.cy bin/Query1
 Tracy finished for Q/Query1.cy
 R moved to bin/Query1

# Make the query accessible as a stored procedure
$ pdq_add_user_stored_proc juser1 bin/Query1
 pdq_add_user_stored_proc completed: \ 
    stored procedure Query1 added for user juser1

# Give another user access (juser2 already exists)
$ pdq_grant_user_stored_proc_to_user juser1 Query1 juser2
 pdq_grant_user_stored_proc_to_user completed: \ 
    stored procedure [juser1~]Query1 access granted to user juser2

# Verify that 'juser2' now has access
$ pdq_show_all_stored_procs_for_user juser2
[... select header removed ...]
    juser1~Query1
ENVIRONMENT

**DS_DIR**
Directory containing the Daytona executables.

**DS_PATH**
Colon-separated list of directories to search for project and application archives as well as other global user-defined Daytona environment files.

**DS_PROJ**
Name of the Daytona project; optional.

**DS_APPS**
Colon-separated list of applications corresponding to the application archives.

**DS_SERVERDIR**
Directory (usually under **DS_DIR**) containing a directory for **pdq** with the following subdirectories:

- **bin:** contains the executable **pdq**, its associated scripts, and the executable tester Poly Daytona Client **pdc**.
- **local:** a place to execute **pdq** listening on localhost,
  *e.g.*, localhost or 127.0.0.1, port 14015.
- **net:** a place to execute **pdq** listening on a network address,
  *e.g.*, trntwn.acme.org or 63.251.83.40, port 14016.
- **test:** contains test scripts for **pdc** to read to exercise **pdq**. (**pdq** and **pdc** are written in **Cymbal**.)

ACKNOWLEDGEMENTS
Based on the Daytona **Cymbal** Example/Test server script:

```
$DS_DIR/EXAMPLES/usr/orders/Q/itsvr.qry.1.IQ
```
by R. L. Greer, AT&T Labs - Research, the father of **Daytona**. Many thanks!!

AUTHOR
J. J. Snyder, Consultant at AT&T Labs - Research

COPYRIGHT
Copyright © 2000, © 2001, © 2002, and © 2003 by:
AT&T Labs - Research, U.S.A.

SEE ALSO
Arthur**(8)**, **DS**(1), **DC-prn**(1), **DC-pkt**(1), **DC**(5), **DS Env**(1), and **egrep**(1).

All About Daytona

backtalk
A Text Generator, Tree Manipulator And Macro Processor,

JDBC API Tutorial and Reference, Second Edition

Microsoft ODBC 3.0
Software Development Kit and Programmer’s Reference,

Programming the Perl DBI

**SQL in a Nutshell**
A Desktop Quick Reference,
NAME
pdc – A Poly Daytona Client for querying a pdq server

SYNOPSIS
pdc [ −host host ] [ −port port_nbr ]
[ −secure ] [ −ver{version} ] [ +? ] file

DESCRIPTION
pdc is a Daytona client used to communicate over a TCP network with a pdq server via an IP address and port number.

The file provided as an argument to pdc contains the list of requests that will be passed to the pdq server. If a ‘-’ is given for file (instead of a file name), then the list of requests will be read from stdin.

Each request must be a message terminated by a line consisting of the 5-character string #####. The remaining protocol for these requests is defined elsewhere (see pdq (8)).

The executable for pdc is found in the directory $DS_SERVERDIR/pdq/bin.

OPTIONS
−host host
   Specifies the internet DNS hostname or IPv4 address (in octet form) at which pdq is listening for client requests.

−port port_nbr
   Specifies the port on which pdq is listening for client requests.

−secure
   Specifies that pdc must operate in secure communication mode, where interactions with the server will be encrypted. A connection will only be accepted by a server that is also running in secure communication mode.

−ver{version}
   Output the version information for pdc and exit.

+? Output usage (help) information for pdc and exit.

EXAMPLES
Assume that a pdq server is running on machine trntwn001 and listening on port 14015, and assume that the file req_list contains the following:

   DS_OPEN cym ctest
   anotyad
   #####
   DS_PING
   #####
   DS_CLOSE
   #####

A good (successful) interaction with the pdq server would look something like:

   $ pdc -host trntwn001 -port 14015 req_list
   SUCCESS
   #####
   PDQ on trntwn001 (127.0.0.1:14015) alive at Sat 2 14:02:02 EST 2002
   #####

Note that DS_CLOSE is the appropriate way to terminate the connection with the pdq server (although the request does not produce a server response).

AUTHOR
AT&T Labs - Research
COPYRIGHT
Copyright © 2010 by:
AT&T Labs - Research, U.S.A.

SEE ALSO
pdq(8).

All About Daytona
Arthur

Arthur is an authorization/security executable for Daytona typically used in conjunction with pdq to provide role-based security.

Theory of Operation
====================

Arthur is typically called as a co-process by pdq and Tracy, though it will also function as a stand-alone program.

Arthur manages user and DBA interaction with a set of Daytona record classes: AGENT, TARGET, TARGET_KIND, NETWORK_GROUP, DOMAIN_GROUP, INTERVAL_GROUP, ROLE_SPEC, ROLE_DENIAL, AUTHORIZATION, and LOG_ENTRY.

Arthur accepts commands to modify the contents of the tables (ADD_PROJECT_TARGETS, INSERT, DELETE, UPDATE, AUGMENT, DECREMENT, REQUEST), check the tables for specific security information (AUTHENTICATED?, AUTHORIZED?), display table contents and parameters (SHOW), and set parameters (SET).

The AUTHORIZATION, ROLE_SPEC, and ROLE_DENIAL tables may be modified only through Arthur SQL DDL-level queries for managing authorizations (grant, revoke, define role, etc.). At the query level, the kinds and set of targets and their associated actions are regarded as fixed. Queries are invoked with the REQUEST command. ADD_PROJECT_TARGETS may be used to add the projects, record classes, and columns of a Daytona database to TARGET. Also regarded as fixed at the query level are the contents of the AGENT, TARGET_KIND, NETWORK_GROUP, DOMAIN_GROUP, and INTERVAL_GROUP.

Database administrators (DBA) will make changes to these tables, at a minimum to add agents, more typically to define named networks, domains, and intervals. A DBA may also wish to use Arthur to manage a set of targets that changes over time, such as a set of executables.

In some cases the DBA who is also the UNIX owner of Arthur, known as _arthur_, may wish to enable pdq to accept commands from others to modify Arthur’s record classes. Arthur can be used to manage this process: initially TARGET contains some of Arthur’s record classes: AGENT, TARGET, TARGET_KIND, NETWORK_GROUP, DOMAIN_GROUP, and INTERVAL_GROUP. The kind of these targets is ‘arthur_recls’, the allowed action is ‘arthur_modify’, and the owner is _arthur_. Thus, to record that agent ‘dba0’ may modify TARGET, the UNIX owner DBA _arthur_ should use REQUEST to grant the action ‘arthur_modify’ on TARGET to dba0. See $DS_ARTHURDIR/Docs/AuthorizingExecutables.txt for a complete description and example.

At the query level, a basic management concept is a privilege. A privilege is the ability to do an action on a target from an IP network/host and from an internet domain during a time interval. An AUTHORIZATION or grant of a privilege is given from a grantor agent to a grantees agent with, optionally, an option to grant the privilege to other agents. Privileges may also be included within ROLE_SPECS; a role being merely a named collection of privileges (a role may also contain another role). A role may also contain denials (ROLE_DENIALS),
privileges which are explicitly disallowed. This makes it easier to specify the inclusion of all but a small number of actions, networks, domains, or intervals (there is an example in the Queries section below).

Commands and queries may be passed in from a remote user via the REQUEST command. In all cases Arthur checks that the user is authorized for the command or query.

Fields
=======

Service
-------

Services are capabilities that pdq or other trusted users of Arthur offer for agents for their use. Examples of services are the pdbi, jdbc, odbc, and cym database interfaces that pdq supports.

Target
------

Targets, contained in TARGET, are hierarchical, in that one target may be the parent of another, recursively. Granting a privilege or role on a target also effectively grants the privilege or role on all the target’s descendants. In a role, denying a privilege on a target denies that privilege on the target’s descendants.

There are two ways to name targets. The first way is to use the ‘.’ syntax to specify all the ancestors of a target in its name. Thus, if Name is a target and its ancestors are SUPPLIER and orders, then orders.SUPPLIER.Name should be used for the name of the target. In this naming scheme every target has exactly one parent, but possibly many children. This naming scheme prevents confusion when there may two targets with the same name, but different parents, for example, orders.SUPPLIER.Name and orders.PERSON.Name, which are similarly named fields in different tables.

The second way to name targets omits the ‘.’ syntax and should only be used when there is no danger of conflicting names. An example would be executables: if formatter, antivirus, spreadsheet, word_processor, utility, and application are targets, then utility could be the parent of formatter and antivirus, and application could be the parent of spreadsheet and word_processor. But in this naming scheme a target can have more than one parent, so application could also be the parent of antivirus. This makes it possible to assemble a set of targets, which may already have logical parent, under a new parent and refer to them all by the name of the new parent. For example, my_apps could be added as the parent of formatter and spreadsheet.

The two naming schemes can be mixed in TARGET, there is no need to stick to one or the other.

Action
Actions are processes or procedures that are applied to targets. For example, select, delete, and update are actions that may be applied to record targets. The TARGET_KIND table specifies which actions are applicable to which targets.

Network

A network is a valid IP address and an optional mask described by an integer from 0 to 32 for IPv4 (and 0 to 128 for IPv6) indicating how many left-justified digits describe the network.

\[
\text{network} := \langle \text{ip} \rangle / \langle \text{mask} \rangle
\]

If the ‘/’ and <mask> are omitted, a network consisting of a single IP address is indicated.

Examples:

<table>
<thead>
<tr>
<th>IPv4</th>
<th>IPv6</th>
</tr>
</thead>
<tbody>
<tr>
<td>132.45.66.0/24</td>
<td>2001:1890:1110:13::/64</td>
</tr>
<tr>
<td>132.45.66.11</td>
<td>2001:1890:1110:13::2000:50</td>
</tr>
<tr>
<td>0.0.0.0</td>
<td>:</td>
</tr>
</tbody>
</table>

The last example in each column describes the network that contains any address and is converted to the wildcard 0. Note that valid network specifications are such that the mask applied to the IP address must yield the address itself.

Domain

Internet domains must conform to RFC 952/921 which are complicated but pretty much distill to strings consisting of characters drawn from the alphabet (A-Za-z), digits (0-9), minus sign (-), and period (.). Here is the regular expression that Daytona uses to validate domain strings:

\[
^\langle a-zA-Z-a-zA-Z0-9.\rangle+\langle a-zA-Z-0-9.\rangle$\]

Interval

Time intervals are represented as follows:

\[
\text{hours} := \langle \text{days} \rangle @ (\langle \text{start\_time} \rangle \rightarrow \langle \text{end\_time} \rangle)
\]

\[
\text{days} := \langle \text{day1} \rangle, \langle \text{day2} \rangle, \ldots\text{ or } \langle \text{day1} \rangle \rightarrow \langle \text{day2} \rangle \text{ or both}
\]

\[
\langle \text{dayN} \rangle := \text{‘Sun’, ‘Mon’, ‘Tue’, ‘Wed’, ‘Thu’, ‘Fri’, ‘Sat’}
\]

where start\_time and end\_time are given in standard 00:00:00 (or AM/PM) format. Duplication is accepted. Examples:
Either hours or days may be empty, in which cases all possibilities are implied. 'Mon->Fri@' means Monday through Friday at any time of the day. '@06:00->20:00' means 6 am to 8 pm any day of the week. '@' is the wildcard and matches any interval.

Option
------

One of: (empty), grant, grantability, supergrantability. Used in queries to control the degree to which an agent may grant to another agent a privilege that they have been granted.

Wildcards
---------

The wildcards for networks, domains, and intervals are '0', '*', and '@', respectively. '.*' is also a wild card that is equivalent to '*', except that '-' or the absence of an input will not yield a match in AUTHENTICATED? and AUTHORIZED?.

Groups
=====

It is possible to refer to actions, networks, domains, and intervals by the names of groups that contain collections of one type of these items. Group definitions are found in the ACTION_GROUP, NETWORK_GROUP, DOMAIN_GROUP, and INTERVAL_GROUP tables. Group definitions are not recursive, so no group may contain the name of another group.

Here are some examples:

ACTION_GROUP: travel = [drive|fly]
    arthur_modify = [INSERT|DELETE|UPDATE|AUGMENT|DECREMENT]
NETWORK_GROUP: local_lan = [137.5.123.0/24|137.17.123.0/24]
    DOMAINS_GROUP: local_domain = [here.com|there.com]
    INTERVAL_GROUP: workday = [Mon->Fri@08:00->20:00|Sat->Sun@09:00->17:00]

Executing `. $DS_ARTHURDIR/bin/setup.arthur` will set the shell environment
for the Arthur user, who is usually the DBA (and only the DBA).
To find out what DS_ARTHURDIR is and be able to set it in the true shell environment, use `getdsenv DS_ARTHURDIR`.
The default for DS_ARTHURDIR can be overridden at Daytona installation by executing the likes of:

   . $DS_DIR/DS_Set 'DS_ARTHURDIR=<myartdir>'

Indeed, the DBA will probably _not_ want to work with the ARTHUR under $DS_DIR since it will come and go with the Daytona release. To that end, the DBA needs to make a copy of $DS_DIR/ARTHUR by running the shell command script install.arthur. This will create the DBA’s own ARTHUR directory with the executables being symbolic links back to the ones under $DS_DIR. That way, when Daytona is upgraded, so also will the user’s ARTHUR executables automatically; otherwise, if that is not desired, the DBA should just copy $DS_DIR/ARTHUR to their own directory, executables and all. The main message is that whatever is done, no-one should set up their application’s use of Arthur in $DS_DIR/ARTHUR itself.

Command line arguments
======================

Arthur runs as a coprocess or as a standalone program, in which case it is invoked as follows:

   Arthur <file1> <file2> ...

where each argument is the name of a script file containing commands that Arthur understands or the single character ‘-’, which directs Arthur to read input from stdin.

Message structure
=================

From each serial source (file or stdin) Arthur looks for a sequence of messages. Each message begins with a word identifying the message class (such as ‘AUTHORIZED?’ or ‘INSERT’), followed by further parameters on the same or several lines, and ending with ‘&&&&&’, which must be on a line by itself and terminated by a newline. Arthur executes the command contained in each message before looking for the next message.

Some parameters may be set-valued. Use ‘|’ to separate values. For example:

   adam|burt|chad

Any blank lines or lines starting with ‘#’ that appear before or after a message will be ignored.

When receiving input from stdin, Arthur can be made to quit with either ^D or the QUIT or EXIT message.
Arthur will respond with one or more lines starting with the response class (such as 'OK' or 'AUTHORIZED') on a line by itself, followed in some cases by one or more lines of information, followed by '&&&&&' on a line by itself.

Placeholders
-------------

Use the hyphen ‘-’ as a placeholder when no input is given for a field. Trailing hyphens may be omitted. For example, these two commands are equivalent:

```
UPDATE AGENT  adam  -  jdbc  -  
&&&&&
```

```
UPDATE AGENT  adam  -  jdbc 
&&&&&
```

Command types and syntax
========================

(The many script.* files in $DS_ARTHURDIR/Tests offer many examples of using these commands; the corresponding result.* show the results of executing these scripts against the test configuration.)

```
AUTHENTICATED?  <name>  <password>  <service>  <client_ip>  <dns_entry>
```

Confirms that AGENT <name> is authenticated for using the indicated service from the indicated IP address and DNS entry at the current date_clock (or at the time in DAY_CLOCK, if it is set). <service> must match one of the Services in the AGENT record uniquely identified by the Name <name>, <client_ip> must match one of the Networks, <dns_entry> must match one of the Domains and the current time must match one of the Interval entries, all in this same AGENT record.

Arthur will produce one of the following responses:

AUTHENTIC or INAUTHENTIC

Examples:

```
AUTHENTICATED?  adam  pass  jdbc  192.168.1.1  foo.com 
&&&&&
```

```
AUTHENTICATED?  adam  pass  jdbc 
&&&&&
```

The second example will confirm that for the AGENT record for ‘adam’, the Password is ‘pass’, ‘jdbc’ is among the Services, that the current time (or the value in DAY_CLOCK) matches one of the Intervals entries, and
that the fields Networks and Domains contain '0' and '*', respectively.

**Wild cards**

----------

In each AGENT record are fields Networks, Domains, and Intervals. When AUTHENTICATED? is called, its arguments <client_ip> and <dns_entry> must each be passed by one of the members of Networks and Domains, respectively. The tables below indicate this behavior. '*' as a member of Domains means that there must be a <dns_entry> supplied with AUTHENTICATED?; hence, '-' is not passed. Note also that if Domains is empty, no <dns_entry> will pass, similarly, if Networks is empty, no <client_ip> will pass, and if Intervals is empty, the current time will not pass. In all these cases, AUTHENTICATED? will return 'INAUTHENTIC'.

### AGENT:Network field contents

<table>
<thead>
<tr>
<th></th>
<th>'1.2.3.4'</th>
<th>0</th>
<th>&lt;empty&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>'1.2.3.4'</td>
<td>pass</td>
<td>pass</td>
<td>deny</td>
</tr>
<tr>
<td>client_ip '5.6.7.8'</td>
<td>deny</td>
<td>pass</td>
<td>deny</td>
</tr>
<tr>
<td>'-'</td>
<td>deny</td>
<td>pass</td>
<td>deny</td>
</tr>
</tbody>
</table>

### AGENT:Domain field contents

<table>
<thead>
<tr>
<th></th>
<th>'att.com'</th>
<th>*</th>
<th>.*</th>
<th>&lt;empty&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>'att.com'</td>
<td>pass</td>
<td>pass</td>
<td>pass</td>
<td>deny</td>
</tr>
<tr>
<td>dns_entry 'gnu.com'</td>
<td>deny</td>
<td>pass</td>
<td>pass</td>
<td>deny</td>
</tr>
<tr>
<td>'-'</td>
<td>deny</td>
<td>pass</td>
<td>deny</td>
<td>deny</td>
</tr>
</tbody>
</table>

### AGENT:Intervals field contents

<table>
<thead>
<tr>
<th></th>
<th>'Mon@11-&gt;12'</th>
<th>@</th>
<th>&lt;empty&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>current date_clock 'Mon@11'</td>
<td>pass</td>
<td>pass</td>
<td>deny</td>
</tr>
<tr>
<td>'Tue@11'</td>
<td>deny</td>
<td>pass</td>
<td>deny</td>
</tr>
</tbody>
</table>
AUTHORIZED? <name> <action> <target> <client_ip> <dns_entry>

Confirms that AGENT <name> is authorized to take the indicated action on the indicated target from the indicated IP address and DNS entry at the current time (or at the time in DAY_CLOCK, if it is set). Example:

AUTHORIZED? adam select orders.SUPPLIER 66.77.88.99 research.att.com

Arthur’s response will either be

AUTHORIZED or UNAUTHORIZED

Wild cards

Grants of privileges and roles can be made explicitly using wildcards, as in

grant actions all on orders to kirk from network 0 from domain * during @

but if any of network, domain, or interval is not specified in a grant query, the corresponding wildcard is inserted. (Thus, unlike the AGENT table, there are never any empty Network, Domain, or Interval fields in the AUTHORIZATION or ROLE_SPEC tables.) The tables below indicate how the various cases are handled. Note that * trumps .*, whereas in AGENT they are not allowed to both appear in the same record.

AUTHORIZATION/ROLE_SPEC:Network fields contents

<table>
<thead>
<tr>
<th>&quot;1.2.3.4&quot;</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;1.2.3.4&quot;</td>
<td>pass</td>
</tr>
<tr>
<td>client_ip</td>
<td>&quot;5.6.7.8&quot;</td>
</tr>
<tr>
<td>&quot;-&quot;</td>
<td>deny</td>
</tr>
</tbody>
</table>

AUTHORIZATION/ROLE_SPEC:Domain fields contents

<table>
<thead>
<tr>
<th>&quot;att.com&quot;</th>
<th>*</th>
<th>.*</th>
<th>* and .*</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;att.com&quot;</td>
<td>pass</td>
<td>pass</td>
<td>pass</td>
</tr>
<tr>
<td>dns_entry</td>
<td>&quot;gnu.com&quot;</td>
<td>deny</td>
<td>pass</td>
</tr>
<tr>
<td>&quot;-&quot;</td>
<td>deny</td>
<td>pass</td>
<td>deny</td>
</tr>
<tr>
<td>AUTHORIZATION/ROLE_SPEC: Interval fields contents</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;Mon@11-&gt;12&quot;</td>
<td>@</td>
<td></td>
<td></td>
</tr>
<tr>
<td>current</td>
<td>&quot;Mon@11&quot;</td>
<td>pass</td>
<td>pass</td>
</tr>
<tr>
<td>date_clock</td>
<td>&quot;Tue@11&quot;</td>
<td>deny</td>
<td>pass</td>
</tr>
</tbody>
</table>

---------------------

REQUEST <agent> <command or query>

Submits a command or a query from AGENT <name>. Returns a message suitable for passing back to the agent.

Commands INSERT, DELETE, UPDATE, AUGMENT, and DECREMENT require that the agent be authorized to take action ‘arthur_modify’ on the table given in the command. All other commands require that the agent be authorized to take the action ‘arthur_execute’ on the target COMMAND. For information on authorizations needed for queries, see the section on Queries below.

Returns OK or ERROR, followed by the command or query response.

Examples:

REQUEST adam
INSERT AGENT burt pass jdbc 0 * Mon->Fri@06:00->20:00

Arthur checks first that adam is authorized to take action arthur_modify on AGENT before carrying out the command.

REQUEST adam
SHOW AGENT

Arthur checks first that adam is authorized to take action arthur_execute on COMMAND before carrying out the command.

REQUEST adam
define role clerk to include actions select on orders.SUPPLIER
grant role clerk to burt with grant option

Arthur checks that adam is authorized to request each of these queries.
INSERT <table> <field values sequence>

Inserts a new record in the indicated table with the given field values. Set-valued fields may be given a set of values by using ‘|’ as a separator. Example: ‘tom|joe’. Returns OK or ERROR.

Currently operative for tables AGENT, TARGET, TARGET_KIND, NETWORK_GROUP, DOMAIN_GROUP, and INTERVAL_GROUP (use a grant query to insert records into AUTHORIZATION, ROLE_SPEC, and ROLE_DENIAL):

INSERT AGENT <name> <password> <services> <networks> <domains> <intervals>

INSERT TARGET_KIND <name> <actions>

INSERT TARGET <name> <owner> <kind> <children>

INSERT ACTION_GROUP <name> <actions>

INSERT NETWORK_GROUP <name> <networks>

INSERT DOMAIN_GROUP <name> <domains>

INSERT INTERVAL_GROUP <name> <intervals>

Examples:

INSERT AGENT tom pass jdbc|odbc 132.45.66.0/24 att.com|gnu.com

INSERT TARGET_KIND pinball reset|play

INSERT TARGET PowerPlay tom pinball -

INSERT ACTION_GROUP shop walk|look|try|buy

INSERT NETWORK_GROUP lan 192.168.0.0/24|90.0.0.0/8

INSERT DOMAIN_GROUP lan inside.com|outside.com

INSERT INTERVAL_GROUP playday Mon->Fri@18:00->22:00|Sat,Sun@10am->5pm

DELETE <table> <field value sequence>

Deletes every record that contains one of the indicated values in each field. Note that set-valued fields accept multiple values. Returns OK or ERROR.

Currently operative for tables AGENT, TARGET, TARGET_KIND, NETWORK_GROUP, DOMAIN_GROUP, and INTERVAL_GROUP:
DELETE AGENT <name> <password> <services> <networks> <domains> <intervals>

DELETE TARGET_KIND <name> <actions>

DELETE TARGET <name> <owner> <kind> <children>

DELETE ACTION_GROUP <name> <actions>

DELETE NETWORK_GROUP <name> <networks>

DELETE DOMAIN_GROUP <name> <domains>

DELETE INTERVAL_GROUP <name> <intervals>

Examples:

DELETE AGENT mallory

DELETE AGENT jdbc|odbc att.com

DELETE TARGET_KIND reset|play

DELETE TARGET pinball

DELETE ACTION_GROUP shop

DELETE NETWORK_GROUP 90.0.0.0/8

DELETE DOMAIN_GROUP lan

DELETE INTERVAL_GROUP playday Sat,Sun@10am->5pm

The first example deletes the AGENT record with Name 'mallory'.
The second example deletes all AGENT records with either 'jdbc' or 'odbc' as
one of the members of the Services field and 'att.com' as one of the members
of the Domains field.
The last example deletes all INTERVAL_GROUP records with Name playday
and 'Sat,Sun@10am->5pm' as one of the members of the Intervals field.

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

UPDATE <table> <key field sequence> <field values sequence>

In the record uniquely identified by the key field sequence, replaces the
indicated field values. Use '-' as a placeholder for unspecified fields.
Returns OK or ERROR.

Currently operative for tables AGENT, TARGET, TARGET_KIND, NETWORK_GROUP,
DOMAIN_GROUP, and INTERVAL_GROUP. The first field is the key field for each
of the following:

UPDATE AGENT <name> <password> <services> <networks> <domains> <intervals>

UPDATE TARGET_KIND <name> <actions>

UPDATE TARGET <name> <owner> <kind> <children>

UPDATE ACTION_GROUP <name> <actions>

UPDATE NETWORK_GROUP <name> <networks>

UPDATE DOMAIN_GROUP <name> <domains>

UPDATE INTERVAL_GROUP <name> <intervals>

Examples:

UPDATE AGENT tom - xml 22.44.66.00/24

UPDATE TARGET_KIND pinball reset|play|poweroff

UPDATE TARGET PowerPlay - - flippers|balls

UPDATE ACTION_GROUP shop drive|buy|pay

UPDATE NETWORK_GROUP lan 90.1.0.0/16

UPDATE DOMAIN_GROUP lan up.com|down.com

UPDATE INTERVAL_GROUP playday Mon,Wed,Fri@2PM->5PM

AUGMENT <table> <key field sequence> <field values sequence>

DECREMENT <table> <key field sequence> <field values sequence>

These two commands add to or remove values from set-valued fields in records indicated by the key. The first field is the key field. Returns OK or ERROR.

Currently operative for tables AGENT, TARGET, TARGET_KIND, NETWORK_GROUP, DOMAIN_GROUP, and INTERVAL_GROUP. The key field for all tables is <name>:

AUGMENT AGENT <name> <services> <networks> <domains> <intervals>

DECREMENT AGENT <name> <services> <networks> <domains> <intervals>

AUGMENT TARGET_KIND <name> <actions>

DECREMENT TARGET_KIND <name> <actions>

AUGMENT TARGET <name> <children>
DECREMENT TARGET <name> <children>

AUGMENT ACTION_GROUP <name> <actions>
DECREMENT ACTION_GROUP <name> <actions>

AUGMENT NETWORK_GROUP <name> <networks>
DECREMENT NETWORK_GROUP <name> <networks>

AUGMENT DOMAIN_GROUP <name> <domains>
DECREMENT DOMAIN_GROUP <name> <domains>

AUGMENT INTERVAL_GROUP <name> <intervals>
DECREMENT INTERVAL_GROUP <name> <intervals>

Examples:

AUGMENT AGENT tom odbc

AUGMENT TARGET_KIND pinball poweron

AUGMENT TARGET PowerPlay lights

AUGMENT ACTION_GROUP shop browse

AUGMENT NETWORK_GROUP lan 55.99.0.0/16

AUGMENT DOMAIN_GROUP lan left.com|right.com

AUGMENT INTERVAL_GROUP playday Tue,Thu@2PM->5PM

DECREMENT AGENT tom odbc

---

SHOW <table> <field values sequences>

Displays the records in a table which contain at least one of the indicated field values in each non-hyphen field. Note that every field, including non-set-valued fields, may here be given a set as input. To see the entire tables, just invoke with the table name only. Returns INFO or ERROR.

Currently operative for tables AGENT, TARGET, TARGET_KIND, NETWORK_GROUP, DOMAIN_GROUP, and INTERVAL_GROUP:

SHOW AGENT <names> <passwords> <services> <networks> <domains> <intervals>
SHOW TARGET_KIND <names> <actions>
SHOW TARGET <names> <owners> <kinds> <children>
SHOW ACTION_GROUP <names> <actions>
SHOW NETWORK_GROUP <names> <networks>
SHOW DOMAIN_GROUP <names> <domains>
SHOW INTERVAL_GROUP <names> <intervals>
SHOW ROLE_SPEC <creators> <names> <included_roles> <targets>
SHOW ROLE_DENIAL <creators> <roles> <targets>
SHOW AUTHORIZATION <grantors> <grantees> <roles> <targets>
SHOW LOG_ENTRY <commands> <date_clock_begin> <date_clock_end>

Additional notes on SHOW LOG_ENTRY: accepts multiple commands separated by |. Commands may contain spaces, in which case the entire commands string should be enclosed in single quotes. If date_clock_begin is not given, records up to date_clock_end are shown. If date_clock_end is not given, records after date_clock_begin are shown. If neither is given, all records are shown.

Examples:

SHOW AGENT - - odbc
SHOW TARGET_KIND - play|poweroff
SHOW TARGET - tom
SHOW ACTION_GROUP - select
SHOW NETWORK_GROUP - 55.99.0.0/16
SHOW DOMAIN_GROUP - left.com|right.com
SHOW INTERVAL_GROUP playday
SHOW ROLE_SPEC - valet
SHOW ROLE_DENIAL - valet
SHOW AUTHORIZATION - tom
SHOW LOG_ENTRY - 07/05/04@11:24:05 2004-07-05@13:34
SHOW LOG_ENTRY 'AUGMENT AGENT'
SHOW LOG_ENTRY 'AUGMENT AGENT|UPDATE AGENT'
SHOW LOG_ENTRY AUGMENT (shows all AUGMENTs)
SHOW LOG_ENTRY AUG|UPD (shows all UPDATEs and AUGMENTs)
SHOW LOG_ENTRY ' [A-Z, ]*on' (shows commands ending in 'on', such as 'SET ECHO on' and 'SET VERBOSE on')

The outputs of all these commands, except SHOW LOG_ENTRY, will be arranged in columns with headings. The output of SHOW LOG_ENTRY is organized as follows: (with 2 spaces between entities on the first line and a blank line after the response):
<timestamp>  <command>  <arguments>
<response>
(blank)

-------------------------
SET CLIENT_DS_PATH <path>

Tells Arthur the location of the project or applications for which
ADD_PROJECT_TARGETS will add target information to TARGET. Must be set before
issuing the ADD_PROJECT_TARGETS command. Example:

SET CLIENT_DS_PATH /export/home/admin/d

SET CLIENT_DS_PATH $ORDERS_HOME

Returns OK or ERROR.

-------------------------
SET CLIENT_DS_APPS <app1> [app2] ...
SET CLIENT_DS_APPS <app1>::[app2] ...

Tells Arthur the names of the applications for which ADD_PROJECT_TARGETS will
add target information to TARGET. If CLIENT_DS_PROJ is set, then each
application must be in that project. Example:

SET CLIENT_DS_APPS orders misc

SET CLIENT_DS_APPS orders:misc

Returns OK or ERROR.

-------------------------
SET CLIENT_DS_PROJ <project>

Tells Arthur the name of the project for which ADD_PROJECT_TARGETS will
add target information to TARGET. Also sets CLIENT_DS_APPS to the applications
in this project. Example:

SET CLIENT_DS_PROJ daytona

Returns OK or ERROR.

('daytona' just happens to be the name of the project that Daytona uses
for its test suite. The user’s own value for DS_PROJ would go here, of course.)
ADD_PROJECT_TARGETS

Adds targets from the applications set by SET CLIENT_DS_APPS to Arthur’s
TARGET table, or all applications in a project if SET CLIENT_DS_PROJ has
been executed. In the latter case, the list of applications in Arthur will be
set to the list of all applications in the project. Use SHOW CLIENT_DS_APPS to
see the list. _arthur_ becomes the owner in the TARGET table of all targets.

Returns OK or ERROR.

HELP [pattern1] [pattern2]

Help does pattern matching on the first two words of commands. The hyphen ‘-’
is treated as a wildcard.

HELP INSERT AGENT

Response:

INFO
INSERT AGENT Name Passwd Services Networks Domains Intervals

HELP INSERT

Response:

INFO
INSERT AGENT Name Passwd Services Networks Domains Intervals
INSERT TARGET Name Owner Kind Children
INSERT TARGET_KIND Name Actions
INSERT NETWORK_GROUP Name Networks
INSERT DOMAIN_GROUP Name Domains
INSERT INTERVAL_GROUP Name Intervals

HELP - AGENT

Response:

INFO
INSERT AGENT Name Passwd Services Networks Domains Intervals
DELETE AGENT Name Passwd Services Networks Domains Intervals
UPDATE AGENT Name Passwd Services Networks Domains Intervals
AUGMENT AGENT Name Services Networks Domains Intervals
DECREMENT AGENT Name Services Networks Domains Intervals
SHOW AGENT Names Passwds Services Networks Domains Intervals

~~~~~~~~~~

SET ECHO on
SET ECHO off

When set, Arthur will, except for REQUEST, first echo the command before giving
the response. REQUEST responses already contain the lines of the input query.

The main use for this option is to make the results files in $DS_ARTHURDIR/Tests
easier to understand.

Returns OK or ERROR.

~~~~~~~~~~

SET VERBOSE on
SET VERBOSE off

Is set to ‘off’ when Arthur starts. When ‘off’, returns only ‘OK’ or ‘ERROR’,
omitting the rest of the response. ‘INFO’ responses are unaffected.

Returns OK or ERROR.

~~~~~~~~~~

SET TRACE <trace level>

Trace level 0 is the default. Trace levels 1 and 2 return additional
information for certain operations.

Returns OK or ERROR.

~~~~~~~~~~

SHOW CLIENT_DS_PATH
SHOW CLIENT_DS_APPS
SHOW CLIENT_DS_PROJ
SHOW VERBOISE
SHOW TRACE
SHOW ECHO
SHOW DAY_CLOCK

Returns INFO and the current value of the requested parameter.
-----
QUIT
EXIT

Causes Arthur to terminate.

[The following commands are used in testing Arthur.]

-------------------
CLEAR_TABLE <table>

Deletes all entries from the named table. Valid are: AGENT, TARGET,
AUTHORIZATION, ROLE_SPEC, ROLE_DENIAL, NETWORK_GROUP, DOMAIN_GROUP,
INTERVAL_GROUP, and LOG_ENTRY. Returns OK or ERROR.

*USE WITH CARE!*

-------------------
GRANT_ALL <name> <target>

Grants all privileges on the target to the agent from _arthur_. Used to give
all privileges to an agent as a starting point for testing grant commands and
AUTHORIZED?. Returns OK or ERROR.

-------------------
SET DAY_CLOCK  day@clock

Sets a day of the week and a time of day for AUTHENTICATED? to use instead of
the current calendar date and time of day. Used in testing AUTHENTICATED?.
Returns OK or ERROR.

-------------------
UNSET DAY_CLOCK

Clears this parameter. Arthur returns to using the current date and time of
day in AUTHENTICATED? and AUTHORIZED?. Returns OK.

-------------------
VALIDATE_ALL

Validates the contents of the tables AGENT, TARGET, TARGET_KIND,
NETWORK_GROUP, DOMAIN_GROUP, INTERVAL_GROUP, ROLE_SPEC, ROLE_DENIAL,
AUTHORIZATION. Returns OK or ERROR.
Queries
=======

Queries are used by agents to manage security on targets via the granting of privileges and roles. (The AUTHORIZED? command is used to check for these authorizations.) Queries are passed to Arthur with the REQUEST command. In the $DS_ARTHURDIR/Tests directory, see script.privileges and script.roles for many examples of queries.

Target Kinds and Actions
------------------------

Every target is of a particular kind. All targets of like kind share their allowed actions. The kinds of targets and their associated actions are stored in the TARGET_KIND record class; initially it contains the following:

- arthur_command;[arthur_execute]
- arthur_recls;[arthur_modify|index|drop]
- project;[select|update|insert|delete|create|index]
- application;[select|update|insert|delete|create|index]
- record_class;[select|update|insert|delete|create|index]
- field;[select|update]
- query;[compile|view]
- executable;[execute]
- directory;[create|delete]

Targets
-------

The targets used in queries must be found in the TARGET table. A typical set of targets would be the projects, record classes and fields of another Daytona database. For example, the orders example supplied with Daytona contains the following three targets, among many others: orders (a project), orders.PART (a record_class), and orders.PART.Number (a field). Targets are hierarchical and the hierarchy is indicated by the Children field in TARGET.

Lists
-----

Within queries there may be lists of actions, targets, agents, networks, domains, intervals, and options. For example:

grant actions select, update to chad, eddy on orders

contains a list of actions and a list of agents (as grantees). Elements of a list are separated by commas. A list is indicated with one of: actions-list, targets-list, agents-list, networks-list, domains-list, intervals-list, or options-list.

Wildcards
---------
For networks, domains, and intervals, the corresponding wild cards are 0, *, and @. Unspecified networks, domains, and intervals default in queries to the appropriate wild card. For domains the wildcard .* may also be specified. It is equivalent to *, except that in AUTHORIZED?, - or the absence of an input is not a match. For actions, ‘all’ is the wildcard. There are no wildcards for agents, targets, target kinds, or roles.

In revoke and deny queries, ‘any’ is used to indicate all actions, targets, agents, roles, networks, domains, or intervals already in the Arthur database that apply. It is used, rather than use ‘all’, because ‘all’ may be one of the items to be revoked or denied, and using ‘all’ is reserved for revoking or denying that item.

Cascade and restrict
---------------------

Some queries may have consequences that affect other parts of the Arthur database. For example, dropping a role that is granted to an agent would leave the agent with the grant of a non-existent role. This is not allowed in Arthur and will produce an error unless ‘cascade’ is used in the query, in which case the grant of the role will also be removed. The default ‘restrict’ may be specified explicitly if desired.

Further granting
-----------------

Privileges and roles may be granted in such a way that the grantee may himself make grants of the privilege or role to others. If a grant is made ‘with grant option’ then the grantee may make further grants, but not with any additional option. If a grant is made ‘with grantability option’ then the grantee may make further grants ‘with grant option’. If a grant is made ‘with supergrantability option’, then the grantee may make further grants ‘with supergrantability option’. Note that the last is the only option that can be further granted through an unlimited chain of grantees.

Other
-----

The pipe | is used to indicate alternative constructions of queries.

[] indicates an optional element or character. (default) indicates the default value of a parameter.

Below is the syntax for queries and notes on their use:

grant action[s] actions-list | grant actions all
on targets-list
to agents-list
A privilege is an action on a target from a network and from a domain during an interval. This query grants privileges to agents with (optionally) further granting options.

revoke

This query removes privileges granted with the grant query.

define role <rolename>

to include action[s] actions-list | to include action[s] all on targets-list

A role is just a named set of privileges. Multiple 'define role' queries may be used to add privileges to a role. The grantor becomes the creator of the role upon the execution of the first query defining it and is the only agent allowed to add further inclusions via this query.

Any agent can create a role on any target, but may grant the role (the entire set of privileges) only if the agent may grant each of the contained privileges.

The grantee of a role is authorized to exercise each privilege within the role. If given an option to further grant the role, the grantee may only grant the entire role, not any individual privilege it contains.
This query removes privileges from a role. Only the creator of the role may use it.

define role <rolename>
  to deny action[s] actions-list | to deny actions[s] any
  on targets-list | on any
  [ from network[s] networks-list | from network[s] any (default) ]
  [ from domain[s] domains-list | from domain[s] any (default) ]
  [ during intervals-list | during any (default) ]
  restrict (default) | cascade

This query negates individual privileges within a role. It is most useful when one wishes to include all but a small number of actions, networks, domains, or intervals. For example, consider:

define role R to include action select on orders from network 0
  from domain * during @

If one wished, in the role, to deny a particular action from a particular network and a particular domain during a particular time, one might use:

define role R to deny action select on orders from network 1.2.3.0/24
  from domain uptonogood.com during @11PM->7AM

Now agents granted this role will not be authorized to take action select on orders from the specified network and domain during the specified interval. The most common use of this query is likely to be the following:

define role R to include actions all on orders
define role R to deny action delete on orders

Granting all actions and then denying a few is much more efficient than granting all but the few. This is the reason for the existence of this feature.
This query is used to remove denials within a role. Only the creator of the role may use it.

define role <rolename1>
  to include role[s] <rolename2>
  [ as model ]

This query includes (the privileges of) one role within another. The including role may be granted by the creator only when he is authorized to grant all included roles and privileges. The grantee of a role may make grants of it if given an option to do so.

define role <rolename1>
  to exclude role[s] <rolename2> | to exclude role[s] any
  [ restrict (default) | cascade ]

This query removes the inclusion of a role in another. Only the creator of the role may use it.

drop role <rolename>
  [ restrict (default) | cascade ]
  [ as model ]

This query removes all inclusions in a role, both privileges and roles, effectively removing it. Only the creator of the role may use it.

grant role[s] roles-list to agents-list
  [ with grant option | with grantability option |
    with supergrantability option ]

This query is used to grant roles to agents.

revoke
  [ grant option for | grantability option for |
    supergrantability option for ]
  role[s] roles-list | role[s] any
  from agents-list | from any
  [ restrict (default) | cascade ]

This query is used to remove grants of roles to agents.
ADMINISTRATIVE COMMANDS
=======================

Arthur also provides a command-line interface to facilitate the administration of an Arthur database instance. Many of these command scripts, found in $DS_ARTHURDIR/bin, translate directly to the preceding command protocol, but provide an additional measure of convenience.

**arthur_check_env**

- Validate the environment for an Arthur instance. This check is done in all of the Arthur command scripts.

**arthur_init**

- Initialize the tables for an Arthur instance.
- This command should only be called once per instance.

**arthur_add_agent** <agent> <services> [[<networks> [<domains> [<password>]]]]

**arthur_drop_agent** <agent>

**arthur_show_agent** <agent>

- Add/Drop/Show a given Arthur agent.
- When adding an agent, comma-separated lists may be given for <services>, <networks>, and <domains>. The defaults for the optional <networks> and <domains> are '0' and '*', respectively, which indicate that all values are accepted. Note that when a constraint list is given for both <networks> and <domains>, both constraints must be met. Also, a password must always be provided. If it is not given on the command line, then the script will prompt for it interactively.
- When showing an agent, a '-' argument will show results for all agents.

**arthur_update_agent_password** <agent> [<password>]

- Update the password for a given Arthur agent.
- A password must always be provided. If it is not given on the command line, then the script will prompt for it interactively.

**arthur_add_agent_service** <agent> <services>

**arthur_drop_agent_service** <agent> <services>

**arthur_update_agent_service** <agent> <services>

- Add/Drop/Update the services for a given Arthur agent.
- A comma-separated list may be given for <services>.

**arthur_add_agent_network** <agent> <networks>

**arthur_drop_agent_network** <agent> <networks>

**arthur_update_agent_network** <agent> <networks>

- Add/Drop/Update the networks for a given Arthur agent.
- A comma-separated list may be given for <networks>.
arthur_add_agent_domain <agent> <domains>
arthur_drop_agent_domain <agent> <domains>
arthur_update_agent_domain <agent> <domains>

- Add/Drop/Update the domains for a given Arthur agent.
- A comma-separated list may be given for <domains>.

arthur_add_agent_target <agent> <target> <target_kind>
arthur_drop_agent_target <agent> <target>
arthur_show_agent_target <agent> [<target_kind>]

- Add/Drop/Show targets for a given Arthur agent.
- When adding an agent target, its <target_kind> must be provided.
- When showing agent targets, a '-' for the agent will show results
  for all agents. A specific <target_kind> may be given to constrain
  the results.

arthur_grant_agent_target <agent> <target> [<actions> [<target_owner>]]
arthur_revoke_agent_target <agent> <target> [<target_owner>]
arthur_show_agent_target_grants <agent> [<target>]

- Grant/Revoke target access to/from a given Arthur agent, or
  Show target access granted to a given Arthur agent.
- When granting or revoking access, a <target_owner> should be provided
  when a particular Arthur agent is desired, otherwise the default is
  '_arthur_'.
- A comma-separated list can be given for <actions>, otherwise it
defaults to 'all'.
- When showing agent grants, a '-' for the agent will show results
  for all agents. A specific <target> may be given to constrain the
  results.

arthur_grant_agent_role <agent> <role> [<role_owner>]
arthur_revoke_agent_role <agent> <role> [<role_owner>]
arthur_show_agent_role_grants <agent> [<role>]

- Grant/Revoke role access to/from a given Arthur agent, or
  Show role access granted to a given Arthur agent.
- When granting or revoking access, a <role_owner> should be provided
  when a particular Arthur agent is desired, otherwise the default is
  '_arthur_'.
- When showing agent grants, a '-' for the agent will show results
  for all agents. A specific <role> may be given to constrain the
  results.

arthur_add_role_target <role> <target> [<actions> [<role_owner>]]
arthur_drop_role_target <role> <target> [<role_owner>]

- Add/Drop targets for a given Arthur role.
- A comma-separated list can be given for <actions>, otherwise it
arthur_add_role_role <role> <role-2> [role_owner]
arthur_drop_role_role <role> <role-2> [role_owner]

- Add/Drop a subordinate role (role-2) to/from a given Arthur role.
- A role_owner should be provided when a particular Arthur agent is desired, otherwise the default is '_arthur_'.

arthur_drop_role <role> [role_owner]
arthur_show_role <role>

- Drop/Show a given Arthur role.
- When dropping a role, a role_owner should be provided when a particular Arthur agent is desired, otherwise the default is '_arthur_'.

arthur_show_all_targets_for_agent <agent> [target_kind]
arthur_show_all_roles_for_agent <agent>

- Show all targets/roles for a given Arthur agent.
- When showing targets, a specific target_kind may be given to constrain the results.

arthur_show_all_targets_for_role <role> [target_kind]

- Show all targets for a given Arthur role.
- A specific target_kind may be given to constrain the results.

arthur_add_target_kind target_kind [actions]
arthur_drop_target_kind target_kind

- Add/Drop a given Arthur target_kind.
- When adding a target kind, a comma-separated list can be given for <actions>.

arthur_add_proj_targets

- Synonymous with the 'Add_Project_Targets' command.
- When a project database is being treated as a collection of accessible targets, this command can be used to populate an Arthur database instance with these targets.
- DS_PROJ must be defined in the environment and there must be a corresponding 'par' file associated with the project.
- This command should only be called once per project.
NAME

DBD::Daytona – Perl Module for a DBI driver for DBMS Daytona

SYNOPSIS

The Perl DBI Driver for Daytona, also known as DBD::Daytona, lets the user run database queries in a client-server environment using Perl. Perl is used on the client side while the Daytona Server pdq handles database requests on the server side.

A user’s Perl script, containing a database connection request and a database query, resides on the client. The database and the Daytona Server pdq reside on a (possibly) remote server. A user’s Perl script requests a connection to the pdq server using either a machine name or an IP address together with a port number.

To run a Perl script using Perl’s DBI and the DBD::Daytona driver to query a Daytona database, four things are needed:

- Perl script(s) on the client using the Perl DBI Driver for Daytona
- Access to the client software, including compatible versions of Perl, the Perl DBI Module, and the Perl DBI Driver for Daytona, as facilitated by establishing the appropriate shell environment
- Access to a Daytona Server pdq via a network connection
- Authorization for pdq access and services as validated by Arthur (a Daytona module)

Sample Perl Script Using the Perl DBI Driver for Daytona

Here is a small example using Perl’s DBI and DBD::Daytona to query a Daytona database using SQL, which could be saved in file mytest.pl as is:

```perl
use DBI;
$city = "St. Paul";
$dbh = DBI->connect("DBI:Daytona:", "mylogin", "mypasswd" );
$sth = $dbh->prepare("select Name, Telephone from SUPPLIER where City = ? ");
$sth->execute($city);
$sth->bind_columns(undef, \$name, \$telephone);
while( $sth->fetch() ){
    print( "$name $telephone\n" );
}
$sth->finish();
$dbh->disconnect();
```

Choose non-null entries for the connect statement arguments mylogin and mypasswd above. Notice that the Daytona DBI does not require any preparation of arguments. It doesn’t matter what the syntax of SQL constants is and it doesn’t matter if the argument is a number or a string to Perl or if Daytona is expecting values for any of its many different kinds of types: just provide a value that looks like some value that Daytona would write out (to stdout) or read in (from stdin) for the appropriate type.

The query here is written in SQL but could also be written completely in Daytona’s Cymbal or even as a hybrid of Cymbal and SQL. The latter form could enclose a sequence of SQL statements in Cymbal’s DSQL delimiters $[ ... ]$. Of course, if Perl functions like bind_columns are used, then whatever form the query takes, the resultant output must be the same as if an SQL select or a Cymbal Display had been executed, since the presumption is that a conventional table is being retrieved.

Assume that the Daytona pdq is already running on the localhost and the default port and that pdq is supporting the sample database shipped with Daytona. Also, assume that the user is in the Daytona environment and is authorized by Arthur for Perl DBI (pdbi) PDQ requests, that export PERL5LIB=$(getdsenv DS_PERLDIR)/lib has been done, and that perl is (perhaps an alias that refers to) a version of Perl compatible with the Daytona DBI driver. Then simply invoking perl mytest.pl on the file mytest.pl containing the above Perl code will return the names and phone numbers of all Daytona Supplier records where City is St. Paul.
numbers of St. Paul suppliers in the SUPPLIER table of the sample database shipped with Daytona:

Acme Shipping  612-149-5678
Bouzouki Receiving  612-943-7416
Julius Receiving  612-309-3492
Sunshine Warehouse  612-303-7074

It’s that easy! And now on to a little more detail.

**Access to Perl, Perl DBI, and Perl DBI Driver for Daytona on the Client**

On the client, the user’s Perl script needs to be able to find the Perl DBI Driver for Daytona and Perl DBI. To help with such tasks, Daytona maintains some environment settings for the Perl DBI Driver for Dayton; to see what they are, use:

```
# Directory containing bin/perl
getdsenv DS_PERLHOME
# Directory for Perl DBI Driver for Daytona
getdsenv DS_PERLDIR
# or simply
DS Env \ grep PERL
```

To tell all subsequent invocations of the Perl interpreter where the Perl DBI Driver for Daytona and the Perl DBI are located, use the following command:

```
export PERL5LIB=$(getdsenv DS_PERLDIR)/lib
```

To use a Perl interpreter that is compatible with the Perl DBI Driver for Daytona, the Perl DBI, and its prerequisites (see INSTALLATION section below), run client perl scripts that use them with the perl under DS_PERLHOME:

```
$(getdsenv DS_PERLHOME)/bin/perl mytest.pl
```

You can get its version by using either:

```
$(getdsenv DS_PERLHOME)/bin/perl -v  # brief version
$(getdsenv DS_PERLHOME)/bin/perl -V  # verbose
```

The reported version of Perl is compatible with the one used to build the Perl DBI Driver for Daytona. For safety, always use the Perl under DS_PERLHOME when using the Perl DBI Driver for Daytona.

As of this writing, the Perl version used for the Perl DBI Driver for Dayton is v5.8.4 and is recommended. (Perl v5.8.2 is required as a minimum.)

**Access to the Dayton Server pdq**

The Dayton Server pdq must be accessible over a TCP network connection using a machine name or IP address together with a port number.

For testing, localhost:14015 or 127.0.0.1:14015 is used as a default with the Dayton sample project daytona database.

To check if pdq is running on a machine where you have access to Dayton, log on to that machine and use the following commands:

```
# check Dayton environment
DS Env
# check if PDQ is running
$(getdsenv DS_SERVERDIR)/pdq/bin/ps.pdq
```

If the Dayton Server pdq is not running, see the beginning of the EXAMPLES section below for more information.
Further details on starting the Daytona Server `pdq` are available in the `pdq` documentation via either of the commands:

```
DS Man pdq
MANPATH=$(getdsenv DS_DIR)/DOCS/man:$MANPATH man pdq
```
or see `pdq(8)` in the Man Pages Appendix of *All About Daytona* by R. L. Greer, AT&T Laboratories.

**Arthur: Authorization and Security for PDQ**

`pdq` relies on `Arthur` to help validate request according to various criteria, such as network address, type of service (e.g., `pdbi`), and user information. More information on `Arthur` is available with the command:

```
DS Man Arthur
```

Note that `Arthur` needs to run on the same machine as `pdq`.

Further details are given in the Examples below.

**Running the Sample Query**

Assuming the sample query above is in file `mytest.pl` a Daytona Server `pdq` is running on localhost:14015, and Arthur is ready to give `pdq` the okay on your behalf, use the following commands to run the client Perl query script:

```
trntwn:> cd mytestdir
trntwn:> export PERL5LIB=$(getdsenv DS_PERLDIR)/lib
trntwn:> $(getdsenv DS_PERLHOME)/bin/perl mytest.pl
```

```
Acme Shipping 612-149-5678
Bouzouki Receiving 612-943-7416
Julius Receiving 612-309-3492
Sunshine Warehouse 612-303-7074
```

Note that it takes a few seconds to obtain the results for the query since, in this case, the query text is sent to the Daytona Server `pdq` which must compile the query, execute it, and then return the results to the client for display. However, the compilation step can be avoided by using Daytona’s Perl DBI stored procedure feature as described below.

For more examples and how to run them, see EXAMPLES below.

**DESCRIPTION OF DBI DRIVER Daytona.pm**

This module and documentation are intended to help Perl and Daytona users access Daytona databases from Perl scripts.

```
“DBI is the standard database interface for the Perl programming language. The DBI is database-independent, which means that it can work with just about any database...” (see Preface to Programming the Perl DBI by Alligator Descartes and Tim Bunce, the definitive book on the subject with which the reader is assumed to be familiar).
```

However, in order for Perl’s DBI to work with AT&T’s DBMS Daytona, DBI needs a *driver* specifically for Daytona. This `DBD::Daytona` module is just that — a DBI driver for Daytona written completely in Perl.

As noted in `DBI(3)`, DBI “itself does not mandate or require any particular language to be used; it is language independent. In ODBC terms, the DBI is in ‘pass–thru’ mode, although individual drivers might not be.” All Daytona queries, including those handled by DBI and driver `DBD::Daytona`, may be written in `SQL`, in `Cymbal`, or a combination of both (see *All About Daytona*).

In order to use the Perl `DBD::Daytona` driver, Daytona, the Daytona Server `pdq`, and Daytona’s Authenticator `Arthur` need to be installed with `pdq` running on a known port for a given database application. There is no need for client Perl scripts to run on the same machine as the Daytona DBMS
The Daytona Server *pdq*, the Daytona Authenticator *Arthur*, and the Perl module *DBD::Daytona* (in *Daytona.pm*) ship with Daytona under their own subdirectories:

- **pdq**
  
  `$DS_SERVERDIR/pdq/bin`, usually `$DS_DIR/SERVER/pdq/bin`

- **Arthur**
  
  `$DS_ARTHURDIR/bin`, usually `$DS_DIR/ARTHUR/bin`

- **Daytona.pm**
  
  `$DS_PERLDIR/lib/DBD`, usually `$DS_DIR/PERL/lib/DBD`

See *DBI* (3) for information on DBI, *DBI::DBD* (3) for writing DBD drivers, and *All About Daytona* for Daytona’s SQL (*DSQL*), for procedural and declarative (non-procedural) *Cymbal*, for the Daytona *pdq* server, and for *Arthur*.

**Driver Attributes Common To All Handles**

The following DBI attributes are handled by DBI itself and not by *DBD::Daytona*; thus they all work as expected:

- Active
- ActiveKids
- CachedKids
- CompatMode (Not used)
- InactiveDestroy
- Kids
- PrintError
- RaiseError
- Warn (Not used)

The following DBI attributes are handled by *DBD::Daytona*

- **AutoCommit**
  
  Always on.

- **ChopBlanks**
  
  Works although Daytona string fields do not typically end with blanks.

- **NUM_OF_PARAMS**
  
  Valid after `$sth->prepare`.

- **NUM_OF_FIELDS**
  
  Valid after `$sth->execute`.

- **NAME**
  
  Valid after `$sth->execute`; undefined for Non-Select statements.

- **NULLABLE**
  
  Not really working, always returns an array ref of one's, as *DBD::CSV* doesn’t verify input data. Valid after `$sth->execute`; undefined for Non-Select statements.

These attributes and methods are not supported:

- `bind_param_inout`
- `CursorName`
- `LongReadLen`
- `LongTruncOk`
New Daytona-Specific Attributes

In addition to the DBI attributes, Daytona-specific handle attributes starting with the prefix `day_` may be used:

**day_host**

These attributes for a database handle provide a host name or host port to the `connect()` method, respectively. Default value is `host=localhost` or `port=14015`, respectively. See method `connect` below.

**day_port**

These attributes for a database handle provide a host name or host port to the `connect()` method, respectively. Default value is `host=localhost` or `port=14015`, respectively. See method `connect` below.

**day_project**

This attribute for a database handle provides a Daytona database project name to the `connect()` method. Default value is `day_project=daytona`.

Values for attributes `day_host`, `day_port`, and/or `day_project` are set in the first argument of the `connect()` method as follows:

```perl
my $dbh1 = DBI->connect(
    "DBI:Daytona:project=daytona;host=localhost;port=14015",
    "user", "password" );
```

or

```perl
my $dbh1 = DBI->connect(
    "DBI:Daytona:daytona@63.251.83.40:14015",
    "user", "password" );
```

In the latter form, the @ sign must be escaped with the backslash if double quotes are used because otherwise Perl will consider that as the beginning of an array variable. If the host is not explicitly mentioned, it is assumed to be `localhost` (i.e., `127.0.0.1`). If the port is not explicitly given, it is assumed to be `14015`.

See also the `connect()` method in the *Daytona Driver Notes* below. Note that the `day_` prefix does not appear in the data source name used by `connect()`.

**day_client**

This attribute for a database handle contains the ‘class’ name of the client. Valid requests sent to the Daytona database server `pdq` contain the name of the client as well as the login ID of the user. `pdq` uses the client name to tune its response to the needs of the client. Default value for Perl DBI is `day_client='pdbi'`.

**day_recls**

**day_type**

These two attributes provide extensions to the method `table_info()`, used to get information on the available database tables (or Daytonas RECORD CLASSES).

To obtain a listing (via a statement handle) showing basic information on available tables, use:

```perl
my $tables = $dbh->table();
```

To obtain information on a given table's fields, use:

```perl
my $fld_sth = $dbh->table_info( { day_recls=>'SUPPLIER',
                                    day_type=>'fields', } );
```

$fld_sth->[‘NAME’]->[‘$i’] for $i increasing from 0 returns COLUMN_NAME, COLUMN_TYPE, COLUMN_SIZE, COLUMN_DEFAULT, REMARKS, resp.

To obtain information on a given table’s keys, use:

```perl
my $key_sth = $dbh->table_info( { day_recls=>'PERSONI',
                                    day_type=>'keys', } );
```
$key_sth->{'NAME'}->[i] for $i increasing from 0 returns KEY_NAME, KEY_UNIQUE, KEY_FIELDS, REMARKS, resp.

To obtain information on the Daytona server’s environmental shell variables, use:

my $env_sth = $dbh->table_info( { day_type=>'env', } );

$env_sth->{'NAME'}->[0], $env_sth->{'NAME'}->[1] refer to the shell variable and its value, resp.

day_qname
To provide a name for a query, set this statement handle attribute in the prepare() method:

my $sth = $dbh->prepare( $query_text, { day_qname=>'Get_Supplier_Info', } );

day_qname enables stored procedures (see below).

day_rows
This statement handle attribute returns the number of rows produced by the preceding execute statement.

day_pdq_prepare
day_pdq_execute
These debugging statement handle attributes show what was sent to the pdq server as a result of executing the respective Perl DBI methods.

Daytona DBI Methods

bind_param() method
Parameters are indicated by question marks as illustrated by the sample query at the start of this document. If it is necessary for an SQL or Cymbal constant to contain a question mark, simply prefix the question mark with a backslash.

Due to the limitations of regular expressions, when ? placeholders are used, any C-style comments present must end with an odd number of asterisks before the final slash; otherwise, the comment will not be handled correctly. /* hello */ is such a well-behaved comment. Of course, ANSI SQL does not support C-style comments anyway. The user is also welcome to use the official ANSI SQL comments starting with -- as well as the C++ double-slash comments.

connect() method
The minimal arguments to connect() to connect to a Daytona database are:

my $dbh = DBI->connect( "DBI:Daytona:", "user", "passwd" );

Note that the first argument to the connect() method must be given as:

DBI:Daytona:

Be sure to include the second colon!

Also note that for Daytona, a valid user ID and password are required as the second and third arguments to the connect() method.

If you want to work with other Daytona hosts or projects, supply the required values for the DBD::Daytona attributes day_host and/or day_port. day_host may be specified as a name, such as trntwn001.dept.company.com or trntwn001, or in octet form, such as 63.251.83.40. The Daytona pdq server requires that the day_port number selected be over 12500; be sure to pick a unique port number for your database.

For example:
my $dbh1 = DBI->connect(
    "DBI:Daytona:project=my_proj;host=localhost;port=16776",
    "user", "password" );

connects to the Daytona project my_proj on the local machine via port 16776. Note the absence of any day_ prefixes to the keywords in the first argument to connect. Note the use of semicolons between the attribute=value pairs inside the first double-quoted argument constituting the data source name. If the host is not explicitly mentioned, it is assumed to be localhost (i.e., 127.0.0.1). If the port is not explicitly given, it is assumed to be 14015.

The Perl DBI client connects to a specified Daytona server pdq given host and port. That server was started with a given Daytona environment (see the command DS Env) that might or might not contain a value for DS_PROJ, but should contain a value for DS_APPS, etc. At this time, day_project is not used.

data_sources() method

The data_sources() method returns a list of the Data Source Names in the form:

    DBI:Daytona:localhost:14015:daytona

Stored Procedures

A stored procedure is the result of pre-processing a query and then reusing that pre-processed query repeatedly later, thus saving the pre-processing time. In the case of Daytona, pre-processing means compiling the SQL/Cymbal query to a named executable, which "stores the procedure". Invoking that executable, perhaps from a different program and perhaps using runtime arguments, constitutes reusing that stored procedure. Such a strategy will save the user the few seconds it takes Daytona to compile queries.

To use Daytona’s Perl DBI to create a stored procedure, simply call prepare with a day_qname argument that will serve to name the stored procedure. Here is an example:

```perl
my ($query) =
    "SELECT SUPPLIER.Name AS Supplier, SUPPLIER.Telephone AS Phone
     FROM SUPPLIER
     WHERE SUPPLIER.City Matches ?
     AND SUPPLIER.Telephone Matches ?
     AND SUPPLIER.Number < ?
     ORDER BY SUPPLIER.Telephone DESC \n";
my ($sth) = $dbh->prepare( $query, {day_qname=>'Supp_Phone3'} );
```

Then to run the stored procedure later, just prepare an empty query with day_qname set to the desired stored procedure and execute with any required runtime arguments as in:

```perl
$sth = $dbh->prepare( "", {day_qname=>'Supp_Phone3'} );
$sth->execute( $city, $telephone, 425 );
```

It’s as simple as that. See Examples for test files to run.

EXAMPLES

The reader is assumed to be familiar with Perl DBI programming as described in the definitive DBI reference the book Programming the Perl DBI.

Clients Using the Perl DBI Driver for Daytona

DS_PERLHOME: Home of a compatible bin/perl

To run these samples, it is necessary to use a version of Perl compatible with the Driver, the Perl DBI Driver Manager, and its prerequisites (see INSTALLATION below). This can be accomplished by using the command:
$(getdsenv DS_PERLHOME)/bin/perl
You may want to add $(getdsenv DS_PERLHOME)/bin to the front of your shell $PATH variable:

trntwn:> PATH=$(getdsenv DS_PERLHOME)/bin:$PATH

PERL5LIB: Where to Find Perl DBI and Driver for Daytona

It is also necessary to tell the Perl interpreter where to look for the Perl DBI Driver for Daytona, the Perl DBI Driver Manager, and its prerequisites. This can be done with the PERL5LIB variable, which the Perl interpreter adds to its directory search path at start-up (even though it may not use the information for some time):

trntwn:> export PERL5LIB=$(getdsenv DS_PERLDIR)/lib

Starting the Daytona Server PDQ

See the pdq(8) man page for an example of how to start the pdq server and all about pdq.

Get the latest information with the command:

DS Man pdq

As pdq goes about its work of handling requests, such as compiling queries, executing queries (perhaps with given sets of parameters), and returning query results, it creates subdirectories and files to organize its workspace. It may be convenient to create a directory in which to run pdq, perhaps with subdirectories for one pdq handling local (localhost) requests and another for another pdq handling network request from other machines:

# directories in which to run PDQ
dpq
    pdq/local  # a PDQ here handles local requests on port x
    pdq/net   # a PDQ here handles network requests on port y

Remember to start each pdq using a unique port number.

The steps to start pdq can be reviewed by running the command:

$DS_SERVERDIR/pdq/bin/start.pdq -?

These are the steps for starting pdq to handle local request on port 14028:

    # add DS_SERVERDIR to your shell environment
    trntwn:d> DS_SERVERDIR=$(DS_DIR)/getdsenv DS_SERVERDIR
    # move to pdq directory for handling local requests
    trntwn:d> cd pdq/local
    # add DS_SERVERDIR to shell PATH
    trntwn:local> export PATH=DS_SERVERDIR/pdq/bin:$PATH
    # start PDQ with log trace level 3 for localhost on port 14028
    trntwn:local> nohup pdq -trace 3 -host 127.0.0.1 -port 14028 &

These are the steps for starting pdq to handle network request on a different port, 14029:

    # add DS_SERVERDIR to your shell environment
    trntwn:d> DS_SERVERDIR=$(DS_DIR)/getdsenv DS_SERVERDIR
    # move to pdq directory for handling local requests
    trntwn:d> cd pdq/net
    # add DS_SERVERDIR to shell PATH
    trntwn:net> export PATH=DS_SERVERDIR/pdq/bin:$PATH
    # start PDQ with log trace level 3 for net connects on port 14029
    trntwn:net> nohup pdq -trace 3 -host 63.251.83.40 -port 14029 &
nohup is used so `pdq' will continue running after the user who issued the command logs out.

To check that `pdq' is running, use the command `ps.pdq':

```
trntwn:net> $DS_SERVERDIR/pdq/bin/ps.pdq
usr_ds 1791 1 0 Sep 13 pts/013 0:00 \n   pdq -trace 3 -host 63.251.83.40 -port 14029
usr_ds 1779 1 0 Sep 13 pts/013 0:01 \n   pdq -trace 3 -host localhost -port 14028
```

**Authorizing `pdbi` Service on `PDQ`**

Get the latest Arthur information with the command:

```
DS Man Arthur
```

When the Daytona Server `pdq` is started, `Arthur` may be initialized to support `pdq` by issuing the command:

```
$DS_ARTHURDIR/bin/arthur_pdq_init
```

To prime `Arthur` to handle PDBI requests for a user ‘ptest’, the following command would be used (with the appropriate domain and password information given for the placeholders):

```
trntwn:net> $DS_ARTHURDIR/bin/arthur_add_user \n      ptest pdbi <net.domains> <password>
```

To check on Arthur, a shell ‘here–document’ can be used to ask what Arthur has for ‘pdbi’ agents:

```
trntwn:net> $DS_ARTHURDIR/bin/Arthur - <<!!
SHOW AGENT   -- pdbi   --
&&&&&
!!
```

**Perl Code Using the Perl DBI Driver for Daytona**

A number of sample Perl clients using the Perl DBI Driver for Daytona are in the directory:

```
$DS_PERLDIR/pdbi/test
```

The intent is for this collection of Perl scripts to contain an example of everything that works. This constitutes a very handy use-guide for the reader.

**Ping the Daytona Server `pdq`**

A sample program can be run to ping the Daytona Server `pdq`:

```
trntwn:> cd $(getdsenv DS_PERLDIR)/pdbi/test
trntwn:> export PERL5LIB=$(getdsenv DS_PERLDIR)/lib
trntwn:> $(getdsenv DS_PERLHOME)/bin/perl day_ping.pl
Driver: Daytona
...
OK  PDQ on myhost (127.0.0.1:14015) alive at Fri Jul 04 12:00:01 EDT 2003
...
```

**Query a Daytona Database**

To find the St. Paul Suppliers in Area Code 612, run the query in file `caac.pl` (city and area code); no arguments are required. Allow a few seconds for the query to be compiled by the Daytona Server `pdq`; the executable will then be executed and the results returned to the client for display:

```
trntwn:> cd $(getdsenv DS_PERLDIR)/pdbi/test
trntwn:> export PERL5LIB=$(getdsenv DS_PERLDIR)/lib
```
```plaintext
trntwn:> $(getdsenv DS_PERLHOME)/bin/perl caac.pl
...
:: Bouzouki Receiving       612-943-7416
     Julius Receiving        612-309-3492
     Sunshine Warehouse      612-303-7074
     Acme Shipping           612-149-5678
::END:: "That’s All, Folks!"
```

You may also try:

```plaintext
trntwn:> $(getdsenv DS_PERLHOME)/bin/perl caac.pl London 612
```

### Stored Procedures with Perl DBI and Daytona

First create a stored procedure by compiling a Daytona Query by ‘preparing’ a Perl DBI statement and saving it with a given name, as in file `storeaproc.1.pl`:

```perl
my($sth) = $dbh->prepare( $query, {day_qname=>'Supp_Phone3'} );
```

`storeaproc.1.pl` can be run to created the Stored Procedure `Supp_Phone3`, which takes three arguments, by using the commands:

```plaintext
trntwn:> cd $(getdsenv DS_PERLDIR)/pdbi/test
trntwn:> export PERL5LIB=$(getdsenv DS_PERLDIR)/lib
trntwn:> $(getdsenv DS_PERLHOME)/bin/perl storeaproc.1.pl 'St. Paul' '^612' 700
...
SQL Stored Procedure `Supp_Phone3` contains 3 parameters
...
###
pdq executed
###
Supp_Phone3 'St. Paul' '^612' 700
###
Title: Suppliers For St. Paul and Area Code ^612 and Supp ID Under 700
Columns: Supplier Phone
     Bouzouki Receiving       612-943-7416
     Julius Receiving        612-309-3492
     Sunshine Warehouse      612-303-7074
     Acme Shipping           612-149-5678
SQL statement returned 4 rows
```

Note that `storeaproc.1.pl` stores a procedure \textit{and} executes it, allowing the stored procedure to be verified with some sample output.

To use the Stored Procedure `Supp_Phone3` \textit{without} recompiling the query and use it with new input parameters, refer to it by its stored query name \textit{without} including query text and then execute it with the new desired arguments:

```perl
$sth = $dbh->prepare( "", {day_qname=>'Supp_Phone3'} );
$sth->execute( $city, $telephone, 699 );
```

To do so, use the following commands to run the companion Perl script in file `storedproc.1.pl`:

```plaintext
trntwn:> cd $(getdsenv DS_PERLDIR)/pdbi/test
trntwn:> export PERL5LIB=$(getdsenv DS_PERLDIR)/lib
```
trntwn:> $(getdsenv DS_PERLHOME)/bin/perl storedproc.1.pl 'St. Paul' '^612' 699
...  
  Args: storedproc St. Paul ^612 699
  pdq DS_EXECUTE +_now_+ for "Supp_Phone3" :
    ###
    Supp_Phone3 'St. Paul' '^612' 699
    ###
    Bouzouki Receiving 612-943-7416
    Julius Receiving 612-309-3492
    Sunshine Warehouse 612-303-7074
    Acme Shipping 612-149-5678
    ###

Note that the same four Suppliers (and Phone Numbers) are returned. Feel free to try other arguments, such as:

trntwn:> $(getdsenv DS_PERLHOME)/bin/perl storedproc.1.pl 'New York' '.' 700

The '.' dot for Area Code is short-hand for any character, or, in other words, an Area Code with anything in it (i.e., don’t care).

An Interactive Demo

For an interactive demo, run:

trntwn:> cd $(getdsenv DS_PERLDIR)/pdbi/test
trntwn:> export PERL5LIB=$(getdsenv DS_PERLDIR)/lib
trntwn:> $(getdsenv DS_PERLHOME)/bin/perl doQuery.pl

EFFICIENCY

For efficient query handling, direct use of Daytona’s Cymbal language is recommended. Cymbal is powerful, a high-level 4GL, contains the full query-and-update portion of ANSI standard SQL as a subset (known as DSQL), and encompasses both procedural and declarative (non-procedural) constructs. Dayton has its own data dictionary capabilities, that are more extensive than those in SQL.

On the other hand, for those who love Perl, DBI and DBD::Daytona enable Perl to take high-level advantage of the power of Daytona ... further enhancing “Perl programmer efficiency.”

Due to the overhead of communications between the client and server in the client-server model, it is generally preferable to design client-side queries that ‘minimize’ the amount of generated query output that will be sent back from the server to the client over the connection or optimize the trade-offs between server-side processing, communications, and client-side processing.

LANGUAGE

DBD::Daytona is written as a pure Perl driver (no C code is used). It communicates via TCP and sockets with the Daytona database Server pdq, which is written in Cymbal.

Note, however, that the Perl Database Interface, DBI, the Driver Manager that provides a unified interface for Perl scripts to diverse databases, and its prerequisites, do contain some C code.

INSTALLATION

The Perl DBI Driver for Dayton, DBD::Daytona, is normally included in the Dayton release and installation process.

To check if the Perl DBI Driver for Dayton is installed on a platform that also has Dayton, use the following commands, which should generate non-empty output something like:

trntwn:> DS Env | grep PERL
DS_PERLDIR = '/usr/local/daytona/ds_bin/PERL'
DS_PERLHOME = '/usr/common/perl5.8.4'
trntwn:> find $(getdsenv DS_PERLDIR)/lib -name Daytona.pm -print
/usr/local/daytona/ds_bin/PERL/lib/DBD/Daytona.pm

The output of the find command above may also be expressed as
$DS_PERLDIR/lib/DBD/Daytona.pm; the non-null value indicates the Perl DBI Driver for
Daytona is installed and available for use on the platform. It may be used to access (possibly) remote
(and localhost) Daytona databases.

Installation Details

Most users of the Perl DBI Driver for Daytona may skip this section, if they like. It provides
information on the prerequisites for the driver, a high-level look at the make process, and general
guidelines on how build the driver on clients where Daytona is not installed.

Although the Perl DBI Driver for Daytona is itself written entirely in Perl, it depends on the Perl
DataBase Interface, DBI, a Driver Manager that provides a unified interface for Perl scripts to diverse
databases. Here is the current dependency chain for the Perl DBI Driver for Daytona:

- Perl DBI Driver Manager
- DBD-Multiplex
- PIRPC
- Net-Daemon
- Storable
- DBD-Shell
- IO-Tee, and
- Text-Reform

All of the above are Perl modules (some also with some C code) available from the Perl
Comprehensive Perl Archive Network CPAN website at:

http://www.cpan.org  # website for Perl modules: DBI.pm ...

Note, however, that these modules contain an implicit chain of version dependencies, including the
version of Perl itself. For this reason, the Daytona environment contains the environment variable
DS_PERLHOME, the directory containing the bin directory of a Perl executable perl that is
compatible with the Perl DBI Driver for Daytona, the DBI, and its prerequisites.

Use the following command to determine the current version of Perl:

trntwn:> # for more info, use: perl -V
trntwn:> $(getdsenv DS_PERLHOME)/bin/perl -v
This is perl, v5.8.4 built for mybox
Copyright 1987-2004, Larry Wall ...

As indicated above, the Perl DBI Driver for Daytona has been built and tested with Perl v5.8.4.

Installing the Perl DBI Driver for Dayton on a Client

As part of the Daytona distribution, the directory

$ (getdsenv DS_PERLDIR)/dist

contains tarred and zipped distribution files of the Perl DBI Driver for Daytona, the Perl DBI, and its
prerequisites all of which are mutually compatible. All but the file for the Perl DBI Driver for Daytona
are from the CPAN site and all file names follow the CPAN convention of including the version number.

These files may be used to build the Perl DBI Driver for Daytona on client platforms where Daytona is
not present but where Perl and C are. Copy them all over to the client into a client-side directory, such as PERL/dist.
A conservative approach is to unpack the driver’s makefile on the client platform in an appropriate
directory and then let it do the work:

```
tntnwn:> mkdir PERL PERL/dist PERL/pdbi
trntwn:> ## copy # host $DS_PERLDIR/dist/* --> PERL/dist # client-side
trntwn:> cd PERL/dist
trntwn:> scp -p ............
trntwn:> # unpack driver’s makefile
trntwn:> cd PERL/pdbi
trntwn:> gunzip -c ../dist/PDBI-Daytona-*.tar.gz | tar xvf - ./makefile
trntwn:> # get essentials
trntwn:> make -f makefile fresh
trntwn:> # make Perl DBI Driver for Daytona
trntwn:> nohup make -f makefile clean install &
```

That avoids the tedium of working through the dependency list (bottom up in the list above, from Text-
Reform on up to DBI and Daytona), and using a compatible version of perl to follow the usual CPAN
Perl module installation process multiple times:

```
perl Makefile.PL
make
make test
make install
```

(In reality, the first command above is augmented with additional installation environment
information:

```
perl Makefile.PL \n    LIB=$DS_PERLDIR/lib \n    PREFIX=$DS_PERLDIR/lib \n    INSTALLMAN1DIR=$DS_PERLDIR/man/man1 \n    INSTALLMAN3DIR=$DS_PERLDIR/man/man3 \n    INSTALLBIN=$DS_PERLDIR/bin \n    INSTALLSCRIPT=$DS_PERLDIR/scripts
```

as well as with C compiler information. See perl doc or man pages for dbi-dbdoc.doc and
ExtUtils::MakeMaker.)

Once the installation is done, assuming pdq is running on port 14055 for the sample daytona project
and that Arthur has been duly alerted (see subsection Arthur: Authorization and Security for PDQ
above), all tests can be run for all the prerequisite modules, the DBI, and the Driver using:

```
tntwn:> # run DBI and driver tests
trntwn:> cd PERL/pdbi
trntwn:> nohup make -f makefile testEm on_PORT=14055 &
```

For running individual driver tests, see the EXAMPLES Section.

**Pesky Perl Files**

In the normal course of testing the Perl DBI and its prerequisites, .exist files may be generated under
b1ib directories. Permissions are such that even for the owner, it can be difficult to remove these files
and directories, such as to clean up for a new edition.

The solution is to use the following commands to remove the offending files so that the owner is free to
administer the directory space:

```
myclient:> cd $(getdsenv DS_PERLDIR)/pdbi
myclient:> make -f makefile clean
```
This is more through than the prerequisites’ make `realclean` commands, which proves insufficient.

AUTHORS
J. J. Snyder, Consultant, AT&T Labs – Research.
R. L. Greer

COPYRIGHT
This module is Copyright (C) 2000 (C) 2001 (C) 2002 (C) 2003 (C) 2004 by
AT&T Labs – Research, U.S.A.
All rights reserved.
You may obtain this module under AT&T’s terms from AT&T by contacting:
R. L. Greer, AT&T Labs – Research.

ACKNOWLEDGEMENTS
This work is based on and with many thanks to the authors of:

Perl5

Perl DBI

DBD-Csv
The Perl modules DBD-Csv–0.1022 and file File.pm by J. Wiedmann, Germany.

Daytona DBMS
The AT&T Daytona documents Getting Started with Daytona, Daytona User’s Guide, and All
About Daytona by R. L. Greer, AT&T Labs – Research.

pdq (8), the man page for the Daytona Server pdq, by J. J. Snyder and P. E. Brown, AT&T Labs –

Arthur (8), the man page for the Daytona Authenticator Arthur, by S. Olmsted, AT&T Labs –

SEE ALSO
perl (1), DBI (3), DBI::DBD (3), All About Daytona. and pdq (8).

DS Man pdbi, DS Man pdq, DS Man Arthur
NAME
jdbc – A Java JDBC Driver Package for Daytona

SYNOPSIS
$JAVAHOME $JAVADIR $JAVADIR/DayDriver.jar $JNDI.jar $JPortal.jar
com.att.research.jdbc.daytona – a Java JDBC Driver Package for Daytona

DESCRIPTION
com.att.research.jdbc.daytona is a Java JDBC Driver Package for Daytona, the AT&T Database Management System by R. L. Greer, AT&T Labs - Research.
DayDriver.jar is the Java Archive (JAR) file of the Java JDBC Driver for Daytona.
JNDI.jar includes Java Naming and Directory Interface (JNDI) API utilities used by DataSource Objects to connect to a Daytona database via the Java JDBC Daytona Driver package.
JPortal.jar is a Java GUI application that provides adhoc query access to a Daytona database via the JDBC interface. The query may either be an SQL query or else a Cymbal program whose output is confined to that provided by a single Display call.

JDBC CLIENT-SERVER ARCHITECTURE
A Java or Java-enabled web application may use a Client-Server Architecture to access data stored in a Daytona database. In such a case, the Java application is known as the client requesting database services and the machine on which the Daytona database resides is known as the database server. The client and server (which typically are on separate machines) communicate with each other over a TCP/IP socket connection.

Client Side
----------
Java App
----------
Java JDBC
----------
JDBC Driver

Server Side
----------
PDQ Server
----------
Arthur
Daytona DBMS
----------
DB Files
() () () () () ()

A Client-Server Architecture for Daytona
The diagram above shows the big picture and how the the main pieces fit and work together — to provide remote access to Daytona and Daytona-managed database files from Java programs. It also helps show how information collected from a user through a Java program can be used with the JDBC to form a query, which is sent to a Daytona database on a remote machine, where it is executed and its results sent back to the Java program for use on the client side.

Starting at the lower-right of the diagram is a typical installation of Daytona and the files that it manages. This comprises a ‘stand-alone’ system that is used to compile and run Daytona queries locally on the machine.

In order to enable socket-based Client-Server capabilities, the (lightweight) Daytona Server PDQ must be running to listen for client requests on some unique port. PDQ, written in Daytona’s Cymbal language, has two tasks to perform: first, it checks with Arthur (a Daytona module, more information is given below) to authenticate the user and client making the request and, second, it passes validated
requests on to Daytona for processing.

Moving over to the left-side of the diagram, a Java program or application that wishes to work with Daytona needs a way to gain access to Daytona. The Java JDBC (“Java DataBase Connectivity”) Application Programming Interface (API) Package provides a set of Java classes and methods that can access nearly any SQL-like database — assuming a JDBC Driver has been provided for said database. In the present case, a Java JDBC Daytona Driver has been written which can communicate over a TCP/IP connection to a Daytona PDQ Server.

Using the diagram above, one can trace briefly how the information flows from a user form in the Java application to Daytona for processing, and then how the results flow back to the user.

User form input could be used to construct an SQL query. For example, it might be determined that the user wants the names and phone numbers of part suppliers in St. Paul, so the SQL query text constructed in the Java code might look like this string:

```
String UsrQuery =
   "SELECT SUPPLIER.Name, SUPPLIER.Telephone
   + "FROM SUPPLIER"
   + "WHERE SUPPLIER.City Matches 'St. Paul'
"
```

The Java application calls methods in the Java JDBC API (included in the normal installation of Java 1.4) to establish a Connection to a database by providing a url of the form:

```
jdbc:daytona://host_name:port_num/daytona
```

The protocol portion of the url, jdbc:daytona:, requests a JDBC Driver for Daytona; the remainder identifies the network address of the Daytona database server at a specific internet host and port, such as trntwn.acme.org:14029, or equivalently, 63.251.83.40:14029, along with any additional database information such as a Project name, etc. (A JDBC DataSource lets a name be used to locate the url via a naming service.)

In order for the JDBC Driver and the database server to communicate successfully, they must share common communications protocols. In this case, the Java JDBC Daytona Driver knows how to format, send, and receive messages with the Daytona Server PDQ listening for requests on the port at the named host. Before the Daytona Server PDQ accepts a connection request, it uses Arthur (see DS Man Arthur) to authenticates the user and service making the request.

The Java application creates a JDBC Statement that contains the desired SQL query text, in this case, the string UsrQuery. The JDBC Driver sends the query text in string UsrQuery by placing it in a message to PDQ; it also requests that the query text be compiled:

```
DS_COMPILE R_UsrQuery
$
SELECT SUPPLIER.Name, SUPPLIER.Telephone
FROM SUPPLIER
WHERE SUPPLIER.City Matches 'St. Paul'
$
#####
```

Assuming all goes well, PDQ responds with:

```
SUCCESS
#####
```

The Java application then calls the Java JDBC API to request that the query be executed; the JDBC Driver sends the message to PDQ to have the query run:

```
DS_EXECUTE R_UsrQuery
#####
```
Again, PDQ responds with:

SUCCESS

But the results are needed back on the client side! The Java application asks for them through the Java JDBC API, which, in turn, has the JDBC Driver forward the request to PDQ:

DS_CAT R_UsrQuery

PDQ responds by sending the requested Daytona query output back to the JDBC Driver over on the client side:

%msg01)delim)|
%msg02)
%msg03)Query File: R_UsrQuery.Q
%msg04)
%msg05)
%msg06)recls)R_UsrQuery
%msg07)flds)Supplier|Telephone
%msg08)types)STR(30)|STR(25)
%msg09)
Acme Shipping|612-149-5678
Bouzouki Receiving|612-943-7416
Julius Receiving|612-309-3492
Sunshine Warehouse|612-303-7074

The JDBC Driver reads the response from PDQ and processes the information (in the %msg lines) and data so that they are available to the Java program through the standard interfaces and calls of the Java JDBC API. The Java program is able to obtain information about the results by using JDBC ResultSetMetaData methods, e.g., to obtain the number of columns per row, column names, column data type, etc.; the Java program can loop through the query data results using JDBC ResultSet methods to get the result data row by row and column by column, until the end.

GETTING STARTED
To see what is needed to start using Java JDBC to access Daytona data, one can refer back to the earlier client-server architectural diagram and begin at the lower-right with an actual installation of Daytona on a given machine.

For discussion, assume Daytona is installed on machine named:

trntwn trntwn.acme.org 63.251.83.40

1. A Daytona Installation
The Daytona release includes a user test application called orders, which is used to test the implementation of the JDBC Daytona Driver. It is used here to illustrate the use of JDBC Daytona. (See Getting Started with Daytona - A Hands-On Tutorial, Section 4, "Looking At Some Data.") Once a user has been given access to Daytona and has DS_DIR set in their UNIX® shell environment, they can check their Daytona environment using the command:

DS Env

If the user wants to try out the Daytona orders application, a one-time script needs to be run to install the needed directories and files in a place of the user’s choice. Then, after executing the command:

. $HOME/setup.orders  # remember the initial dot!

to set the variables DS_APPS and DS_PATH, the user can move to the designated directory to run queries on the orders application. (The variables DS_APPS and DS_PATH may be changed via .
For example, suppose the designated directory is `d`. Daytona is ready for for work, compiling and running queries as usual:

```
trntwn:d> . $HOME/setup.orders  # remember the initial dot!
trntwn:d> cd d
trntwn:d> cat Q/stpaul.1.S

select Name as Supplier, Telephone as Phone
from SUPPLIER
where City = 'St. Paul'

trntwn:d> DS Compile Q/stpaul.1.S stpaul.1
R moved to stpaul.1
compilation completed successfully
trntwn:d> stpaul.1
Sizup():
Sizup starting at Mon ...
Sizup():
fyi: All relevant indices are up-to-date
Sizup():
Sizup finished at Mon ...
```

| %msg01) delim) |
| %msg02) |
| %msg03) Query File: Q/stpaul.1.S |
| %msg04) |
| %msg05) |
| %msg06) recls) STPAUL_1 |
| %msg07) fids) Supplier | Phone |
| %msg08) types) STR(30) | STR(25) |
| %msg09) |
| Acme Shipping | 612-149-5678 |
| Bouzouki Receiving | 612-943-7416 |
| Julius Receiving | 612-309-3492 |
| Sunshine Warehouse | 612-303-7074 |

2. PDQ Server for Dayton

PDQ is the process that serves as a gateway for Client-Server access to the ‘stand-alone’ Daytona environment; PDQ listens on a particular port for requests and passes any valid requests on to Daytona. PDQ relies on Arthur, a Daytona module, to handle authorization and security.


PDQ relies on Arthur to help validate requests according to various criteria, such as network address, type of service (e.g., JDBC), and user information. More information on Arthur is available with the command:

```
DS Man Arthur
```

To run JDBC tests via PDQ on the orders application (and assuming that install.orders, install.arthur, and install.pdq have already been run), the following Arthur commands will be useful:
To prime Arthur to handle JDBC requests for a user `jtest', commands like the following would be used (with the appropriate domain and password information given for the placeholders):

```
trntwn:d> $DS_ARTHURDIR/bin/arthur_pdq_init
trntwn:d> $DS_ARTHURDIR/bin/arthur_add_user \njjtest jdbc <net.domains> <password>
```

To check on Arthur, a shell `here-document' can be used to ask what Arthur has for `jdbc' agents:

```
trntwn:pdq> $DS_ARTHURDIR/bin/Arthur - <<!!
SHOW AGENT - - jdbc - - - ******
!!
```

### 2.b. PDQ’s Daytona Environment

When PDQ is started, it inherits the current Daytona environment in which it is started. All incoming requests are then handled within that Daytona environment. So if a Daytona PDQ Server is started once the Daytona environment has been set for the orders application, that PDQ Server will only handle queries associated with the orders application. Therefore, when starting PDQ, it is important to set up the desired Daytona environment.

Clients gain access to a particular Daytona environment by requesting a connection to a particular PDQ associated with that environment; a particular PDQ is identified by its host and the port number on which it is listening for requests. A single PDQ is able to handle multiple connections simultaneously. Each PDQ on a given machine must listen on its own unique port number.

Two things to note about systems software that handle TCP/IP socket communications. First, once a listener on a port is killed (such as PDQ), it may take a minute or two for time-outs to complete before the port is free for a new listener to start. Second, a port number assigned to localhost (known also as IP address 127.0.0.1), cannot also be used for a 'real' network connection. Here a 'real' network connection refers to one coming in over the network to the same machine using its network name or network IP address, such as: trntwn, trntwn.acme.org, or 63.251.83.40. In short, for initial testing, it may be handy to set up two PDQ servers, one to handle local JDBC Daytona request on the local machine via localhost on, say port 14028, and another to handle network requests via 63.251.83.40 on, say, port 14029.

### 2.c. Starting PDQ

Information on PDQ and its arguments is available using the command:

```
DS Man pdq
```

As PDQ goes about its work of handling requests, such as compiling queries, executing queries (perhaps with given sets of parameters), and returning query results, it creates subdirectories and files to organize its workspace. It may also be convenient to create a directory in which to run PDQ, perhaps with subdirectories for one PDQ handling 'local' (localhost) requests and another for another PDQ handling network request from other machines:

```
# directories in which to run PDQ
pdq
pdq/local # a PDQ here handles local requests on port x
pdq/net # a PDQ here handles network requests on port y
```

Remember to start each PDQ using a unique port number.

The steps to start PDQ can be reviewed by running the command:
These are the steps for starting PDQ to handle local request on port 14028:

```
# add DS_SERVERDIR to your shell environment
trntwn:d> DS_SERVERDIR=${DS_DIR}/getdserv DS_SERVERDIR
# move to pdq directory for handling local requests
trntwn:d> cd pdq/local
# add DS_SERVERDIR to shell PATH
trntwn:local> export PATH=${DS_SERVERDIR}/pdq/bin:$PATH
# start PDQ with log trace level 3 for localhost on port 14028
trntwn:local> nohup pdq -trace 3 -host 127.0.0.1 -port 14028 &
```

These are the steps for starting PDQ to handle network request on a different port, 14029:

```
# add DS_SERVERDIR to your shell environment
trntwn:d> DS_SERVERDIR=${DS_DIR}/getdserv DS_SERVERDIR
# move to pdq directory for handling local requests
trntwn:d> cd pdq/net
# add DS_SERVERDIR to shell PATH
trntwn:net> export PATH=${DS_SERVERDIR}/pdq/bin:$PATH
# start PDQ with log trace level 3 for net connects on port 14029
trntwn:net> nohup pdq -trace 3 -host 63.251.83.40 -port 14029 &
```

nohup is used so PDQ will continue running after the user who issued the command logs out.

To check that PDQ is running, use the command `ps.pdq`:

```
trntwn:d> $DS_SERVERDIR/pdq/bin/ps.pdq
```

```
usr_ds 1791 1 0 Sep 13 pts/013 0:00 \
    pdq -trace 3 -host 63.251.83.40 -port 14029
usr_ds 1779 1 0 Sep 13 pts/013 0:01 \
    pdq -trace 3 -host localhost -port 14028
```

3. Java JDBC Daytona Driver

One PDQ and Arthur are running on the Server Side, the Java JDBC Dayton Driver can be put to work on the Client Side.

The Java application and JDBC Dayton Driver can run on the same machine as Dayton — and talk to PDQ on a localhost port such as localhost:14028 or 127.0.0.1:14028. This can be convenient for testing and is shown first below. More often than not, the client software, consisting of some Java software and the Java JDBC Dayton Driver, are on another machine which has network connectivity to the machine with the Dayton database. Two common scenarios are discussed below.

3.a. JDBC Client on Same Machine as Dayton Driver

The Dayton installation includes the Java JDBC Dayton Driver, a test suite for the driver, and a directory with a symbolic link to a version Java that can be used to run the JDBC Driver tests. Once Arthur and PDQ have been started as outlined above, (in this case for 127.0.0.1:14028 or localhost:14028), the following commands can be used to run the JDBC Dayton Driver on the same machine to ‘ping’ PDQ to see if it is up and running okay:

```
trntwn:d> $DS_JAVAHOME/bin/java -version # show Java version
java version "1.4.2_03"
Java(TM) ...
Java HotSpot(TM) Client VM ...
trntwn:d> cd $DS_JAVADIR/jdbc/test
trntwn:test> $DS_JAVAHOME/bin/java -classpath $DS_JAVADIR:. DayPing -hpp localhost:14028/daytona
```
Ping PDQ @ localhost:14028/daytona
Pinng: PDQ on ttrntwn (127.0.0.1:14028) alive at ... 15:13:18 ...
Proj: daytona

$DS_JAVADIR/jdbc/test is a directory with the Java JDBC Daytona Driver tests in it. DS_JAVAHOME is the 'home' of the bin of the java executable, the Java virtual machine. DS_JAVADIR is the root for the Java JDBC Daytona Driver in Java package com.att.research.jdbc.daytona. DayPing is the name of the Java JDBC Daytona Driver test to be run, and -hpp 127.0.0.1:14028/daytona tells it to use host : port / project values of 127.0.0.1, 14015, and daytona. Although JDBC officially considers the portion of the url after the / slash to be optional, it is safest to include the Daytona project name or, if none, the text daytona.

To have JDBC send a test SQL query to PDQ for execution, run the JDBC Driver test DaySql5; it sends the SQL query text in:

```java
String UsrQuery = "SELECT SUPPLIER.Name, SUPPLIER.Telephone
+ "FROM SUPPLIER
+ "WHERE SUPPLIER.City Matches 'St. Paul'");
+ " AND SUPPLIER.Telephone Matches '"612'"
+ "ORDER BY SUPPLIER.Name"
```
to PDQ for processing by Daytona. Run the test with the command:

```bash
trntwn:test> $DS_JAVAHOME/bin/java -classpath $DS_JAVADIR:. \ DaySql5 -hpp localhost:14028/daytona
```

Once PDQ has the query compiled, executed, and the results returned to JDBC, Java is used to format the output:

```java
DS_Ping: PDQ on ttrntwn (127.0.0.1:14028) alive at ...
DS_Proj: daytona

Query File: MY_DaySql5.Q

Suppliers in St. Paul and Area Code 612

<table>
<thead>
<tr>
<th>Supplier</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acme Shipping</td>
<td>612-149-5678</td>
</tr>
<tr>
<td>Bouzouki Receiving</td>
<td>612-943-7416</td>
</tr>
<tr>
<td>Julius Receiving</td>
<td>612-309-3492</td>
</tr>
<tr>
<td>Sunshine Warehouse</td>
<td>612-303-7074</td>
</tr>
</tbody>
</table>
```

In this test case, JDBC ResultSetMetaData is being used to get column display size information, as handed back from PDQ and Daytona, to format the output in the Java code on the Client Side.

The Java source for DayPing and DaySql5 above are in the same directory as the Java class files: DS_JAVADIR/jdbc/test; you are free to copy them to your own directory and tune the defaults, etc. They can then be compiled and run with Java JDBC using:

```bash
# cd to your own jdbc test directory
cp $DS_JAVADIR/jdbc/test/DayXxxx.java .
# compile (tuned) Java JDBC code
$DS_JAVAHOME/bin/javac -classpath ..:$DS_JAVADIR \ DayXxxx.java
# run (tuned) Java JDBC code -- send request to PDQ
```
$DS_JAVAHOME/bin/java \n    -classpath .:$DS_JAVADIR:$DS_JAVADIR/JNDI.jar \n   DayXxxx [-hpp localhost:14028/daytona]

Note that the \texttt{java/c/ -classpath $DS_JAVADIR:.} option is used to help locate the Java JDBC Daytona Driver. That may be satisfied either by the subdirectories beginning with \texttt{com} under \texttt{DS_JAVADIR} or by the Driver JAR (Java Archive) file \texttt{DayDriver.jar}, also in \texttt{DS_JAVADIR}. Note that the \texttt{JNDI.jar} file is included in the classpath, should any of its utilities be needed to help \texttt{DataSource} Objects establish connections.

\textbf{3.b. JDBC On-Line Javadoc Documentation}

On-line documentation for the Java JDBC Daytona Driver is available by pointing a browser at the file:

\texttt{file:$DS_JAVADIR/doc/index.html}

The links \texttt{Overview} and \texttt{Package Summary} serve as introduction. The \texttt{Package Summary} contains the complete code for the \texttt{DayPing} example above and describes the code.

\textbf{3.c. JDBC Daytona Driver on the Client}

One way to put the JDBC Daytona Driver on the Client is to copy over the \texttt{DS_JAVADIR} directory onto the client. The minimal requirements are \texttt{DS_JAVADIR} as a root directory with the contents of four subdirectories; the first two subdirectories are for the JDBC Daytona Driver and its extensions/utilities:

\texttt{DS_JAVADIR}
\texttt{DS_JAVADIR/com/att/research/jdbc/daytona}
\texttt{DS_JAVADIR/com/att/research/jdbc/daytona/ext}
\texttt{DS_JAVADIR/com/sun}
\texttt{DS_JAVADIR/com/sun/jndi}

The \texttt{DS_JAVADIR/com/sun} subdirectories contain utilities used with \texttt{DataSource} and \texttt{JNDI} naming and look-up services.

An alternative is to copy over the Java ARChive (JAR) file for the JDBC Daytona Driver to the same root directory, along with a second Java ARChive for the JNDI utilities:

\texttt{DS_JAVADIR}
\texttt{DS_JAVADIR/DayDriver.jar}
\texttt{DS_JAVADIR/JNDI.jar}

Note that Java requires the above directory or directory/subdirectory structure so that the top level directory can be used in the Java CLASSPATH in order to find the JAR or class files for the JDBC Daytona Driver.

If installing the JAR file(s), see also the section below: \texttt{JDBC Daytona Driver JAR File: DayDriver.jar}.

The JDBC Driver test suite can also be copied over to the Client from the same test directory as mentioned above:

\texttt{DS_JAVADIR/jdbc/test}

To run JDBC Daytona Driver tests on the Client, Arthur and PDQ need to be running on the Server Side: on the Daytona host machine. First be sure to start Arthur for JDBC request on the Daytona host machine if it is not doing so. Then start PDQ on on the Daytona host machine on a unique network port using a network IP address and port number, such as

\begin{verbatim}
  # start PDQ with log trace level 3 for net connects on port 14029
trntwn:d> cd pdq/net
  trntwn:net> nohup pdq -trace 3 -host 63.251.83.40 -port 14029 &
  trntwn:net> cd -
\end{verbatim}
Back on the Client Side, go to the subdirectory into which the JDBC Daytona Driver tests were copied, perhaps under directory `Client_DS_JAVADIR`:

```
mycl:d> cd Client_DS_JAVADIR/jdbc/test
```

Now the same PDQ ping test can be run as before, but this time the JDBC Daytona Driver needs to be told to connect to a PDQ listening for network connections on port 14029. Also check that the Java on the client is compatible with the one used to build the driver:

```
mycl:d> cd Client_DS_JAVADIR/jdbc/test
mycl:test> java -version
java version "1.4.2_04"
Java(TM) ...
Java HotSpot(TM) Client VM ...
mycl:test> $DS_JAVAHOME/bin/java \\
   -classpath .:$DS_JAVADIR/DayDriver.jar:$DS_JAVADIR/JNDI.jar \\
   DayPing -hpp trntwn.acme.org:14029/daytona
Ping PDQ @ trntwn (63.251.83.40:14029) alive at ... 15:18:47 ...
```

### 3.d. Run a Client-Side Java JDBC Daytona Query

Now use the same SQL query test for the Java JDBC Daytona Driver on the Client Side. Send the same SQL query text in the Java String

```
String UsrQuery =
   "SELECT SUPPLIER.Name, SUPPLIER.Telephone\n   + "FROM SUPPLIER\n   + "WHERE SUPPLIER.City Matches 'St. Paul'\n   + " AND SUPPLIER.Telephone Matches '612'\n   + "ORDER BY SUPPLIER.Name\n";
```

to PDQ on the Server Side for processing by Daytona. The Java code example from the Server Side in

```
$DS_JAVADIR/jdbc/test/DaySql5.java
$DS_JAVADIR/jdbc/test/DaySql5.class
```

should now be copied (if not previously done so) to the Client Side into a (‘parallel’) directory:

```
Client_DS_JAVADIR/jdbc/test
```

When the query is run on the Client Side, the following output is generated, thanks to help from PDQ on the Server Side:

```
mycl:d> cd Client_DS_JAVADIR/jdbc/test
mycl:test> $DS_JAVAHOME/bin/java \\
   -classpath .:$DS_JAVADIR/DayDriver.jar:$DS_JAVADIR/JNDI.jar \\
   DaySql5 -hpp trntwn.acme.org:14029/daytona
DS_Ping: PDQ on trntwn (63.251.83.40:14029) alive at ... 
DS_Proj: daytona
Query File: MY_DaySql5.Q

Suppliers in St. Paul and Area Code 612

<table>
<thead>
<tr>
<th>Supplier</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acme Shipping</td>
<td>612-149-5678</td>
</tr>
<tr>
<td>Bouzouki Receiving</td>
<td>612-943-7416</td>
</tr>
<tr>
<td>Julius Receiving</td>
<td>612-309-3492</td>
</tr>
<tr>
<td>Sunshine Warehouse</td>
<td>612-303-7074</td>
</tr>
</tbody>
</table>
Again, JDBC ResultSetMetaData is being used to get column display size information, as handed back from PDQ and Daytona, to format the output in the Java code for display on the Client Side.

The results obtained on the Client Side are the same as those when the same JDBC test case was run locally on the server side, with the exception of the information on the DS_Ping line.

3.e. JDBC Daytona Driver JAR File: DayDriver.jar

The second way to put the JDBC Daytona Driver on the Client is to copy over its JAR file: DayDriver.jar.

If DataSource Objects are being used to establish connections and the JNDI utilities are not already on the client, the JAR file JNDI.jar may also need to be copied over.

Client-side applications, such as Crystal Reports® v10 or ones using Apache Tomcat or Jakarta Struts, typically have a designated place in which to place JAR files for known drivers, along with special places to store information about the database url, host, and database server. In such cases, follow their installation instructions.

The JDBC url for Daytona includes the host name and port num with which the client wishes to communicate. The complete JDBC Daytona url is of the form:

```
jdbc:daytona://host_name:port_num/daytona[,att=val...]
```

The JDBC Daytona Driver JAR file is included in the Daytona installation in:

```
DS_JAVADIR/DayDriver.jar
```

It can be copied to the client to the location desired.

If needed, the Driver JAR file can also be generated on the client side as long as the Driver subdirectories have been copied over to some top level directory (the doc documentation subdirectories are optional for the JAR file):

```
mycl:d> cd Client_DS_JAVADIR
mycl:Client_DS_JAVADIR> jar cf DayDriver.jar \com/att/research/jdbc/daytona/*ss \
com/att/research/jdbc/daytona/ext/*ss \doc/*[sm][lst] doc/com/att/research/jdbc/daytona
```

ENVIRONMENT

To see the settings of Daytona environmental variables, execute the following command at the UNIX® shell prompt:

```
DS Env
```

**DS_DIR**

Directory containing the Daytona executables.

**DS_JAVAHOME**

Directory containing the bin directory of java for the JDBC Daytona Driver package. To check its version number, use the command:

```
$DS_JAVAHOME/bin/java -version
```

**DS_JAVADIR**

Directory containing the JDBC Daytona Driver package, including its JAR file: DayDriver.jar.

or the DS_JAVADIR directory may be copied to a client for use as a Java JDBC Daytona Driver, as long as the client’s version of Java is compatible with the version of Java in
DS_JAVAHOME.

DS_JAVADIR/DayDriver.jar
The JAR file for the JDBC Daytona Driver package. May be copied to a client

DS_JAVADIR/JNDI.jar
A JAR file for Java Naming and Directory Interface (JNDI) API utilities used by DataSource Objects to connect to a Daytona database via the Java JDBC Daytona Driver package. If JNDI utilities are not already on a client and DataSource Objects are being used to establish connections, it may be copied to a client.

DS_JAVADIR/JPortal.jar
The JAR file for the JPortal GUI application for Daytona database access. This file may be copied to a client machine all by itself and will run as a standalone app. Refer to the JPortal man page for more information.

DS_JAVADIR/doc
Directory containing the javadoc on-line documentation for the JDBC Dayton Driver and JDBC Dayton Driver Extension packages. To view, point your browser at:

    file: $DS_JAVADIR/doc/index.html

The Overview and Package provide an overview of the JAVA JDBC Dayton Driver with links to full documentation. The Package page contains a complete code example.

DS_JAVADIR/jdbc/test
Directory containing JDBC Dayton Driver test programs. Once PDQ is running on the appropriate port and accessing the sample demo database of PARTS, SUPPLIERS, and ORDERS, the tests may be run using the commands:

    // client-side directory with jdbc tests
cd $DS_JAVADIR/jdbc/test
    // compile java code
$DS_JAVAHOME/bin/javac \ 
    -classpath ..:$DS_JAVADIR/DayDriver.jar DayXxxx.java
    // run java virtual machine ... without JNDI services
$DS_JAVAHOME/bin/java \ 
    -classpath ..:$DS_JAVADIR/DayDriver.jar DayXxxx
    // run java virtual machine ... with JNDI naming services
$DS_JAVAHOME/bin/java \ 
    -classpath ..:$DS_JAVADIR/DayDriver.jar:$DS_JAVADIR/JNDI.jar DayXxxx

DS_SERVERDIR
Directory for the Daytona Server. Contains a subdirectory with the PDQ executable and utilities:

    DS_SERVERDIR/pdq/bin

DS_ARTHURDIR
Directory for the Dayton authorization/security module. Contains a subdirectory of Arthur modules. Arthur is used as a coprocess by PDQ to authenticate requests; Arthur may be used standalone by administrators.

    DS_ARTHURDIR/bin

AUTHOR
J. J. Snyder, Consultant at AT&T Labs - Research

COPYRIGHT
Copyright © 2004 by:
AT&T Labs - Research, U.S.A.
SEE ALSO

DS(1),    DS Env(1),
DS Man pdq(8),    DS Man Arthur(8),
DS Man jportal(8).

On-line Javadoc Documentation via Browser
 file: $DS_JAVADIR/doc/index.html
    --> Overview Package Summary

Getting Started with Daytona - A Hands-On Tutorial

All About Daytona

JDBC API Tutorial and Reference, Third Edition

Microsoft ODBC 3.0
   Software Development Kit and Programmer’s Reference,
NAME
CrystalReports – Configuring it to use Daytona via PDQ + JDBC

DESCRIPTION
Getting CrystalReports to make queries against a Daytona database involves many preconditions, most of which are beyond the scope of this man page. References to additional documentation are included below. This documentation assumes that the installer knows how to create folders (aka directories), copy files, and edit plain text files under the MS Windows system. The administrator for Daytona will need to set the user up for remote access and provide the drivers.

Outline
Here is a short outline of the steps need to set up CrystalReports:
install CrystalReports;
make folders;
copy driver files;
edit the configuration file;
run CrystalReports.

Basic Installation
No custom settings were needed during the installation process. It is not yet known which custom settings might interfere with this set up.

Folders
The installation of CrystalReports creates and populates many folders. Two main trees of their files exist but we need to concern ourselves with only one of these, \Program Files\Common Files\Business Objects, (all three folder names contain a space character). In this folder is a folder for the release or version, 3.0 as of this writing, and in that folder is the folder java which has a configuration file that will need to be modified.

The JDBC drivers for Daytona need to have a place on the MS Windows system, and by analogy that folder can be named \Program Files\Common Files\Daytona and you will need to create this folder.

Parameters and Driver Files
The administrator of Daytona on the server will need to enable remote access via pdq and setup a user with the ‘client class of jdbc’. The administrator will need to provide you the host name, port number, user name and password to use as well as the two driver files, DayDriver.jar and JNDI.jar (from the Daytona distribution).

The driver files will need to be copied onto the MS Windows system, into the folder that you created for them.

Edit the Configuration File
The CrystalReports installation provides a configuration file, in XML that is used by the Java subsystem, CRConfig.xml in the java folder (mentioned above). The XML element, <Classpath>, will need to be extended. Insert the full path names for two driver files you copied onto the system, with a semicolon after each, just in front of the reference ‘‘$/CLASSPATH’’. (You can use either forward slashes, ‘’/’’, or back slashes, ‘’\’’, or a mix of both in those path names.)

Using CrystalReports
When running CrystalReports, to establish a connection to the server, select Data Explorer, then Create New Connection, and then JDBC. You should then be presented with a form. The Connection URL will need to start with jdbc:daytona:// followed immediately by the host name of the server, then a
colon, then the port number, then finally /daytona to complete it. Enter 
att.com.research.jdbc.daytona.Driver into the field Database Classname. Then click Next and a new 
form will prompt for the user name and password to use. After entering the values you got from the 
Daytona administrator, click Finish and you are set up and connected.

FILES

$DS_DIR/JAVA/DayDriver.jar, 
$DS_DIR/JAVA/JNDI.jar, 
These are the two driver files that the Daytona administrator will give you.

C:\Program Files\Common Files\Business Objects\3.0\java\CRConfig.xml, 
This is the configuration file that needs to modified. (The path name may change with newer 
versions of CrystalReports.)

C:\Program Files\Common Files\Daytona
This is a new folder to put the driver files in.

SEE ALSO

pq(8), jdbc(8)

All About Daytona
NAME
Py2Daytona – A Python module to communicate with pdq as a pydbapi client

SYNOPSIS
import Py2Daytona
a = Py2Daytona.connect(host=hostname, user=id, passwd=secret, port=portnum)
b = a.cursor()
b.execute(sql_query)
r = b.fetchone()

DESCRIPTION
The module Py2Daytona is written in Python to facilitate access to the data and services provided by the Daytona data management system. Daytona goes beyond the traditional features of a database management system, but not all of those features are accessible through this interface.

Py2Daytona has been written to conform as close as reasonable to the “Python Database API Specification v2.0” (http://www.python.org/dev/peps/pep-0249) with numerous extensions and a few limitations. The mandated support for DATE, TIME, and DATETIME is just a pass-through from the datetime module supplied by Python; if those types are used, then Python must be version 2.3 or later. There is no reasonable mapping from the type Binary to anything in Daytona and invoking that constructor will raise an exception.

The specification seeks to expose the functionality of the DBMS in the context of the Python language. Thus, this implementation does not attempt to make the support by Daytona of SQL any more or any less conformant to the ANSI standard than it is. Here are some of the highlights of the ways that Daytona SQL is different (that a user of this module might need to be careful of):

* While Daytona has missing values, it does not have NULL values. The difference is explained in Chapter 4 of All About Daytona and in Daytona Basics. Suffice it to say here that no Daytona SQL query will return NULL as part of an answer record and that a query’s reference to a FIELD that happens to have a missing value in a given record will cause that record to be skipped over for the purpose of generating answers. Note that “select *” refers to all FIELDS.

* It may omit duplicate records (every unique record will occur at least once).

* Names of tables, names of columns, and literals that might confuse SQL should be enclosed in carets, also known as “hats”, “” (such as “from 'ORDER'” to reference the table whose name is also an SQL keyword).

* Since SQL is translated into Cymbal, the types of values must ultimately be those that Cymbal uses. Casts may reference Cymbal type names by putting a colon both before and after those names inside the parentheses, like “:(DATE:)”. (See also the discussion about casting parameters below under Query Class.)

* Functions and predicates from Cymbal can be invoked but those names are case sensitive. (Function names must be lower case and predicate names must be “Uplows”.)

* Full regular expression matching is provided with “Matches”, which can be used wherever the standard SQL expression “LIKE” would be allowed.

* Only those portions of SQL that deal with queries (DQL), inserts, deletes, and updates (DML) are accepted, and not meta data queries (DDL) nor access control queries (DCL) (although a portion of the later is accessible through separate methods).

Getting Python programs to make queries against a Daytona database involves many preconditions, some of which are beyond the scope of this man page. References to additional documentation are included below. To access a Daytona database, this implementation depends on the services of pdq, “a server for Polyclient Daytona Queries”.

Daytona Last change: 27 March 2009 1
**Notation**

The names of classes, attributes, functions, and formal parameters are shown here in italics with the exact spelling used in the implementation. This information permits the reader to invoke most functions with either positional or keyword arguments. Keyword arguments can be useful with functions that accept more than one optional argument, such as in a call to the `.callproc` method of the `Cursor` class that needs to specify a value for `user` but not for `keys`.

**Driver Files and Initial Parameters**

The administrator of Daytona on the server will need to enable access to Daytona via pdq (and Arthur) and set up an account with the “client class” of “pydbapi”. The administrator will need to provide the host name, port number, user name, and password for use in establishing a connection. (See Connection Class below.) The network administrator may also need to make changes to routers and firewalls to enable access.

If the user is running Python somewhere other than on the server machine, the directory `$DS_DIR/PYTHON` (and everything under it) will need to be copied from the server machine. Either way, Python needs to be told where to find the Py2Daytona module. The environment variable `SPYTHONPATH` can be set to include that directory before running Python, or the module `sys` will need to be imported and the attribute `sys.path` changed to include it, in order for the import to succeed.

**Warnings and Extensions**

The writers of the specification sought to provide for the maximum in interoperability, and as such, documented not only some extensions that would be commonly implemented, but also called for the issuance of warnings when these, or any other extensions, are used. For the most part, this implementation follows that specification, with two major exceptions. While the standard tools in Python provide for filtering out uninteresting warnings, this module also has a global attribute that controls whether to even issue those warnings, and that can be changed from its default after the module has been imported. (See `.do_warnings` below.) Second, this module can not readily detect when the user changes module attributes (see “Module attributes, functions, and classes” below), but will occasionally check the setting of `.paramstyle`. (All other standard module attributes are for reading only and changes to them are never noticed.)

All attribute or method names for extensions beyond those spelled out in the specification (that is, created for Py2Daytona) contain at least one embedded underscore as a distinguishing characteristic, while those names in the specification either have none at all or have only initial and trailing pairs, such as `__iter__()`. The only exceptions to this rule are the method `Cursor.rewind()`, which is just an alias for another routine from the specification, the module attribute `__version__`, the attribute `.paramstyle` on any object other than the module, and the added exception classes.

**Module attributes, functions, and classes**

This module exposes the following attributes (“module globals”) that are required by the published specifications:

- `.apilevel` - currently set to the string “2.0” (the version of the specification followed).
- `.threadsafety` - currently set to 0 to indicate that it is not safe for more than one thread to execute in this module at a time.
- `.paramstyle` - currently set to the string “named”, expresses the initial default for how parameters are marked in SQL queries. This attribute is only used as the default setting for a state in Connection object. Other valid settings are “qmark” (for queries like those used with JDBC) and “numeric” (for parameters marked with an ordinal). (See the discussions of `.paramstyle` in the section on the Cursor class, below.)
Also according to specification, there are the ‘‘Type’’ attributes used in the tuples provided by `Cursor.description` (see below): NARY, and ROWID (although the last two are never used by this interface). Each of the Type attributes has a value that is simply the string that matches the attribute name.

There are also the following nonstandard attributes:

- `__version__` is a string of three numbers separated by periods, referred to as ‘‘major’’, ‘‘minor’’, and ‘‘edit’’ numbers, where major == 1 for ‘‘beta’’ and major > 1 for released versions.

- `default_datetime_type` sets the initial value of the `datetime_type` attribute of `Cursor` objects. As of this writing, it is initially set to ‘‘string’’. Each new instance of the class `Cursor` copies this global attribute at its creation and uses its own copy to determine its behavior. (See `Cursor.datetime_type`.)

- `do_warnings` is a flag to indicate whether to issue warnings on use of nonstandard features. As of this writing, it is initially set to True. Each object from this module copies this global value at its creation and uses its own copy to determine whether to issue a warning.

There are eight required constructor functions exposed:

- `connect(...)` is a synonym for `Connection()`, the constructor of the class `Connection`, and takes the same exact arguments (see below).

- `Date(year, month, day)` is a synonym for `datetime.date()` and the arguments are just passed through.

- `Time(hour, minute, second)` is a synonym for `datetime.time()` and the arguments are just passed through.

- `Timestamp(year, month, day, hour, minute, second)` is a synonym for `datetime.datetime()` and the arguments are just passed through.

- `DateFromTicks(ticks)` is a synonym for `datetime.date.fromtimestamp()` and the argument is just passed through.

- `TimeFromTicks(ticks)` uses `datetime.datetime.fromtimestamp()` but returns the time object only.

- `TimestampFromTicks(ticks)` is a synonym for `datetime.datetime.fromtimestamp()` and the argument is just passed through.

- `Binary(string)` raises the exception `NotSupportedError`.

There are also three extension type management functions exposed:

- `date_from(val, dtyp)` returns a `date`, `time`, or `datetime` object as appropriate given a string, `val`, containing a Daytonia formatted date and/or time and a string containing the full Daytonia type, `dtyp`.

- `py_type(dtyp)` returns a string representing the Python type (as would be returned by the function `type`) that would normally be used for representing a value given the Daytonia type `dtyp`.

- `api_type(dtyp)` returns the ‘‘Type’’ attribute that corresponds to the string representation of the Daytonia type given as `dtyp`.

Other than the numerous exception classes, the module defines four classes to do the work and are described here. Two of these, `Connection` and `Cursor`, are covered by the specification, `Server` objects are used to handle the persistent aspects of a user account, and `Query`, which defines objects for internal bookkeeping. Each instance of `Server` is uniquely defined by the triplet of hostname, port number, and user name. Given the nature of pdq, connections that share these properties will affect each other. Each instance of `Server` may have one or more instances of `Connection`, which in turn can have zero or more instances of `Cursor`. Each instance of `Cursor` must have exactly one associated instance of `Connection`, and each instance of `Connection` must have only one associated instance of `Server`. Reflecting the nature of pdq, instances of both `Server` and `Connection` may have zero or more instances of `Query` and every instance of `Query` is known by only one instance of either `Server` or `Connection`. Each `Query` object maps to an executable file on the server and, when that `Query` object is associated with a `Connection` object, it reflects that the file will be deleted when the connection closes.
This module generally follows a “lazy implementation” policy. This means that the constructor method for `Connection` actually opens a connection only because it has to validate its parameters. (It needs to complain immediately if they are faulty.) Results are not transferred back from the server until they are called for. Actions are performed only when required, not as soon as they might be attempted. No work is done on the speculation that it might be needed.

**Exception Classes**

This module defines the ten exception classes called for by the specification as well as four more. Each exception class here is ultimately a subclass of `StandardError` with no change to the supplied methods or attributes (except for the documentation string that hints at what might have caused it to be raised). The required exception classes form a hierarchy. Subclassed under the exception `Error` are:

- `InterfaceError` - for errors that are specific to the interface.
- `DatabaseError` - for errors that are related to the operations of the database.
- `DataError` - for bad data detected by the database system.
- `OperationalError` - for error conditions outside of the control of the programmer, such as memory exhaustion.
- `IntegrityError` - for problems that manifest themselves as a loss of integrity for the database.
- `InternalError` - for errors associated with artifacts of access, such as a cursor that is no longer valid.
- `ProgrammingError` - for errors in programming, such as a syntax error in a query.
- `NotSupportedError` - to indicate that some method invocation attempts to do something that the database system, in this case Daytona, does not support.

This interface goes beyond the specification with four additional exceptions, all subclasses of `InterfaceError`, with the following names:

- `BadParameter` - raised when the value provided as an argument to a method is of the wrong type or is out of bounds.
- `MissingParameter` - raised when a method (the creation of a `Connection`) is called without a required named parameter.
- `OperationDisallowed` - raised in response to an operation that is no longer or not yet allowed, such as issuing a query after the cursor or connection has been closed.
- `Inconsistency` - raised if and when the internal data structures used to record the state of the interface are determined to not be sane. (Please report any occasion when this appears.)

**Connection Class**

The constructor function for the class `Connection` takes only keyword arguments with string or integer values, requires exactly four of them (as noted in the `__doc__` attribute), and accepts one optional keyword argument:

- `host` - the sufficiently qualified host name as a string (required)
- `user` - the user name known to `pdq` (required)
- `port` - the number of the port on `host` (required)
- `passwd` - the password to use with `user` (required)
- `paramstyle` - a value to override the default that would be inherited from the global attribute of the same name

The public interface has three methods and three attributes. The specification only calls for these three methods:
System Administration Commands and Daemons

```
.close()  - explicitly close the connection (and all associated Cursor objects).
.commit() - does nothing. Daytona does not provide support for transactions that span across queries; a query is committed or aborted before control returns to this driver.
.cursor() - return an associated Cursor object that provides most of the useful interaction with the database. This method accepts the optional keyword argument paramstyle, which would override inheriting a value from this Connection object.

The following four attributes are extensions beyond the specification:

.status_string - a read-only attribute that contains either an empty string (if the last request to pdq through this connection was deemed a success) or the full text of the response.
.do_warnings - either True or False, to indicate whether use of features beyond those required by the specification should cause a warning to be issued. This flag is copied from the module attribute .do_warnings when a Connection object is created, and can be changed anytime thereafter.
.server_ - a read-only attribute that refers to the Server instance that is associated with this connection.
.paramstyle - a read-only attribute that was set at object creation and will be used as the default inherited by Cursor objects as they are created, and may be the setting chosen for use by associated Server objects.

In addition to the public interface, this class provides many methods for use by the objects of other classes in this module. Everything pertaining to the protocol for communicating with pdq is contained here, including formatting of messages, parsing of responses, and remembering the state of the connection. It actually transfers the queries and retrieves the responses. This class does delegate some actions and state information to the Server class, which only deals with the aspects (and statefulness) of the user account that are persistent across connections.

Cursor Class

The Cursor class is the publicly seen "work horse" of this module. The public methods that it provides coordinate the private services from the Connection and Query objects to effect the work the user requests. Consequently, this description of the Cursor class is long enough to need its own table of contents:

- General properties
- Execution of anonymous SQL
- Execution of anonymous Cymbal
- Execution of named procedures
- Fetching results
- Results metadata
- Database metadata
- Behavior controls
- Miscellaneous

All the methods and attributes of Cursor objects pertain to executing a query and retrieving the results from the execution of a query. The specification only spells out the treatment of SQL, while this interface allows arbitrary Cymbal code to be passed as well. The specification views almost all queries as anonymous, with just one method for executing a persistent named query. (See the Server class below for other operations on persistent named queries.) In contrast, in Daytona all queries are named executables, so this module manages the namespace of queries, generates unique names for each seemingly anonymous query, and distinguishes between persistent queries and those that are to be deleted when the connection closes.
```
There are several general extensions to the behavior of Cursor objects as pertains to the treatment of some of the common arguments given to class methods:

+ While the specification expects that each query statement will be a single string, this implementation allows a query statement to be a list of strings that can be joined with new lines between them.

+ While the specification expects that the SQL will be a single, albeit complex, “select”, Daytona does not really have “cursors”, and a query may in fact contain multiple SQL statements and each may produce a results set. (See .nextset() below.)

+ According to the specification, the parameters for each query are supposed to be provided as a dictionary (for queries with .paramstyle == “named”) or as a sequence (for other settings of .paramstyle). However, this implementation allows any query to be given parameters as a dictionary, a sequence, or a single string. Whenever a set of parameters is presented as a single string, it is passed with no further processing to the query executable on the server machine (via pdq and the shell), and the query executable may fail if that string does not get parsed correctly by (both the shell and) the query.

+ The nature of Daytona queries requires that all query parameters be serializable as a text string. The interface (via the class Query) automatically handles all necessary quoting, unless the parameters are presented as a single string.

+ By default, this implementation supports named parameters in an SQL query with those names used to index into the supplied dictionary of values ( .paramstyle == “named”) and saves information about that mapping in the executable queries. However, this kind of mapping is not a common Daytona practice and named queries from other Daytona interfaces may fail to provide this information. To provide a consistent workaround for this case, all the methods that execute queries accept an optional list named “keys”. On those occasions when the interface cannot determine the names (to lookup in the dictionary) and the order to present those values, from examining the query on the server, the list keys will be used. If the interface needed keys and it was not provided, an exception will be raised.

+ Persistent, “named”, queries are, by the design of pdq, associated with a user account. The method to execute named procedures also allows an optional argument, user, to select which user catalog of queries to look in.

+ Anonymous queries only go through the process to transfer them to the server machine and convert them into an executable file at the first time in a session that they are presented to this interface. This process, including the handling of .paramstyle, is controlled by the Query class (see below).

The methods for executing anonymous SQL are:

execute(sql, parm, keys) - the simple execution of an anonymous SQL query. The arguments parm and keys are optional. If any results sets are produced, the first of these will become available to the various fetch methods (see below) and any others are held in reserve.

executemany(sql, parms, keys) - similar to the execute() method, but with a few differences. The first is that the query, sql, will be executed zero or more times. Second, if the optional argument parms is provided and not None, it must be a sequence of values that can be passed as parm to execute() because this method is implemented as such a loop and the query will be executed once for each value in sequence. Lastly, the query is expected to not produce any result sets, although no exception is raised when result sets are produced. However, results sets from all but the last set of parameters (i.e., the last execution of the query) will be silently discarded. This method will not return until all the executions have finished. (This method is intended for use with SQL updates and inserts.)
.execute_multi(sql, parms, keys) - similar to .executemany() with the explicit provision for each entry in the sequence parms to generate results sets (see .nextset()). Only the first of the multiple executions will be done and the others will be queued up to be performed only when .nextset() determines that another execution is needed (remember this is a ‘’lazy implementation’’). Any other method that executes a query on this cursor will discard any unconsumed parameter sets in the queue (and those executions of the query will not happen).

The extension methods for executing anonymous queries in Cymbal are:

.execute_alt(prog, parm, keys) - like .execute() except that prog is a string or list of strings containing a program in Cymbal, and the program will not be analyzed by this interface to recognize parameter names; so it must either have no parameters, or a list of keys needs to be provided as keys, or parm needs to be a function instead of a dictionary. If the results of the query are displayed in the ‘‘DC data format’’, the interface will detect the metadata and present the results sets as it would for SQL queries. Otherwise, the data will be treated as a one-column table with each line of output being a single row. That one column will have the name ‘’’’ (the empty string) and be of type ‘’STRING’’.

.execute_alt_many(prog, parms, keys) - this is to .execute_alt() as .executemany() is to .execute(); a method to invoke an anonymous Cymbal query once for each set of parameters. All requested executions will have been performed before the method returns control to the user. If the query produces any results, all but the last of them will be discarded.

.execute_alt_multi(prog, parms, keys) - this is to .execute_alt() as .execute_multi() is to .execute(); a method to invoke an anonymous Cymbal query once for each set of parameters. Only the first is done immediately and the others are queued up to be executed as calls to .nextset() expose those results.

The method for calling named procedures (from the specification) is:

.callproc(name, parm, keys, user) - like .execute() or .execute_alt() except that the first argument is the name of the query instead of the text of a query. (See the Server class below for how to create named queries.) The argument user is optionally given to specify which account on the server has the query executable, with the obvious default being the account used for the connection. The optional argument keys is a list of keys for use with parm, in the order that the parameters need to be serialized (their lexical order in the query). The interface will only use keys if it can not divine the needed information from the named query itself.

Each method that performs an execution of a query discards the results of any previous query. Each result set is the product of a query acting on one set of parameters. The results in a set are presented as a table of rows, and the interface can show them a row at a time or as a list of rows. An exception is raised when results are requested before any query is executed or after a query that produced no results. An exception will also be raised when results contain a database data type that this driver cannot convert into a standard type. Queries may even return malformed results that cannot be automatically parsed (due to the possibility that some of those unsupported data types may contain the field separator as data). Each of the three fetch methods take an optional argument, named ‘’cooked’’, which can be used by an intrepid programmer to deal with unsupported data types. The default value for ‘’cooked’’ is True (or any other value greater than 0), False (or 0) separates the record into fields without trying to convert them (leaving them as strings), and any value less than 0 will cause each record to be returned as a single unsplit string (as are the results of a Cymbal query that does not use the ‘’DC data format’’).

The methods for accessing the results are:

.fetchone() - returns one row, as a list of values, if there is at least one row remaining, otherwise it returns None.

.fetchmany(size) - returns a list of rows, where the count of rows is either the requested number (size) or all that remains, which ever is less (the list will be empty if no results remain), and size defaults to the value set as .arraysize.
.fetchall() - returns a list of the remaining rows (may be empty).

,__iter__() - an extension that permits the cursor to be an iterator (and is generally called implicitly).

.next() - like .fetchone() except that it raises StopIteration when the results set is exhausted instead of returning None (and is generally called implicitly).

.next_dict() - like .next() except that instead of returning a row as a list of values, it returns a row as a dictionary, using the column names as keys.

.fetch_dict() - like .fetchone() except that instead of returning a row as a list of values, it returns a row as a dictionary, using the column names as keys.

.iter_dict() - an iterator that returns rows as dictionaries.

.nextset() - discard the current results set (if any) and will advance to the next results set (if any) with the return value being 1 if a new results set is in place and None otherwise. (This may involve re-executing the current query with the next set of parameters if any of them have been queued by .execute_multi().)

Besides the result data, some of the metadata is available as attributes:

.description - a list of items, one per column, where each item is itself a tuple of 7 items, where only the first two items of a tuple are not None, and they are the name of the column (as a string) and the type as one of the values of the module “Type” attributes.

.rowcount - the count of rows produced by the previous query and is -1 before the first query, when the query failed, or when the query was an update or insert.

.Daytona_types - a list of strings showing the types that Daytona reported for the columns.

.column_names - a list of strings showing the names that Daytona reported for the columns.

To discover metadata about the database, there is a single method:

.exec_metadata( category, ...) - performs one of several canned queries and sets up the information as a results set. Some values for category will also accept keyword modifiers to limit or guide the results. (The descriptions of the common roles of the modifiers follows the descriptions of the canned queries.)

The accepted values for category (of metadata query) are:

“app” or “schema” takes no modifiers and the results set has one column, containing the application names that the Daytona server pdq knows about.

“table” or “record” takes modifiers “app”, “rec”, and “type” (see below) and the results set contains 4 columns: “Table_Schem” (which is the application name), “Table_Name”, “Table_Type”, and “Remarks”.

“key” or “index” takes modifiers “app”, “rec”, “type”, and “uniq”. The number of columns and their content depends on the modifier “type”, which takes either of two values: “primary” (the default) or “index”. With the former, each row has 5 columns: “Table_Schem”, “Table_Name”, “Column_Name”, “Key_Seq” (the sequence number of the column in the key), “PK_Name” (the name of the key). With “index”, each row has 8 columns: “Table_Schem”, “Table_Name”, “Non_Unique”, “Index_Name”, “Type”, “Ordinal_Position”, “Column_Name”, and “Cardinality”. An index “Type” is either 1 to indicate that the index is clustered, 2 when the index is hashed, or 3 otherwise. If the modifier “uniq” is assigned the string “true” then the output will be limited to indices that are unique.

“field” or “column” takes modifiers “app”, “rec”, and “col”. The results set has one row for each column in the examined tables. The 10 columns in the results set are: “Table_Schem” (the application containing the table), “Table_Name”, “Column_Name”, “Position” (of the field in the table), “Type_Name” (the Daytona class), “SubType_Name” (the subclass specifier if
that specifier is non-numeric), “Supertype_Name”, “Col_Cnt” (any numeric subclass specifier), “Null_Code”, and “Default_Value”. If the column reported on is part of a complex data class, then the value shown in the “Supertype_Name” column will be the complex data class: “list”, “tuple”, or “set”. Such complex data classes are “unrolled” by Daytona when output.

“exec” or “queries” takes only the modifier “bin”. The results set contains one row for each query (executable) that the user is authorized to execute. There are only 2 columns: “Procedure_Name” and “Procedure_DIR”, where the latter is either the name of a pdq user (followed by a percent sign) or a dot to represent the current session directory. (The directory names returned are not the actual names to found on the server but rather are values useful for correctly invoking queries from this module.)

“exio” or “parameters” takes only the modifier “bin”. The results set contains one row for each input parameter or output column of each query (executable) that the user is authorized to execute. There are 6 columns: “Procedure_Name”, “Procedure_LOC”, “Column_Name”, “IO” (which is either “In” or “Out”), “Key” (which, for input parameters, is the name to be used as the key in a dictionary of parameters), and “Type” (the string representation of the Daytona type of the output column).

“fpp” or “functions” takes only the modifier “type”. The results set has two columns: “FPP_Type” and “FPP_Name”, with one record for each function.

The modifier keywords for .exec_metadata() are:

app - takes a regular expression or string with SQL wildcards and the results set will be limited to those records describing pieces within applications or schemas whose names match this qualifier.

rec - takes a regular expression or string with SQL wildcards and the results set will be limited to those records describing pieces within records or tables whose names match this qualifier.

col - takes a regular expression or string with SQL wildcards and the results set will be limited to those records describing fields or columns whose names match this qualifier.

uniq - either “false” which is the default, or “true” which will limit the results set to records describing keys that require unique values.

type - one of a limited set of values depending on the category of metadata being requested. When qualifying a “table” query, type is expected to match either: “TABLE”, “STDIN”, “STDOUT”, “PIPE”, “PIPEIN”, or “PIPEOUT”; when qualifying a “key” query, type is expected to be either: “index” or “primary”; when qualifying a “fpp” query, type is expected to be one of: “common”, “clock”, “date”, “date_clock”, “ip”, “misc”, “num”, “str”, “sys”, “time”, or “time_date”.

bin - takes a regular expression or string with SQL wildcards and the results set will be limited to those records describing executable queries whose names match this qualifier.

To control the behavior of queries, the specification includes the following methods and attribute:

.arraysize - the default size to use in .fetchmany() and is initialized to 1 when the cursor is created.

.setinputsizes(sizes) - a no-op in this implementation.

.setoutputsize(size, column) - also a no-op in this implementation. (The second argument is optional).

This implementation also provides some miscellaneous functionality, both extensions from the specifications and non-standard additions. These include the following attributes and methods:

.scroll(delta, mode) - allows the cursor to reposition itself within a results set. The argument mode is optional and specifies that delta either is “relative” to the current position (the default) or is an “absolute” position, where a negative absolute position is interpreted as relative to the end of the results set.
rewind() - a convenient synonym for `.scroll(0, 'absolute')`.

datetime_type - either the string “string”, to indicate that (for this Cursor object) datetime values returned from Daytona should be left as strings, or the string “object”, to indicate that those values should be returned as `datetime` objects. Assigning an empty string to this attribute will cause it to be set with the value from the module attribute .default_datetime_type.

connection - a read-only attribute that contains the Connection object that this cursor belongs to.

status_string - a read-only attribute that contains either the empty string (if the last request made of the server was considered a success) or the full text of the error message.

do_warnings - like Connection.do_warnings (see above) in that it regulates the issuance of warnings for this object.

paramstyle - a read-only attribute that will control how place holders will be marked in SQL queries. (See Query class.) The value is set at creation of the Cursor object.

When the user is finished with a cursor, the method .close() can be used to end its usefulness. (The method .close() will also trigger garbage collection of this Cursor object and all its associated buffers.)

**Server Class**

The Server class exists as a repository of information about queries that persist from connection to connection and other properties of an account with pdq, such as:

* The hostname of the server where pdq is running.
* The port number on that server where pdq is listening for connections.
* The user name to give to pdq.
* The secret password used to authenticate the user to pdq.

All Connection objects that are created with the same values for hostname, port number, and user name also share the same Server object.

The methods that create and operate on persistent queries are part of this class. The names of queries are required (by this interface) to begin with either an alphabetic or numeric character, and may only contain letters, digits, underscores ("_"), hyphens ("-"), and periods ("."."). The methods for defining named, persistent procedures are:

`def_proc(name, sql)` - like the first half of Cursor.execute() in that it modifies the supplied SQL statement to handle parameter passing (see the class Query below) and then converts it into an executable file on the server. However, it differs in that the file is stored with the given name in the catalog of persistent queries for the account that defines this Server instance.

`def_proc_alt(name, prog)` - like `def_proc()` except that prog is the text of a Cymbal program that will be passed unchanged to pdq for conversion into a persistent named query The initial portion of prog must process arguments and issue a usage message when the query is not given the correct number of arguments. That initial portion of the program can be produced by the utility function cym_usage Persistent queries that do not have a properly issued usage message will not be callable through the methods of this driver. (This driver automatically handles this requirement for SQL queries.)

This class (Server) exposes the vital utility method:

`cym_usage(name, count, src)` returns a (multi-line) string to be used as the initial part of a Cymbal program (essential for persistent queries and optional for anonymous queries). The resultant string contains the code that defines a usage message, exits with complaint if the the argument count is not right, and reads in any arguments to variables named with the prefix, ".cl_arg_", followed by the argument number (starting at 1). The name argument must be the name of the query, that is to say, the name that will be given to .def_proc_alt to become the name of the
The `count` argument provides the exact number of command line arguments that the query will expect. The optional `src` argument provides a block of text containing one line per argument of the form (where the argument number and variable name would be altered as appropriate):

argument 1 comes from `variable` formatted `'%s'`

There is an alternative invocation of that method:

`.cym_usage(name, keys)` where keys is an ordered list of the names (keys) that are to be used for lookup in the dictionary. This invocation uses the size of the list as the count and the contents of the list to generate the string in place of `src`.

Persistent, named queries can be shared with other users. The following two methods are to control access to those queries and both will return True or False to reflect if they succeeded (and set `.status_string` as well):

`.grant_query(name, other_user)` - grant permission to the other user (where `other_user` is the user ID of the other user) to use the persistent, named query executable.

`.revoke_query(name, other_user)` - undo the effect of a `.grant_query()`.

Six attributes of `Server` objects are exposed:

`.status_string` is a read-only attribute that contains either the empty string (if the last request made of `pdq` was considered a success) or the full text of the response.

`.do_warnings` is like `Connection.do_warnings` (see above).

`.user_name` returns, as a string, the user name associated with this `Server` object.

`.host_name` returns, as a string, the host name associated with this `Server` object.

`.port_num` returns, as an integer, the port number associated with this `Server` object.

`.paramstyle` is a read-write attribute, used in processing the SQL statement in `.def_proc()`, which defaults to a value gotten from one of the `Connection` objects associated with this `Server` object.

### Query Class

The class `Query` provides no methods nor attributes that the user would be expected to access directly. It provides to the other classes of this interface services such as the following:

* Transforming SQL queries by adding boilerplate to get parameters from the command line and replacing parameter references with idiomatic Cymbal expressions, according to the setting of the attribute `.paramstyle` of the `Cursor` object or `Server` object that got the query.

* Producing a properly quoted argument string for submission to this query’s executable given either a dictionary of parameter values or a sequence of objects in needed order.

* Issuance of requests through an associated connection to cause the query to be compiled into an executable.

The interface supports three different styles of marking, in an SQL query, placeholders for the values of the query parameters:

```
``qmark'' uses a simple question mark, `''?', to mark the position in the SQL query. The parameters need to be organized into an ordered sequence such that the first parameter will go into the query at the place of the question mark that occurs lexically nearest the start of the query text. Parameter sets provided as a dictionary will have to have a list of keys (the `keys` argument in an execute method) to ‘linearize’ the dictionary. With this style, if the same value needs to be reused in a query it simply must be given again in the corresponding place in the sequence.

``numeric'' uses a colon, `''::'`, followed immediately by a (small) integer. The integer is used as an index into the sequence of parameters. As with the ‘qmark’ style, any dictionary of
parameters will need to be 'linearized'. A parameter used more than once in a query can be given the same number and only needs to occur once in the parameter sequence, corresponding to its number assignment.

‘named’ also uses a colon, but this time followed by alphanumeric characters. The alphanumeric string is used as the key to index into any supplied dictionary. If a parameter set is supplied as a sequence, the elements will be assigned to the query placeholders in lexical order. (The first value will be used for the first placeholder, and so on.)

This driver transforms SQL queries (to deal with parameters) using incomplete information and may need help from the user to get it all right. The transformation is done independent of the actual parameter values to be used, and, as such, the driver can neither infer nor discover the data types that those parameters should assume when the query is executed. Daytona provides a special indirect data type to use in a cast, ‘\`(::FIELD:)\'’, which will convert the string (all parameters become strings at the interface) into the type appropriate for the column of the table being referenced. This driver automatically inserts such a cast whenever a parameter appears in a query without an explicit cast. Sometimes, such as when a function is to be applied to a parameter, this helpful addition backfires. The user can override this default by including an explicit cast of the parameter in the query. This make for very unusual looking code when using the SQL cast function on an expression that includes a parameter for it would now need an explicit cast operator on the parameter, which might not obviate the need for the cast function.

Each Query object also retains the following information:
* The name of the query, as provided by the user or as generated by the interface
* The degree of persistence: either ‘sess’ for session limited, or ‘user’ for a query that will persist after the connection is closed
* The original query string
* Any transformation of that original query to one suitable for consumption by pdq and Daytona
* The parameter names found in the query
* A copy of .status_string from the Connection object whenever a compilation is requested.

EXAMPLES

This trivial example is taken from an interactive session with Python running on a Linux system. The user input is shown in bold while the system responses are shown in italics. (This example contains real accounts and real passwords albeit they only work on localhost and with the sample database. Users should be careful with all account information, especially when those accounts access significant data over the network.)

```
$ python
Python 2.3.4 (#1, Feb 2 2005, 11:44:49)
[GCC 3.4.3 20041212 (Red Hat 3.4.3-9.EL4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import Py2Daytona
>>> conn = Py2Daytona.connect(host='localhost', user='a_python_client', ...
... passwd='gvr9', port=14023)
>>> cur = conn.cursor()
>>> cur.date_time = 'object'
__main__:1: UserWarning: DB-API extension cursor.date_time used
>>> cur.execute("""select * from ^order^
... where date_placed > :mydate""", ['1984-10-15'] )
>>> print cur.fetchone()
[32L, 448, 107L, datetime.date(1986, 3, 14), datetime.date(1984, 12, 28), 4678, '0']
>>> print cur.rowcount
```
>>> Sql = []
>>> Sql.append("select Name, City, Telephone from SUPPLIER")
>>> Sql.append("where City Matches :city")
>>> lst = [{'city':'St. Paul'}, {'city':'Fair'}, {'city':'Trenton'}]
>>> cur.execute_multi(Sql, lst)
__main__:1: UserWarning: DB-API extension cursor.execute_multi() used
>>> # here we will use the cursor as an iterator,
>>> # and python will automatically invoke __iter__
>>> for row in cur:
...   print row
 [...] 
...   print row
 [...] 
['Acme Shipping', 'St. Paul', '612-149-5678']
['Bouzouki Receiving', 'St. Paul', '612-943-7416']
['Julius Receiving', 'St. Paul', '612-309-3492']
['Sunshine Warehouse', 'St. Paul', '612-303-7074']
>>> cur.nextset()
1
>>> print cur.rowcount
5
>>> rows = cur.fetchall()
>>> print rows[3]
['Rameses Import-Export', 'Fairlawn', '213-696-2343']
>>> print rows[4]
['Barbary AG', 'Fairfield', '201-923-1288']
>>> cur.nextset()
1
>>> print cur.rowcount
0
>>> cur.nextset()
>>> print cur.rowcount
-1
>>> 

This example did not show even one quarter of the feature set nor can it be a substitute for a textbook on the language Python but it should give the reader some sense of how this package could be used. Notice how the initial use of an extension results in a warning message, how the parameters for a query can be provided in various structures (not only in a dictionary, as the standard speci®es), and how the success of .nextset() (indicated by the “1” returned) says nothing about the size of the results set (which is empty for the query using `city':"Trenton'`).

FILES

$DS_DIR/PYTHON/Py2Daytona

This directory, along with everything under it, makes up the Py2Daytona module. If you have any difficulty locating it, consult your Daytona administrator.

SEE ALSO

pdq(8)
The daemon providing access to Daytona services.
http://docs.python.org/index.html
The best source for documentation about Python.

http://www.python.org/dev/peps/pep-0249
The official statement of the Python Database API Specification v2.0

http://docs.python.org/lib/module-datetime.html
Documentation for the datetime module.

All About Daytona
R. L. Greer, AT&T Labs - Research, 2009. Daytona Basics Larry Rose, Rick Greer, AT&T
Labs - Research, 2009.
NAME
JPortal — A JDBC-based Java App for Daytona Database Access

SYNOPSIS
java -jar JPortal.jar

DESCRIPTION
JPortal is a Java GUI application which provides adhoc SQL query access (through a JDBC interface) to a Daytona database supported by pdq (a Daytona DBMS server). Connection is made via an IP address and port number for the pdq server.

INVOCATION
On UNIX-based platforms, assuming that the bin directory for a JAVA installation is on the user’s PATH, JPortal is invoked by typing:

java -jar JPortal.jar

Since there are good javas and bad javas and more than one java can be installed on a machine, if this doesn’t work, please consult a Daytona developer to find out if you are using a good java or not: you will need to provide the UNIX path to the java being used as well as the result of `java -version'.

For Windows-based platforms, the application is started by simply double-clicking on the JPortal.jar file.

CONNECTION
The application starts by raising a Connection window. This window requires input for four fields:

- User name
- Password
- Server - hostname or IP address
- Port

The combination of Server and Port must represent a valid pdq instance. The User name and Password must also be recognized by this pdq instance.

Submit the connection request by pressing the Connect button.

INTERACTION
The attempt to connect will raise the main JPortal window. Note that, if the connection failed, there will be an additional window containing the associated error message.

The main JPortal window contains three frames and seven buttons. The frames, from top to bottom, are:

- Query - this frame contains the text of a user’s query. The query can be either an SQL query or a Cymbal program whose output is confined to that produced by a single Display call.
- Status - this frame provides the status of any user request.
- Results - this frame contains the results of a user’s query.

The buttons are:

- Connection - this button pops up the connection window to allow the user to establish a new database connection
- Schema - this button accesses a connected database and provides information about the database (tables, types, and defaults) in an expandable tree format
- Clear Query - this button clears the text in the Query frame
- Load Query - this button pops up a file browser to allow the user to load a locally stored user query
- Save Query - this button pops up a file browser to allow the user to save the contents of the Query frame as a file.

- Fetch Results - this button executes the query present in the Query frame and places the results in the Results frame. The Result frame can be modified by the user to sort by individual columns (by clicking on column headers) or to display columns in a different order (using drag and drop). Note that query executions are synchronous, so the user must wait for the results of a query execution before submitting any subsequent queries. Any errors in the execution of the query will pop up in a separate window.

- Save Results - this button pops up a file browser to allow the user to save the contents of the Results frame as a file.

**TERMINATION**

Currently, the application is terminated by closing the main **JPortal** window.

**ACKNOWLEDGEMENTS**

**JPortal** is based on publicly-available Java classes from Sun Microsystems (www.sun.com) and Java CodeGuru (codeguru.earthweb.com).

**SEE ALSO**

DS Man jdbc(8), DS Man pdq(8)
**NAME**

v10sort – sort and/or merge files

**SYNOPSIS**

```
v10sort [ -cmusMbdgfinrt ] [ -o output ] [ option ... ] [ file ... ]
```

**DESCRIPTION**

`v10sort` sorts lines of all the files together and writes the result on the standard output. The name `-` means the standard input. If no input files are named, the standard input is sorted.

The default sort key is an entire line. Default ordering is lexicographic by bytes in machine collating sequence. The ordering is affected globally by the following options, one or more of which may appear.

- `-v` Print out the `v10sort` version.
- `-?` Print out help message.
- `-b` Ignore leading white space (spaces and tabs) in field comparisons.
- `-d` 'Phone directory' order: only letters, digits and white space are significant in string comparisons.
- `-f` Fold lower case letters onto upper case.
- `-i` Ignore characters outside the ASCII range 040-0176 in string comparisons.
- `-n` An initial numeric string, consisting of optional white space, optional sign, and a nonempty string of digits with optional decimal point, is sorted by value. An empty string is treated as zero.
- `-g` Numeric, like `-n`, with e-style exponents allowed.
- `-M` Compare as month names. The first three characters after optional white space are folded to lower case and compared. Invalid fields compare low to `jan`.
- `-r` Reverse the sense of comparisons.
- `-tx` 'Tab character' separating fields is x. x may be the shell expressions '\t' (tab) or '\a' (bell) in addition to any single character, which may need to be escaped from the shell.
- `-k pos1,pos2` Restrict the sort key to a string beginning at `pos1` and ending at `pos2`. `Pos1` and `pos2` each have the form `m,n`, optionally followed by one or more of the flags `Mbdgfinr`; `m` counts fields from the beginning of the line and `n` counts characters from the beginning of the field. If any flags are present they override all the global ordering options for this key. If `n` is missing from `pos1`, it is taken to be 1; if missing from `pos2`, it is taken to be the end of the field. If `pos2` is missing, it is taken to be end of line.
- `-a pos1,pos2` `Pos1,pos2` designates an accumulating –n-style field, disjoint from the sort key. When the keys of two records compare equal, discard the second and replace the accumulating field in the first with the sum of that field in both. The receiving field must be large enough for the sum to fit without destroying any field separator. The sum has a ones digit always, a decimal point if either addend has one, a fraction part as long as that of either addend, leading zeros padded to full field width if either addend has a leading zero before the ones digit, and a + sign if either addend has a + sign and the sum is not negative.

Under option `-tx` fields are strings separated by x; otherwise fields are non-empty strings separated by white space. White space before a field is part of the field, except under option `-b`. A `b` flag may be attached independently to `pos1` and `pos2`.

When there are multiple sort keys, later keys are compared only after all earlier keys compare equal. Except under option `-s`, lines with all keys equal are ordered with all bytes significant.
Single-letter options may be combined into a single string, such as `-cnrt`.
The option combination `-di` and the combination of `-n` with any of `-diM` are improper.

These option arguments are also understood:

- `-c` Check that the single input file is sorted according to the ordering rules; give no output unless the file is out of sort. If `-u` is present, then check that there are no duplicate keys.
- `-m` Merge; the input files are already sorted.
- `-u` Unique. Keep only the first of two lines that compare equal on all keys. Implies `-s`.
- `-s` Stable sort. When all keys compare equal, preserve input order. Unaffected by `-r`.
- `-o` output
  Place output in a designated file instead of on the standard output. This file may be the same as one of the inputs. The option may appear among the file arguments, except after `--`.

`+pos1 -pos2`
Classical alternative to `-k`, with counting from 0 instead of 1, and `pos2` designating next-after-last instead of last character of the key. A missing character count in `pos2` means 0, which in turn excludes any `-t` tab character from the end of the key. Thus `+1 -1.3` means the same as `-k 2,2,3` and `+1r -3` means the same as `-k 2r,3`.

Options `-a`, `-g`, `-M`, and `-s` are not in the Posix standard, nor are the following tuning options.

- `-T` `tempdir`
  Put temporary files in `tempdir` rather than in `/usr/tmp`.
- `-y` `n`
  Use up to `n` kilobytes of internal store, or a huge number if `n`=0. v10sort’s temporary files have a size roughly equal to this number of kilobytes. The default is a mere 6MB.
- `-w` `n`
  Merge up to `n` files at a time.

There is but a single merge step where all the temporary files must be open at the same time. If there are not enough file descriptors available for this, the sort will abort. To enable a successful sort in that situation, either increase ulimit `-n`, increase the `-y` value, or else specify `-w` (to get a multi-stage merge).

EXAMPLES

v10sort `-u` `-klf` `-k1` `list`
Print in alphabetical order all the unique spellings in a list of words where capitalized words differ from uncapitalized.

v10sort `-t`: `-k3n` `/etc/passwd`
Print the password file (`passwd(5)`) sorted by userid (the third colon-separated field).

v10sort `-umM` `dates`
Print the first instance of each month in an already sorted file.

v10sort `-k1,1` `-a2,2` `items and costs`
Reduce a file of items (field 1) and costs (field 2) to a summary file of total cost per type of item.

FILES

`/usr/tmp/stm` or `/tmp/stm`

SEE ALSO

`comm(1)`, `join(1)`, `uniq(1)`, `look(1)`

DIAGNOSTICS

v10sort comments and exits with non-zero status for various trouble conditions and for disorder discovered under option `-c`. Overflow in a `-a` field warns and leaves the record uncombined.
BUGS

- When `–o` overwrites an input file, premature termination by interrupt, crash or file-system overflow can destroy data.
- Overflow in any but the first of multiple `–a` fields is fatal.
- Perhaps fields delimited by `–t` should grow to avoid `–a` overflows.

AUTHOR

v10sort was written by Doug McIlroy in the old days at Bell Labs for Tenth Edition Research UNIX.
Name

lpm - a C library for finding the Longest Matching Prefix

SYNOPSIS

Synthesizing the C Library lpm within Cymbal

DESCRIPTION

Glenn Fowler’s lpm C library functions can be used to construct a trie of IPV4 internet addresses of arbitrary cardinality. The constructed trie can subsequently be searched to find the node with the longest matching prefix.

A trie, or prefix tree, is an ordered tree data structure that is used to store an associative array where the keys are usually strings. All the descendants of a node have a common prefix of the string associated with that node. This property can be exploited to provide fast prefix matches, even for large tries.

For instance, the longest matching prefix to 134.88.157.34 might be found by the entry 134.88.0.0/10 in our trie of IP netmasks. This means that the first sixteen bits of the IPV4 prefix (8 bits each to represent 134 and 88) match. The /10 in this example denotes a mask whose ten lowest order bits are 0 whereas the rest are 1 (i.e. 255.255.252.0).

We utilize these functions within the lpm library (lpmmeth, lpmopen, lpmputmask, lpmmake, lpmatch, lpmclose) as declared in header file lpm.h, where:

- lpmmeth() specifies the method for trie access
- lpmopen() allocates and initializes an empty trie
- lpmputmask() adds the IPV4 address/net mask to the lpm structure
- lpmmake() builds the trie in lpm order
- lpmmatch() returns the tag of the longest prefix match
- lpmclose() deallocates the trie

USAGE EXAMPLE

The code synthesis to incorporate library lpm into a Cymbal application is best shown by example.

All code for this example can be found under your Daytona install:

```bash
> cd $DS_DIR/EXAMPLES/usr/orders && ls @lpm
ANS.lpm_test.out day_lpm.cy lpm_check.sh
day_lpm.c lpm_test.msk lpm_test.addr
```

We illustrate how to build, search, and free an IPV4 trie by constructing C external wrapper-FUNs callable from Cymbal, which in turn invoke C functions from within library lpm. The key role these C wrappers play is to hold the trie and pointers and to facilitate the transfer of Cymbal IP or IPNET addresses to their C equivalent Ipnet_Uni types for lpm lib integration.

C Wrapper Files

We define "day_lpm.c" to declare and define 3 static variables: a pointer to the lpm trie, a pointer to its method-type, and its discipline struct. We can now define our custom C-wrapper functions callable from Cymbal with this API.

This C file need not be stand alone; it could, for instance, be included within an application’s *.env.c file and serve the same purpose. This file provides the interface between Cymbal and the lpm library.
The function my_lpminit clears out and initializes the discipline function, initializes the method to retrieve via the lpmmeth call, and allocates and initializes the lpm structure via the lpmopen call. The load_ipnet_uint function splits the Ipnet_Uint (which is the C equivalent of the Cymbal IPNET(uint)) into the component Ip and Mask. The IP is converted from network to host order via htonl. The IP net and mask are added to the lpm trie via the lpmputmask call. The my_lpmmake, my_lpmmatch, and my_lpmclose functions are straight pass throughs from Cymbal adding the static lpm structure pointer as an argument.

Cymbal File
Now the process for trie construction and then utilization for matching can be readily embodied within the commented Cymbal program "day_lpm.cy".

Test Data Files
Data file lpm_test.msk contains 41327 IP addresses/masks: these will be built into the search trie for which we’ll use lib lpm to find longest prefix matches. Here are several test entries within this file:

152.16.0.0/12
152.158.0.0/16
152.184.0.0/13

Data file lpm_test.addr contains 110 test IP addresses. Some examples include:

152.186.221.115
152.117.209.209
152.120.146.101

Output file ANS.lpm_test.out can be used to verify your test results. For example:

Best match for 152.186.221.115 is at index 3458 = 152.184.0.0/13

KSH Build File
The file "lpm_check.sh" illustrates how to create a Daytona executable which one can run to test a trie of masks against a file of addresses to find the longest prefix match, if it exists. The critical point here is to use the same C-compiler flags that Daytona uses.

SEE ALSO
Index

+  
-  

?  

?  

_  

Daytona 2013-07-26
Daytona

- conn_aborted_.............................. 8-5, 22-8
- conn_refused_.............................. 8-5, 22-6
- conn_reset_................................. 8-12, 8-19
- continue_.................................. 19-37
- data_...................................... 2-4, 4-5, 9-4, 13-35
- deadlock_.................................. 8-5, 8-12
- Define_.................................... 3-38
- desc_...................................... 2-4, 4-5, 9-4, 13-35
- device_full_............................... 8-12
- dummies_.................................. 13-23
- dummy_.................................... 13-23
- eai_addrfamily_........................... 7-25
- eai_again_................................. 7-25
- eai_badflags_.............................. 7-25
- eai_fail_.................................. 7-25
- eai_family_................................ 7-25
- eai_memory_................................. 7-25
- eai_nodata_................................. 7-25
- eai_noname_................................. 7-25
- eai_overflow_............................... 7-25
- eai_service_................................. 7-25
- eai_socktype_............................... 7-25
- eai_system_................................. 7-25
- EC ........................................ 7-38
- exclusive_................................ 17-8
- fail_on_block................................ 8-4
- failed_comparison.......................... 8-19
- failedKid_comparison........................ 8-31, 19-15
- failed_re_match............................ 8-19
- fifo........................................ 8-1
- fifo........................................ 8-26
- file........................................ 8-1, 8-3
- file_notthere................................ 8-3
- filesize_exceeded.......................... 8-12
- fin.......................................... 8-10
- fin1000s..................................... 8-10
- fipfio...................................... 8-1
- fopen_failed................................ 8-5
- funnel...................................... 8-1
- hangup...................................... 19-37
- host_notfound................................ 7-25
- host_unreachable............................ 8-5, 8-12, 8-19, 22-6
- huge........................................ 5-2
- Init_Time_Store............................. 7-32, 19-13
- instant_eoc................................ 8-19
- int.......................................... 5-2
- interrupt.................................. 19-37
- interrupted................................ 8-5, 8-12, 8-19
- import_forbidden............................ 22-6
- iterative_server........................... 22-2
- kill......................................... 19-37
- killed_kid.................................. 8-31
- last........................................ 12-18
- last_signal.................................. 17-29
- link_down.................................. 8-12
- long......................................... 5-2
- Lowercase.................................. 7-38
- missing_terminator.......................... 8-19
- missing_value.............................. 8-19
- Must_Eq..................................... 7-38
- nbr_of_args................................ 8-1
- negative_read.............................. 8-19
- net_unreachable............................. 8-5
- no_data..................................... 7-25
- no_dot...................................... 8-10
- no_lock..................................... 17-8
- no_recovery................................. 7-25
- null_chan................................... 8-5
- null_str..................................... 5-4
- null_vbl..................................... 5-32
- ok_kid....................................... 19-15
- overflow.................................... 8-19
- packet....................................... 2-4, 4-5, 9-4, 13-35
- parent....................................... 19-18
- pipe.......................................... 8-1
- pipe_broken.................................. 19-37
- plain........................................ 8-10
- popcorn..................................... 8-1, 19-19
- present..................................... 3-2, 5-50
- prev_chan................................... 8-20
- Print_Time_Delta............................ 7-32, 19-13
- quit......................................... 19-37
- quota_exceeded............................. 8-12
- read.......................................... 8-2
- safe.......................................... 4-5, 9-4, 9-6, 13-35
- Say........................................... 7-39
- Say_Eq....................................... 6-13, 7-35, 7-38
- share........................................ 17-8
- short......................................... 5-2
- Show_Exp_To.................................. 6-13, 7-38
- Show_Vbl_To.................................. 7-38
- stale_nfs.................................... 8-12, 8-19
- stderr....................................... 8-1
- stdin........................................ 8-1
- stdout....................................... 8-1
- stop.......................................... 19-37
- stopped_kid.................................. 8-31
- string....................................... 8-1, 8-3, 8-31
- system_error............................... 8-5, 8-12, 8-19
- table......................................... 2-4, 4-5, 9-4, 13-35
- tcp.......................................... 8-1
- terminate.................................... 19-37
- text.......................................... 8-1
- timed_out.................................... 8-5, 8-12, 22-6
ds_m4, problem with ................................... 6-32
DS_MAKE ............................................. 3-30
DS_PATH ............................................. 3-27, 6-30
DS_PROJ .............................................. 3-5, 3-27, 6-30
DS_RFLAGS .......................................... 3-28
DS_Set .............................................. 3-27
DS_SFILGS .......................................... 3-29, 4-5
DS_SORT .............................................. 3-54
DS_SORTOPTS ........................................ 3-54
DS_SQLONLY .......................................... 3-29, 4-5
DS_TFLAGS .......................................... 3-29
DS_Unset ............................................ 3-28
ds_whence .......................................... 3-30
DS_ZFLAGS .......................................... 3-29
DSQL .................................................. 4-1
Dump .................................................. 13-34
duplicate answers .................................. 4-3, 9-32, 12-22
dynamic ............................................. 6-20
dynamic hparti ...................................... 23-16
dynamic SQL ........................................ 18-17
dynara ............................................... 11-2
dynara-former ...................................... 11-18

E
Edit ................................................... 3-34
EDITOR .............................................. 3-34
efficiency, space .................................. 3-26
efficiency, speed .................................... 3-24
Elv_Count .......................................... 11-5, 12-13
emacs ............................................... 3-34
EMACS .............................................. 3-34
embedded new-lines ................................ 3-4
embedded SQL ...................................... 18-17
Emacs .............................................. 3-34
endian .............................................. 5-21, 7-24
ending_with ...................................... 8-7, 8-15
Ends[] .............................................. 7-8
ensuring_string_sanity ......................... 17-5
ensuring_string_sanity_nonascii_ok ........ 17-5
Env ............................................... 3-27
env.cy ............................................. 6-30
ev_fl_for_fpp ...................................... 3-36
Eol_Seen[] ........................................ 8-27
EP .................................................... 5-18
error, system ..................................... 3-50
escaped new-lines ................................ 3-4
Eval_Dicts ......................................... 23-38
EXAMPLES .......................................... 3-30
exceptions ........................................ 6-41
Exclaim() .......................................... 8-7
Exclaim_Signal_Msg .............................. 19-38
exclusive or ....................................... 7-2
Exec ................................................. 3-44
Executable_Path .................................. 3-8
eXistent quantifier ................................ 5-40
Exit ............................................... 6-5
Exit() .............................................. 7-30
extending_on_val_err ............................ 17-5
exp() ............................................... 7-2
expanded_fl_name ................................ 7-28
export ............................................. 6-28
exports ............................................ 6-27
Expr ............................................... 3-43
extended predicate .............................. 5-40
extension ......................................... 5-1
extensional ........................................ 13-1

F
fl_4() ............................................... 8-10
fcmb() .............................................. 8-11
fet ............................................... 6-15, 6-16
FIELD .............................................. 5-47
FIELD cast ........................................ 4-6
field filtering ................................... 23-34
fio ................................................. 23-31
file glob ......................................... 10-7
file_acc_time() ................................... 7-26
FILE_BASE ......................................... 3-9
File_Exists[] ...................................... 7-25
file_gld() ......................................... 7-26
FILE_INFO_FILE .................................. 23-7
File_is_Openable_For[] ......................... 7-26
File_is_Ordinary[] .............................. 7-26
File_is_Read_Only[] ............................. 7-26
File_isa_Dir[] .................................... 7-26
File_isa_Fifo[] ................................... 7-26
file_mod_time() ................................... 7-26
file_mode() ....................................... 7-26
file_size() ....................................... 7-26
file_stat_time() ................................... 7-26
file_uid() ........................................ 7-26
Filter_Fun ........................................ 23-34
find-it tools ..................................... 3-36
filters, data/formatting ...................... 3-33
financial_block() ................................ 8-10
financial_comma_block() ...................... 8-11
Find_Dict ......................................... 23-38
finditically define ............................. 9-19
finditically define variables ................. 9-28
finditically define ............................... 9-35
fio file .......................................... 3-37
Fio_Dir ........................................... 23-29
<table>
<thead>
<tr>
<th>Function</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>first_day_of_month()</td>
<td>7-16</td>
</tr>
<tr>
<td>flat file</td>
<td>3-1</td>
</tr>
<tr>
<td>FLOAT</td>
<td>5-2</td>
</tr>
<tr>
<td>floor()</td>
<td>7-2</td>
</tr>
<tr>
<td>FLT</td>
<td>5-3</td>
</tr>
<tr>
<td>FLT, indexing/rounding issue</td>
<td>5-3, 11-1, 12-21</td>
</tr>
<tr>
<td>fit_for_time()</td>
<td>7-22</td>
</tr>
<tr>
<td>Flush()</td>
<td>8-30</td>
</tr>
<tr>
<td>Flush, otherwise</td>
<td>8-12</td>
</tr>
<tr>
<td>flush_on_return</td>
<td>6-38, 17-27, 17-29</td>
</tr>
<tr>
<td>flushing</td>
<td>8-7</td>
</tr>
<tr>
<td>for</td>
<td>8-3</td>
</tr>
<tr>
<td>for_each_time</td>
<td>6-14, 12-32</td>
</tr>
<tr>
<td>for_each_time_loop</td>
<td>9-15</td>
</tr>
<tr>
<td>for_each_time, break</td>
<td>6-23</td>
</tr>
<tr>
<td>for_each_time, continue</td>
<td>6-23</td>
</tr>
<tr>
<td>for_the_first_time</td>
<td>6-18, 12-28, 12-30</td>
</tr>
<tr>
<td>for_the_last_time</td>
<td>6-18, 6-20</td>
</tr>
<tr>
<td>for_which</td>
<td>5-52</td>
</tr>
<tr>
<td>forbidden chars</td>
<td>3-1, 17-6</td>
</tr>
<tr>
<td>format, description</td>
<td>9-4</td>
</tr>
<tr>
<td>format, output</td>
<td>3-33</td>
</tr>
<tr>
<td>format, packet</td>
<td>2-4, 3-32, 4-5, 8-11, 9-4, 13-35</td>
</tr>
<tr>
<td>format, table</td>
<td>2-4, 3-32, 4-5, 8-11, 9-4, 13-35</td>
</tr>
<tr>
<td>format, xml</td>
<td>9-4</td>
</tr>
<tr>
<td>fpp</td>
<td>6-25</td>
</tr>
<tr>
<td>fpp as args</td>
<td>6-35</td>
</tr>
<tr>
<td>fpp_tracing</td>
<td>3-42</td>
</tr>
<tr>
<td>fppvbls</td>
<td>6-35</td>
</tr>
<tr>
<td>free tree</td>
<td>3-25, 17-3</td>
</tr>
<tr>
<td>free variable</td>
<td>5-42</td>
</tr>
<tr>
<td>free_on_begin_when</td>
<td>6-38</td>
</tr>
<tr>
<td>free_on_return_when</td>
<td>6-38</td>
</tr>
<tr>
<td>freeing boxes (not)</td>
<td>12-27</td>
</tr>
<tr>
<td>freeing boxes, dynara</td>
<td>6-38</td>
</tr>
<tr>
<td>freeing dynara</td>
<td>11-8</td>
</tr>
<tr>
<td>from_a_bin_sample_of_fraction</td>
<td>13-47</td>
</tr>
<tr>
<td>from_a_bin_sample_of_size</td>
<td>13-47</td>
</tr>
<tr>
<td>from_section</td>
<td>19-5</td>
</tr>
<tr>
<td>from_shmem</td>
<td>15-2, 21-2</td>
</tr>
<tr>
<td>from_version</td>
<td>23-41</td>
</tr>
<tr>
<td>fsync</td>
<td>8-30, 17-21</td>
</tr>
<tr>
<td>ftokens</td>
<td>10-6</td>
</tr>
<tr>
<td>function calls</td>
<td>5-35</td>
</tr>
<tr>
<td>funnel</td>
<td>19-30</td>
</tr>
</tbody>
</table>

**G**

| Generalized hparti                | 15-35   |
| Get_File_Lock                     | 7-27    |
| get_hard_resource_limit           | 7-34    |
| Get_Load_Avg                      | 7-33, 20-10 |

**H**

| hash_join                        | 13-15, 20-13, 21-12 |
| hash_join, outer                 | 13-27    |
| hats, DSQL                        | 4-8      |
| hatting                           | 3-8      |
| HEKA                              | 3-26, 5-14 |
| heka_as_str                       | 5-15    |
| HEKCLOCK                          | 5-16    |
| HEKDATE                           | 5-16    |
| HEKDATE_CLOCK                     | 5-16    |
| HEKINT                            | 3-26, 5-14 |
| HEKSTR                            | 5-14    |
| HEKTIME                           | 5-16    |
| HEKUNIT                           | 5-14    |
| helper fpp                        | 6-25    |
| hex                               | 5-2     |
| hex_str_for_int()                 | 7-3     |
| hidden new-line                   | 5-4     |
| homonym variables                 | 3-42    |
| homonym VBLs                      | 5-43    |
| horizontal partitioning           | 3-52, 3-54, 8-4, 23-1 |
| hparti                            | 23-3, 23-4 |
| hparti, dynamic                   | 23-16   |
L

lambda OPOCFUN ....... 15-47
last_day_of_month() ....... 7-46
lb() ....... 8-10
least fixed point ....... 16-3
leave ....... 6-22
left outer join ....... 13-21
left outer join, DSQL ....... 4-7
left_block() ....... 8-10
length() ....... 7-9, 7-30
lexicographic order ....... 7-9, 12-9
linear recursion ....... 6-3
LIST ....... 5-20
LIST/SET-valued field ....... 3-1, 3-15, 3-16, 3-25, 3-48, 14-4
LIST/SET-valued field, in keys ....... 3-20
LIST/SET-valued FIELDS ....... 17-12
LIST/SET-valued FIELDS, updating ....... 17-12
LIT ....... 5-11
local ....... 6-28
local_date_clock_for() ....... 7-30
local_date_clock_for_unix_time() ....... 7-22
locals ....... 6-27
localtime_for_unix_time() ....... 7-32
lock files ....... 17-28
lock files, orphans ....... 17-28
Lock_Blockers_For_File_Path ....... 7-28
Lock_Box ....... 20-30
lock_file() ....... 7-27
locking ....... 17-36
log files ....... 17-31
log() ....... 7-2
log_option ....... 17-31
logging ....... 17-31
logging transactions ....... 17-21
loop ....... 6-11
loop_again ....... 6-22
LOWER ....... 5-25
lower_off() ....... 7-11

M

m4 ....... 3-57
M4 ....... 3-58
M4PATH ....... 3-57
macro predicate ....... 15-1
macros ....... 3-57
Maintain_Index_Stats ....... 3-47
Make_AR ....... 3-8
Make_CC ....... 3-8
MAKE_GOODS ....... 3-8, 3-9
Make_LD ....... 3-8
Make_Nicer_By ....... 19-4
Malloc_Stats ....... 11-3
manifest ....... 6-29, 6-35
manifest dynara ....... 11-2
manifest TUPLE ....... 6-2
map element ....... 11-2
map ....... 11-2
masked_ip ....... 7-24
Matches[] ....... 7-7
matching ....... 8-16
matrix ....... 5-41
matrix occurrence ....... 5-42
matrix of OPCODE ....... 9-10
max ....... 14-3
max() ....... 7-15
Max_Do_Queue_in_K ....... 17-26
Max_Log_File_in_K ....... 17-31
Max_Open_Bins ....... 17-28, 23-6
Max_Open_Random_Acc_Bins ....... 23-6
Max_Open_Seq_Acc_Bins ....... 23-6
max_time() ....... 7-22
Max_Value ....... 3-17, 3-48, 17-5
MDQ ....... 17-26
meaning ....... 5-51
median ....... 14-3, 14-9
medians ....... 12-18
merging_with_lexico_order ....... 20-5
merging_with_reverse_lexico_order ....... 20-5
merging_with_sort_spec ....... 20-5
message ....... 19-11
message protocol ....... 8-30
message terminator ....... 19-11
messages ....... 8-15
min ....... 14-3
min() ....... 7-15
min_time() ....... 7-22
Min_Value ....... 3-17, 3-48, 17-5
missing value ....... 3-32, 13-35
missing value, 1/O channel ....... 8-16
missing values ....... 3-1, 3-48, 4-1, 13-31
mkdir ....... 7-28
mkdirp ....... 7-28
MLF ....... 17-31
mod ....... 7-1
mod_clock_time() ....... 7-22
mod_flt() ....... 7-1
mod_hg ....... 7-1
mod_int ....... 7-1
mod_re ....... 7-10
mod_sg() ....... 7-1
mod_str ....... 7-10
null value ........................................ 3-2, 3-48, 4-1, 13-32
null .................................................. 3-2, 8-2
object record ........................................ 3-4, 5-17
objcond ............................................. 5-9
occurrence, matrix .................................. 5-42
occurrence, scoped .................................. 5-42
occurrence, scoping .................................. 5-42
octal_str_for_int() .................................. 7-3
ODBC ............................................. 18-17
offset .............................................. 8-31
on-line backup ...................................... 8-4
on_abort_return .................................... 6-38, 17-29
one which ........................................... 5-52
one which is ......................................... 5-52
OPCOFUN .......................................... 15-47
OPCOND ........................................... 9-9
OPCOND_SOMEHOW vbls .............................. 9-10
open assertion ...................................... 5-42
open-coded condition ............................... 9-9
order by ............................................ 4-13
otherwise ............................................ 6-4
otherwise, Flush .................................... 8-12
otherwise, Write .................................... 8-12
otherwise_ok ........................................ 6-6
otherwise_switch .................................... 6-6
Output_Filter_Fun .................................. 23-34
outside ............................................. 15-3, 15-28
outside variable ..................................... 5-44, 9-10

N
Name ................................................ 5-51
named .............................................. 5-51
named_pipe ......................................... 23-31
nano_de_now() ...................................... 7-18
nanosecs_of ........................................ 7-21
nbr_config_cpus .................................... 7-33
nbr_online_cpus ..................................... 7-33, 20-10
nbr_values_found .................................... 8-30
negation as failure .................................. 15-4
negation() .......................................... 7-37
nested_tables ....................................... 15-16
new_lines, embedded ................................ 3-4
new_lines, escaped .................................. 3-4
new_channel ........................................ 8-2
new_channel_call_status ............................ 8-2, 8-5
Next_By_Justia_Fam ................................ 13-2
Next_By_Offset_Fam ................................ 13-2
next_day_of_week() .................................. 7-15
next_io_ready_tendril ................................ 19-24
next_waited_for_tendril ......................... 19-12
nice .................................................. 19-4
no_argument_keyword ................................ 5-35
no_lock_files ....................................... 17-28
no_share_lock_files ................................. 17-28
NonUnique_Btree_Key_Fam ............................ 13-2
Not .................................................. 5-30
not Ensuring_string_sanity ......................... 17-5
note .................................................. 3-8
notrustme ........................................... 3-28
null .................................................. 3-2

O
OBJECT ............................................. 5-2
object_record ....................................... 3-4, 5-17
occurrence, matrix .................................. 5-42
occurrence, scoped .................................. 5-42
occurrence, scoping .................................. 5-42
octal_str_for_int() .................................. 7-3
ODBC ............................................. 18-17
offset .............................................. 8-31
on-line backup ...................................... 8-4
on_abort_return .................................... 6-38, 17-29
one which ........................................... 5-52
one which is ......................................... 5-52
OPCOFUN .......................................... 15-47
OPCOND ........................................... 9-9
OPCOND_SOMEHOW vbls .............................. 9-10
open assertion ...................................... 5-42
open-coded condition ............................... 9-9
order by ............................................ 4-13
otherwise ............................................ 6-4
otherwise, Flush .................................... 8-12
otherwise, Write .................................... 8-12
otherwise_ok ........................................ 6-6
otherwise_switch .................................... 6-6
Output_Filter_Fun .................................. 23-34
outside ............................................. 15-3, 15-28
outside variable ..................................... 5-44, 9-10

P
package, Cymbol .................................... 18-2
packages, Cymbol ................................... 18-19
packet format ....................................... 2-4, 3-32, 4-5, 8-11, 9-4, 13-35
Pad_To_LEN ......................................... 3-51, 17-10
par .................................................... 3-9, 17-31
parallel Sizup ....................................... 3-53
parallel_for ........................................ 20-1
parallelizing ........................................ 20-1
partition by .......................................... 4-15
partitioning directory-based ...................... 23-26
partitioning_horizontal ............................ 23-2
to_partitioning .......................... 23-2
partitioning_vertical ............................. 23-2
PARTITIONING Fields .............................. 23-6
passive close ........................................ 22-2
Pause() ............................................. 7-30
pqd .................................................. 22-13
performance ......................................... 3-19, 3-24, 13-49, 17-7
performance, Sizup ................................ 3-54
Perl DBI ......................................... 18-17
physical RECORD_CLASS ......................... 15-11
physical_mem_in_K ................................ 7-33
physical_mem_in_M ................................. 7-33
pipe, named ......................................... 23-31
pipe_FILES ........................................ 23-30
pipe_status ...................................... 8-31
pipein_FILES ........................................ 23-30
pipeout_FILES ...................................... 23-30
piping_to_cmd .................................... 9-17
pjdr .............................................. 3-9, 17-31
placeholder ...................................... 12-10
plus_clock_time() ............................... 7-17
plus_date_clock_time ............................. 7-19
plus_str() ......................................... 7-9
plus_str_int() .................................... 7-11
plus_time() ....................................... 7-22
pointer, Cymbal ................................... 5-31
pow() ............................................. 7-2
preceqence, composite type .................... 5-33
predicate .......................................... 5-36
prefix index searches ............................ 13-7
Print_Current_Sighandler_For ................ 19-38
Proc_Exists() ..................................... 7-30
Proc_Exists_On() .................................. 7-30
Proc_is_Alive_And_Well() .................... 7-30
procedural ......................................... 9-1
procedure calls .................................. 6-23
prod ............................................. 14-3
PROJ ............................................. 17-31
project .......................................... 3-5
project_description ............................. 3-5
project.env.in ................................... 3-58
ptokens ........................................... 10-6
put_shell_env() ................................... 7-34
Python DBAPI ..................................... 18-17

Q

QQ .................................................. 3-42
quantifier ....................................... 5-41
quantifier, existential .......................... 5-40
quantifier, universal ............................ 5-41
quantile ......................................... 14-3, 14-4, 14-9
quantiles ........................................ 12-18
Query_Output_Path ................................ 3-8
Query_Path ....................................... 3-8
Query_Preprocessor Invocation ............... 3-20

R

Raise .............................................. 6-42
rand_int() ....................................... 7-2
rand_uni() ....................................... 7-2
Random_Acc_Bufsize .............................. 3-11, 3-26, 23-32
range query ...................................... 13-9
rb() ............................................. 8-10
rcl .................................................. 8-10
rcl, targetable .................................. 23-28
rcl, HPART1_BIN ................................ 23-11
RDA .................................................. 18-17
RE .................................................. 5-5
RE_Match_Failed .................................. 7-6
RE_Match_Worked .................................. 7-6
read() ........................................... 8-14
read() ........................................... 8-14
read_call_status .................................. 8-18
read_Failed ....................................... 8-18
read_line ......................................... 8-15
Read_Only ......................................... 23-33
read_words ......................................... 8-15
Read_Worked ....................................... 8-18
rebuild_app_objs ................................ 3-42
rec_tracing ....................................... 3-42
record class ...................................... 3-4, 3-5, 5-47
record class description ...................... 3-10
record length, max ................................ 3-1
record slot ....................................... 13-51
Recover .......................................... 17-31
rectify ............................................. 5-43
reduced_channel .................................. 8-5
regular expressions ............................. 7-4
relation .......................................... 3-4
Relocate .......................................... 3-40
rename ............................................. 7-29
rename.m .......................................... 3-57
renewing_with .................................... 6-19, 20-20
report writers .................................... 6-20
Resync ............................................. 3-37
return ............................................. 6-16, 6-22, 6-37
reuse, data buffer ................................ 13-34
Reuse_Freed_Space ................................ 3-25, 13-52, 17-17
Rewind() .......................................... 8-30
right_block() ..................................... 8-10
Rm_Lock_File_For ................................ 7-27
mdir .............................................. 7-29
mdirp ........................................... 7-29
Rollback .......................................... 17-23
rollback flag ...................................... 17-32
rooted_expanded_fl_name ....................... 7-28
round_down_to_for_clock() ...................... 7-28
round_down_to_for_date_clock() ............... 7-19
round_down_to_for_time() ....................... 7-23
row_number ....................................... 4-12
rtn() ............................................. 8-11
S

SAFE_STR ........................................... 5-11
same() ............................................. 7-37
sampling ........................................... 13-47
satchain ........................................... 5-35
satisfaction claim ................................ 5-35
satisfaction LIST .................................. 9-19
satisfy ............................................ 9-2
Save .............................................. 17-23
scan-and-quit key .................................. 3-22
schema evolution .................................. 23-40
scope .............................................. 5-41, 5-44
scope of OPCOND vnls ............................... 9-10
scoped occurrence ................................ 5-42
scopes of variables ................................. 6-28
scoping occurrence ................................ 5-42
scoping of variables ............................... 5-43
sCP ................................................... 5-43, 6-15
secs_for_hms() .................................... 7-18
secs_off() ......................................... 7-23
Seek_In() ........................................... 8-30
selecting_when ..................................... 12-11, 16-22
selection index .................................... 12-11
selection order .................................... 12-9
Self .................................................. 5-51
semicolon ......................................... 6-28
semicolons ......................................... 6-7
separator .......................................... 8-15
Seq_Ace_Bufsize .................................. 3-26, 23-32
Serial_Nbr ......................................... 17-22
set ................................................. 6-1
SET .................................................. 5-36
set function .......................... 6-3, 11-13, 21-22
set-former ......................................... 12-1
Set_Bit_For ....................................... 5-18
Set_Hard_Resource_Limit ......................... 7-34
Set_Len_To ....................................... 5-5
Set_Soft_Resource_Limit ......................... 7-34
shared memory .................................... 21-1
shell_date() ....................................... 7-15
shell_echo() ....................................... 7-34
shell_env() ......................................... 7-34
shell_eval() ....................................... 7-34
Shell_Execute ...................................... 19-42
shell_exec() ....................................... 7-33
Shell_Execute() .................................... 7-33
SHELLLP ............................................ 19-41
shift, bitwise ...................................... 7-2
shmenv ............................................. 15-1
shmenv VBL VBL ................................. 21-1
Show .............................................. 3-32

SIGALRM ............................................ 19-38
Signal .............................................. 19-36
sin() .............................................. 7-2
size file ......................................... 3-23, 3-25, 3-54
size Access ....................................... 3-25, 3-51
size ................................................. 5-2
Sizup .............................................. 3-44
Sizup, parallel ................................... 3-53
skipping .......................................... 8-7, 13-51, 13-52
skolem ............................................ 2-15, 6-7, 8-9, 8-24, 12-19
Sleep() .............................................. 7-29
slot, record ....................................... 13-51
so that_previous ................................. 12-33, 14-21
some-of-all ....................................... 20-6
somehow .......................................... 5-44
sort spec .......................................... 12-9
sorted_by_index .................................. 13-3
Source ............................................ 3-11, 3-15
space efficiency .................................. 3-26
spawn .............................................. 19-2, 19-40
specifier, subclass ................................ 5-2
speed .............................................. 3-24
splice .............................................. 7-10, 7-36
sql-if_dummy_then ................................ 13-27
sqrt() ............................................. 7-2
squawks ........................................... 3-57
sqr .................................................. 3-4
Stacy .............................................. 3-41, 4-4
Starts[] .......................................... 7-8
Starts_With ....................................... 13-7
stated size ....................................... 8-8
static ............................................. 6-29
stdinFILES ....................................... 23-29
stdoutFILES ..................................... 23-30
stokens .......................................... 10-6
stop_finding_children_when .................... 16-25
stopping_when .................................... 12-11, 16-23
STR(*) ............................................. 6-28
STR(=) ............................................. 6-28
str_as_heka ....................................... 5-15
str_as_ip_heko .................................... 5-21
str_as_ipport_hekop ............................. 5-21
Str_Ends_Str[] .................................... 7-8
str_for_bitseq .................................... 5-18
str_for_chan() .................................... 8-30
str_for_clock() ................................... 7-17
str_for_date() .................................... 7-15
str_for_date_clock() ......................... 7-18
str_for_dec() ..................................... 8-10
str_for_time() ................................... 7-22
str_for_utty_array() ......................... 7-14
Str_Starts_Str[] .................................. 7-8
<table>
<thead>
<tr>
<th>Function/Concept</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>type-factoring</td>
<td>6-31, 6-33</td>
</tr>
<tr>
<td>type_dummy</td>
<td>13-25</td>
</tr>
<tr>
<td>typed_like</td>
<td>6-41</td>
</tr>
<tr>
<td>utctime</td>
<td></td>
</tr>
<tr>
<td>U</td>
<td></td>
</tr>
<tr>
<td>UINT</td>
<td>5-2</td>
</tr>
<tr>
<td>uint_for_binary_str()</td>
<td>7-3</td>
</tr>
<tr>
<td>uint_for_ip()</td>
<td>7-23</td>
</tr>
<tr>
<td>undo</td>
<td>17-23</td>
</tr>
<tr>
<td>Unique</td>
<td>3-18, 23-4</td>
</tr>
<tr>
<td>Unique_Btree_Key_Fam</td>
<td>13-2</td>
</tr>
<tr>
<td>Unit_Sep</td>
<td>3-15</td>
</tr>
<tr>
<td>Unit_Seps</td>
<td>3-15</td>
</tr>
<tr>
<td>universal quantifier</td>
<td>5-41</td>
</tr>
<tr>
<td>UNIX flat file</td>
<td>3-1</td>
</tr>
<tr>
<td>unix_time()</td>
<td>7-32</td>
</tr>
<tr>
<td>unix_time_for_utc_date_clock()</td>
<td>7-20</td>
</tr>
<tr>
<td>unlink()</td>
<td>7-27</td>
</tr>
<tr>
<td>Unlink()</td>
<td>7-27</td>
</tr>
<tr>
<td>Unlock_Box</td>
<td>20-26</td>
</tr>
<tr>
<td>unlock_file()</td>
<td>7-27</td>
</tr>
<tr>
<td>Unset_Bit_For</td>
<td>5-18</td>
</tr>
<tr>
<td>until loop</td>
<td>6-11</td>
</tr>
<tr>
<td>update_box_element</td>
<td>12-20</td>
</tr>
<tr>
<td>update_record</td>
<td>17-9</td>
</tr>
<tr>
<td>updating_LIST-valued_FIELDS</td>
<td>17-12</td>
</tr>
<tr>
<td>updating_TUPLE-valued_FIELDS</td>
<td>17-14</td>
</tr>
<tr>
<td>updating_views</td>
<td>15-43</td>
</tr>
<tr>
<td>UPLOW</td>
<td>3-18, 5-25</td>
</tr>
<tr>
<td>uplow_offset()</td>
<td>7-11</td>
</tr>
<tr>
<td>UPPER</td>
<td>5-25</td>
</tr>
<tr>
<td>upper_offset()</td>
<td>7-11</td>
</tr>
<tr>
<td>upthru</td>
<td>8-15</td>
</tr>
<tr>
<td>upto</td>
<td>8-14</td>
</tr>
<tr>
<td>usage</td>
<td>3-32, 3-44</td>
</tr>
<tr>
<td>Usage</td>
<td>3-44</td>
</tr>
<tr>
<td>Use_Box</td>
<td>12-8</td>
</tr>
<tr>
<td>useless_VBL</td>
<td>15-22</td>
</tr>
<tr>
<td>user_C-extension</td>
<td>3-9</td>
</tr>
<tr>
<td>user_C-extensions</td>
<td>6-30</td>
</tr>
<tr>
<td>using_no_index</td>
<td>13-4</td>
</tr>
<tr>
<td>using_reverse_siz</td>
<td>13-4, 17-17</td>
</tr>
<tr>
<td>using_siz</td>
<td>13-4</td>
</tr>
<tr>
<td>using_source</td>
<td>13-38</td>
</tr>
<tr>
<td>usr_env.c</td>
<td>6-40</td>
</tr>
<tr>
<td>usr_env.cy</td>
<td>3-56, 6-30</td>
</tr>
<tr>
<td>usr_env.m</td>
<td>3-58</td>
</tr>
<tr>
<td>utc_date_clock_for()</td>
<td>7-30</td>
</tr>
<tr>
<td>utc_date_clock_for_unix_time()</td>
<td>7-19</td>
</tr>
<tr>
<td>utc_date_clock_now()</td>
<td>7-19</td>
</tr>
<tr>
<td>utctime_for_unix_time()</td>
<td>7-32</td>
</tr>
<tr>
<td>UTF-8</td>
<td>5-13</td>
</tr>
<tr>
<td>uty_for_ch()</td>
<td>7-14</td>
</tr>
<tr>
<td>V</td>
<td></td>
</tr>
<tr>
<td>ValCall</td>
<td>5-32</td>
</tr>
<tr>
<td>Validation_RE</td>
<td>3-17, 3-48, 17-5</td>
</tr>
<tr>
<td>var</td>
<td>14-3</td>
</tr>
<tr>
<td>variable</td>
<td>5-25</td>
</tr>
<tr>
<td>VARIABLE</td>
<td>5-32</td>
</tr>
<tr>
<td>variable specification</td>
<td>6-32</td>
</tr>
<tr>
<td>variable, bound</td>
<td>5-42</td>
</tr>
<tr>
<td>variable, free</td>
<td>5-42</td>
</tr>
<tr>
<td>variable, outside</td>
<td>5-44, 9-10</td>
</tr>
<tr>
<td>variables, scoping of</td>
<td>5-43</td>
</tr>
<tr>
<td>variant field</td>
<td>15-12</td>
</tr>
<tr>
<td>variant record</td>
<td>15-12</td>
</tr>
<tr>
<td>VBL</td>
<td>5-32</td>
</tr>
<tr>
<td>VBL VBL</td>
<td>5-31</td>
</tr>
<tr>
<td>vbl_tracing()</td>
<td>3-42</td>
</tr>
<tr>
<td>VBLSPEC</td>
<td>6-36</td>
</tr>
<tr>
<td>Version</td>
<td>23-40</td>
</tr>
<tr>
<td>version, query</td>
<td>6-26</td>
</tr>
<tr>
<td>version, red</td>
<td>23-40</td>
</tr>
<tr>
<td>vertical partitioning</td>
<td>15-15</td>
</tr>
<tr>
<td>very large transaction (VLT)</td>
<td>17-25</td>
</tr>
<tr>
<td>VU</td>
<td></td>
</tr>
<tr>
<td>VQ</td>
<td></td>
</tr>
<tr>
<td>waiting, for processes</td>
<td>19-12</td>
</tr>
<tr>
<td>Was_Modified_After[]</td>
<td>7-27</td>
</tr>
<tr>
<td>well-formed assertion</td>
<td>9-20</td>
</tr>
<tr>
<td>when-else command</td>
<td>6-8</td>
</tr>
<tr>
<td>whence()</td>
<td>7-27</td>
</tr>
<tr>
<td>where</td>
<td>5-52</td>
</tr>
<tr>
<td>which</td>
<td>5-52</td>
</tr>
<tr>
<td>which_is</td>
<td>5-52</td>
</tr>
<tr>
<td>while loop</td>
<td>6-11</td>
</tr>
<tr>
<td>wipe_out_nonascii()</td>
<td>7-12</td>
</tr>
<tr>
<td>wipe_out_punct()</td>
<td>7-12</td>
</tr>
<tr>
<td>wipe_out_these_clars</td>
<td>5-3</td>
</tr>
<tr>
<td>wipe_out_these_clars()</td>
<td>7-12</td>
</tr>
<tr>
<td>with_acc_time_vbl</td>
<td>23-10</td>
</tr>
</tbody>
</table>
with_bin_max_rec_len_vbl          23-40
with_bin_min_rec_len_vbl          23-40
with_bin_pos_vbl                  13-44
with_bin_vbl                      23-10
with_byte_size_vbl                23-40
with_candidate_indices_stored     12-12
with_clones_doing_the_do          20-21
with_col_labels                   9-16
with_default                      11-6, 11-16
with_default_arbitrary_order     12-9
with_default_bia                  8-17
with_default_selection_order     12-9
with_deletions_ok                 12-30
with_dirty_reads_ok              3-52, 13-37
with_distance_vbl                 16-22
with_duplicates_ok                12-9
with_elt_vbl_vbl                  12-31, 17-13
with_final_fsync                  17-21
with_format                       2-4, 9-17
with_fsync                        8-30
with_growth_factor                11-3, 12-10, 21-8
with_identity                     16-9
with_ignoring_failed_opens_handler 3-52, 13-37
with_indices_source_vbl           23-10
with_init_max_nbr_elts           11-3, 12-10, 21-8
with_init_size_in_K              11-3
with_lexico_order                  12-9
with_locking                      8-4
with_logging                      17-31
with_logging_flag                 17-31
with_logging_optional             17-31
with_match_length_vbl             7-7
with_match_start_index_vbl        7-7
with_mod_time_vbl                 23-10
with_mode_vbl                     8-2
with_nbr_bin_rec_slots_vbl       23-10
with_nbr_bin_recs_vbl             23-10
with_no_closing                   9-16
with_no_deletions                 11-3
with_no_duplicates                12-9
with_no_heading                   9-16
with_no_logging                   17-31
with_outcount_vbl                 16-22, 16-23
with_output_com_char              9-17
with_output_sep                   9-17
with_path                        16-33
with_patience                     8-4
with_random_indices_stored        12-17
with_rec_map_spec_file_vbl        23-10
with_reverse_lexico_order         12-9
with_selection_indices_stored     12-12
with_sep                         8-7
with_sort_indices_stored          12-12
with_sort_specs                   12-9
with_source_vbl                   23-10
with_stat_time_vbl                23-10
with_sudden_death                 12-29, 17-3
with_title_line                   2-4, 9-16
with_title_lines                  2-6, 9-16
with_tuple_C_format               8-8, 9-17
with_tuple_format                 8-8, 9-17
with_user_sync                    19-33
with_val_err_msg_vbl              17-5
with_version                      6-26
wrapping_up_each_clone_with      20-21
Write()                          8-6
Write_otherwise                  8-12
Write_Line()                     8-7
Write_To_Syslog()                7-32
Write_Words()                    8-7
writing_to_chan                  9-17
writing_to_file                  9-17

X
XML                                9-4, 13-35

Z
zlib()                             8-10
zero_block()                      8-10
zombie process                     23-7, 23-17