

Using Daytona On Network Data: Three Paradigms

Rick Greer

AT&T Research

rxga@research.att.com



What Daytona Does

Daytona™ provides a high-level way to do data management, i.e.,

- to organize & store large amounts of data on disk
- to concisely express sophisticated queries in a very high-level language
- to get the answers to the queries quickly
- to enable users to change their data in a concurrent, crash-proof environment supported by indices and a data dictionary



Principal Advantages of Daytona

- **Speed**
due to its server-free, compilation-based architecture
- **Capacity**
single Daytona tables exist with more than 174 billion records
- **Powerful Cymbal query language**
includes the SQL DML as a subset
- **Simplicity And Reliability**
simple, low-overhead, server-free architecture
simple ASCII flat-file data format available
- **Flexibility**
open to UNIX[®] tools
basic data format open to Perl, awk, etc.
shell-level administration
- **Special Features**



Special Features Of Daytona

- **Compression technology** stores more info per disk
- **Horizontal partitioning** of tables into (many, many) files
- **SPMD query parallelization**
- Scalar and tuple-valued multi-dimensional **associative arrays**
- **Boxes** (an omnibus (in-memory) tuple collection type) used for sorting, duplicate elimination, multiple indexes, caching -- all in-memory.
- **Generalized transitive closure** for querying graphs
- **Set- and list-valued fields**
- fully general, high-level **stream tokenizing**



Power Of Code Generation

Scenario:

Use Cymbal as a programming language
to read a file of Cisco NetFlow records,
each describing a flow between IP+port pairs.

Netflow Record Format:

```
Source_Ip, Source_Netmask, Dest_Ip, Dest_Netmask,  
Packet_Count, Byte_Count
```

Compute the number of flows, total bytes and total packets
between the corresponding network pairs.

Note that here Daytona is being used like Perl/awk on flat file records
without the use of a data dictionary or indices.



Power Of Code Generation (cont.)

Performance Results

	Elapsed Time	Relative Speed
Cymbal	1m55s	1.00
Perlwref	15m37s	8.14
Perltypi	16m3s	8.37
gawk	18m30s	9.65
awkcc	20m3s	10.46
nawk	34m25s	17.96

The winning factors are code generation + compilation plus a clever associative array implementation.

For details, see <http://www.research.att.com/projects/daytona>



Daytona Data Processing Paradigms

- **Full-Service Database** Paradigm
 - Multi-paradigm query language (Cymbal)
 - Data dictionary
 - Data loading with sorting, compression, validation, indices
hence random access!
 - Concurrency, transactions, logging, recovery
- **Programming Language** Paradigm
 - Procedural languages like Perl, C, awk -- and part of Cymbal!
 - None of the above
 - Sequential processing only
- **Hybrid** Paradigm
 - Full power of Cymbal
 - Use data dictionary for convenience
 - No data loading with sorting, compression, or indices

All three paradigms can use Cymbal.



Cymbal

*Cymbal*TM is a very high-level, multi-paradigm programming language.

It synthesizes *procedural* programming constructs
(comparable in power to Perl)

with *declarative* constructs taken from:

- first-order symbolic logic (including generalized transitive closure),
- set theory (i.e., comprehensions),
- (extended) 1989 SQL DML (data manipulation language),
- Cymbal database record descriptions,
- regular expressions for pattern matching *and* lexical analysis.

Daytona processes SQL by translating it into Cymbal first-order logic.

Daytona processes Cymbal by translating it into C.



Using Cymbal

Write Cymbal by using:

- SQL alone,
- procedural Cymbal alone,
- SQL embedded in procedural Cymbal (i.e., as a 4GL)
(better than embedding SQL in C)
- logic, set-theory and procedural Cymbal can be used together,
- any or all of the above paradigms in the same query.

Moral: Use the best paradigm for the problem at hand.

Data Dictionary Conveniences

Data Dictionary == a programmatically accessible compendium of useful information about the data

Information stored in Daytona data dictionaries:

- who, what and where the data files are
horizontal partitioning
- if data produced by pipes, how?
- the names, types, defaults, and layout of the fields
Supports automatic type conversion
LIST/SET-valued FIELDS supported
- how data compressed if at all (by field, by record)
- key fields and kinds of btree indices -- and the seq .siz index
regular btree and cluster btree
- supports blind appends for new data creation



AXIS Netflow Daytona Data Dictionary Report

RECORD_CLASS: AXFLOW

FIELDS:

(hparti)	DATE(_yyyymmdd_)	Start_Date
(hparti)	STR	Router_Ip_Str
Field 1:	STR(1)	Customer_Flag
2:	STR(1)	Direction
3:	STR	Application
4:	IP	Customer_Ip
5:	IP	Endpoint_Ip
6:	HEKCLOCK(_24_hhmmss_)	Start_Clock
7:	HEKTIME(_s_)	Duration
8:	HEKINT(_long_)	Pkts_Sent
9:	HEKINT(_long_)	Octets_Sent
10:	HEKINT(_long_)	Customer_Mask_Bits



11:	HEKINT(_long_)	Endpoint_Mask_Bits
12:	HEKA	Customer_Port
13:	HEKA	Endpoint_Port
14:	HEKINT(_long_)	Tcp_Flags
15:	HEKA	Ip_Proto
16:	HEKA	Ip_Tos

AXIS Netflow Daytona **Data Dictionary** Report (cont.)

RECORD_CLASS: AXFLOW

KEYS:

KEY ca: Non-unique [Customer_Ip, Application]

INDEX ca: Non-unique cluster_btree

KEY a: Non-unique [Application]

INDEX a: Non-unique cluster_btree

KEY ep: Non-unique [Endpoint_Ip, Endpoint_Port]

INDEX ep: Non-unique btree

KEY (hparti): Non-unique [Start_Date, Router_Ip_Str]

Applications: www, https, gss-http, http-alt, compaq-https

Application_Class: web



AXIS Netflow Daytona **Data Dictionary** Report (cont.)

 RECORD_CLASS: AXFLOW

HORIZONTAL PARTITIONING:

Field	Type	Default Value
-----	----	-----
Start_Date	DATE(_yyymmdd_)	^2001-11-3^
Router_Ip_Str	STR	"24.23.184.137"

FILE_INFO_FILE: axflow_fls

Source: \${AXIS_DIR:-.}

Unit_Sep: |

Comment_Beg: #

Tuple_Delims: []



AXIS Netflow Daytona Data Dictionary Report (cont.)

FILE: axflow_data

Source: /netflow/axis/d/2002_04_03/12.123.1.230

Indices_Source: /netflow/axis/i/2002_04_03/12.123.1.230

Reuse Freed Space: no

Rec_Map_Spec_File: \$AXIS_DIR/axflow_dict.CD

Unit_Sep: |

Comment_Beg: #

PARTITIONING_FIELDS:

Start_Date = ^2002-04-03^

. . . // 120 more FILES



AXIS Full-Service Database Group-By Aggregation Query

```
// Compute Total Duration by Application
// and Total [ Pkts_Sent, Octets_Sent ] by Customer/Endpoint Networks

local: DATE .dt ; STR .rtr ;

    TIME(_s_) ARRAY[ STR ] .total_duration = { @ => ^0s^TIME }

    TUPLE[ INT .tot_pkts, INT .tot_octs ] ARRAY [ IP, IP ]
        .pairwise_totals = { @ => [ 0, 0 ] }

set [ .dt, .rtr ] = read( from _cmd_line_ ) otherwise
    with_msg "usage: R <start_date> <router_ip> " do Exit(1);

// more on next slide
```



AXIS Full-Service Database Group-By Aggregation Query (cont.)

```
for_each_time [ .app, .cust_net, .end_net, .dur, .pkts, .octs ]
is_such_that( there_isa AXFLOW where(
    Start_Date = .dt and
    Router_Ip_Str = .rtr and // eliminate to loop
    Application = .app and
    Customer_Ip = .cust_ip and
    Customer_Mask_Bits = .cust_mask and
    Endpoint_Ip = .end_ip and
    Endpoint_Mask_Bits = .end_mask and
    Duration = .dur and Pkts_Sent = .pkts and Octets_Sent = .octs )
and .cust_net = masked_ip( .cust_ip, .cust_mask )
and .end_net = masked_ip( .end_ip, .end_mask )
){
    set .total_duration[ .app ] += .dur;
    set .pairwise_totals[ .cust_net, .end_net ] = [ $#1+.pkts, $#2+.octs ];
}
. . . // output statements given on next slide
```



AXIS Full-Service Database Group-By Aggregation Query (cont.)

```
. . . // print the answers!
do Write_Line( 25 * "=" );
for_each_time [ .app, .tot_dur ] Is_The_Next_Where(
    .tot_dur = .total_duration[ .app ] ) in_lexico_order
{
    do Write_Words( .app, .tot_dur );
}
do Write_Line( 25 * "=" );
for_each_time [ .cust_net, .end_net, .tot_pkts, .tot_octs ]
Is_The_Next_Where(
    [ .tot_pkts, .tot_octs ] = .pairwise_totals[ .cust_net, .end_net ] )
sorted_by_spec[ 2, -4 ]
{
    do Write_Words( .cust_net, .end_net, .tot_pkts, .tot_octs );
}
do Write_Line( 25 * "=" );
```



AXIS Full-Service Database Aggregation Query (cont.)

Advantages Of Full-Service

- Details hidden on data files, directories, file opens/closes
- Details hidden on record layout, field types, defaults
Most types are inferred
- Insulates queries from changes in data files
- TUPLE -> TUPLE associative arrays
Exactly one array lookup per update!
- Two group-by queries on one sequential scan!
- Built-in network data types and functions like IP and *masked_ip()*
- Easy sorting
- Declarative query paradigms available (e.g., logic, SQL)



Perl Analog

```
#!/usr/common/bin/perl
$dt=$ARGV[0];
$rtr=$ARGV[1];
### logistical details
open (DATA, "/netflow/dathome/d/$dt/$rtr") or die;
while( <DATA> ){
    chomp ;
    @f = split( /\|/, $_, 16 );
    ### obscure array refs
    $total_duration{ $f[3] } += $f[7];
    $idx = &network($f[4],$f[10])."|".&network($f[5],$f[11]);

    ### 4 lookups: inefficient: reading/writing for each pkt/oct total
    $pairwise_pkt_totals{ $idx } += $f[8];
    $pairwise_oct_totals{ $idx } += $f[9];
}
```



Comparison Of **Cymbal** With **Perl**

- Daytona is much faster: compiled and better associative arrays
- No need to remember what `$f[13]` is
- Hiding of filesystem/implementation details
- (Persistent record) indices available
 don't even need to own the data!
- Powerful parallelization available.
- Rich variety of scalar types supported (as opposed to int/float/string)
- Cymbal can read binary and fixed format data!
- Cymbal can read filtered data through pipes!

AXIS Programming Language Group-By Aggregation Query

```
// NO DATA DICTIONARY USED HERE:
// same query as before except using ftokens instead of there_isa:
. . .
for_each_time [ STR .app, IP .cust_net, IP .end_net, TIME(_s_) .dur,
               INT .pkts, INT .octs, INT .cust_mask, INT .endp_mask ]
is_such_that(
    [ 2?, .app, .cust_ip_str, .end_ip_str, ?, .dur, .pkts, .octs,
      .cust_mask, .endp_mask, 5? ]
      = ftokens( for "/netflow/dathome/d/.dt/.rtr"ISTR upto "|\n" )
and .cust_net = masked_ip( (IP) .cust_ip_str, .cust_mask )
and .end_net = masked_ip( (IP) .end_ip_str, .endp_mask )
){
    set .total_duration[ .app ] += .dur;
    set .pairwise_totals[ .cust_net, .end_net ] = [ $#1+.pkts, $#2+.octs ];
}
. . .
```



AXIS Hybrid Aggregation Query

-- SAME QUERY AS WITH FULL-SERVICE !! --

The only difference is a suitable annotation in data dictionary to prevent the construction of any indices, including .siz

```
#{ KEYS
    <Siz_Access, no>
    <Indices_Banned, yes> }#
```

Amplifiers

- Use Ted Johnson's **EGIL** for expanded SQL-like aggregation queries on Daytona databases
- Use John Snyder's JDBC and Perl DBI Daytona data interfaces

Summary

Use Daytona to query network data because:

- It's many times faster -- single-threaded and in-parallel !
- You can use indices!
to quickly access just the data you are interested in.
- You can hide and be insulated from implementation details.
- Daytona has built-in IP types and functions
- Daytona has faster, better associative arrays
- Daytona can read from pipes/sockets
- Daytona can read from binary data directly

