Data Stream Warehousing in Tidalrace

Marios Hadjieleftheriou
AT&T Labs - Research
marioh@research.att.com

Theodore Johnson
AT&T Labs - Research
johnsont@research.att.com

Vladislav Shkapenyuk
AT&T Labs - Research
vshkap@research.att.com

ABSTRACT

Big data is a ubiquitous feature of large modern enterprises. Many organizations generate huge amounts of on-line streaming data – examples include network monitoring, Twitter feeds, financial data, and industrial application monitoring. Making effective use of these data streams can be challenging. While Data Stream Management Systems can provide support for real-time alerting and data reduction, many applications require complex analytics on a data history to best make use of the streams.

We have been developing technologies for *data stream warehousing*, starting with the DataDepot [13] system. A data stream warehouse continually ingests data streams, computes complex derived data products, and stores long (perhaps years-long) histories. To take advantage of new technologies, we have developed a next-generation data stream warehousing system. In this paper we describe the *Tidalrace* system, our motivations for developing it, and architectural features of Tidalrace that support data stream warehousing.

1. INTRODUCTION

Modern applications continually generate large volumes of streaming data, ranging from web clicks to financial transactions to instrumentation of industrial processes. Making effective and profitable use of these feeds is the focus of the "Big Data" field. A significant aspect of the value of streaming data is its immediacy. If the data can be processed and analyzed rapidly, the managing entity can take advantage of emerging opportunities or react to critical alerts.

Over the last decade, Data Stream Management Systems (DSMS), such as Borealis [1], GS Tool [7], Streambase [37] and Storm [38], have emerged to perform rapid processing of data streams. These systems generally operate in-memory and have little permanent storage. However, many applications require access to historical as well as real-time data.

For example, the Argus system [43] is designed to detect end-to-end service anomalies in the network of a very large Internet Service Provider (ISP). Argus detects subtle service anomalies, such as excessive TCP retransmissions, by comparing the current state to historical trends.

Traditionally, data warehouses operate on an alternating data loading / data querying cycle. Data collected during operating hours is gathered and, when stable, loaded into the warehouse. During this process, querying is disabled. While this mode of operation permits optimized data loading [11], it is slow to make data available. Even moving from nightly data loading to e.g. hourly data loading is not satisfactory to perform mission critical operations such as network troubleshooting.

The needs of web applications (Facebook, Amazon, etc.) have led to real-time systems for *signature* collection. Data streams can be considered to be collections of records generated by collections of entities. Records of a particular entity are gathered together and summarized as the entity's signature. Streaming updates of signatures have been used for fraud detection [6], and are well suited to representing a customer's interaction with a web site (e.g. shopping cart, advertising profile, etc.). Since the signatures are easily partitioned using the entity's ID, they can be scalably implemented using distributed key-vale stores. An interesting example of this kind of system is Muppet [25], which combines a DSMS front-end with a distributed key-value store back-end.

However, many applications require complex analytics involving wide-ranging data fusion and aggregation. For example, the Darkstar data warehouse [22] loads hundreds of data streams and maintains more than two thousand tables with real-time data loading and long-term histories. This data stream warehouse, which is built on top of DataDepot [7], supports networking research as well as real-time alerting and troubleshooting applications for AT&T network operations. As discussed above, alerting generally requires access to both real-time and historical data [43]; troubleshooting alerts requires seamless access to current, recent, and historical data. A similar tool is PRISM [29], which monitors the network for service disruptions due to maintenance activities. Darkstar has also supported long-term data mining studies, such as G-RCA [12] (Root Cause Analysis of network problems) and studies of IPTC set top box reliability [42].

While we are most familiar with networking applications, many other big data applications, some recently discussed in the popular press, have similar requirements.

- **Semiconductor manufacturing**: Tight control of the manufacturing processes and fast response to alerts is critical for modern semiconductor manufacturing. As reported in the popular literature [20], Applied Materials has developed a suite of tools [2] for monitoring and alerting of semiconductor manufacturing facilities. A 14nm fabrication facility is expected to generate 140+ TB per year from a disparate collection of sensor streams.
- **Industrial Internet**: More generally, large-scale manufacturing requires tight control of manufacturing processes and supply chains. General Electric has recently announced a focus on organizing and analyzing the data streams produced in an industrial setting [18].

We argue that large-scale data stream warehousing can be greatly improved by adjusting the notion of consistency that one expects

Conference'10, Month 1–2, 2010, City, State, Country.

from the warehouse. Instead of requiring some type of strong consistency, the system should just try to *make progress in the stream*. This principle applies to the base tables (which are sourced directly from the incoming streams) and also to the derived data products (i.e., materialized views). For example, see Figure 1. Base tables R and S are sourced from external streams, and table T is defined to be their join. When new raw data for S arrives, base table S can make progress (1). Since S has been extended, T can also be scheduled for an update extending its progress (2). As long as the system has adequate resources and continually advances a table to catch up with its sources, a user will be able to get a consistent view of the (hopefully recent) past.

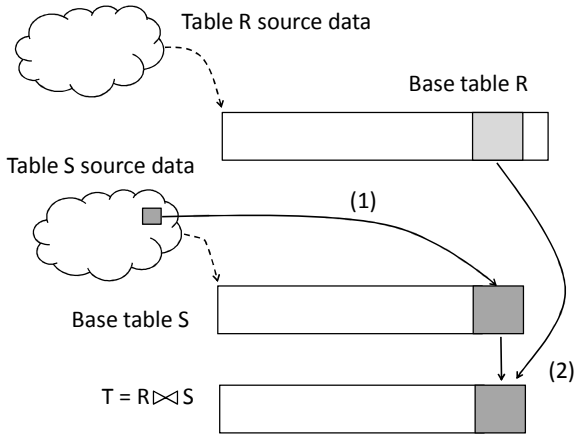

Figure 1. Making progress in the stream.

This approach to streaming data management is being promoted in the popular press, notably by Marz and Warren [30][31]. Marz and Warren argue that data stream analytics systems can be efficiently and readily built by abandoning strong consistency, and instead using a "make progress in the stream" approach. These authors suggest that large-scale stream warehousing systems can be constructed from open-source tools including Storm, Hadoop Cassandra and Hbase.

1.1 Tidalrace

We have been developing data streaming warehousing systems since 2005, and have built very large networking applications on top of them. One prominent result is the Darkstar [22] streaming data warehouse that is used to support networking research, network operations, and real-time network troubleshooting.

We have embarked on a project to develop a next-generation stream warehouse, *Tidalrace*, to support new and more demanding network monitoring and maintenance applications within AT&T Some of the more significant features and optimizations include

- Support for *temporal consistency*.
- Incremental in-the-past updates using partition *revisions*.
- Streaming updates to valid-time temporal tables.
- Partition re-organization
- Partition-wise optimization
- Distributed storage and execution

Our previous data stream warehousing system is DataDepot, which we have described in [13]. While DataDepot has had widespread application within AT&T, we felt compelled to develop a new system for two main reasons:

- DataDepot uses Daytona [17] as its underlying storage engine. Daytona requires a single address space (e.g. a mainframe or a POSIX-compliant shared file system), so scale-out is prohibitively expensive.
- We began development of DataDepot in 2005, before we had a full understanding of data stream warehouse semantics. Adding new features while maintaining backwards-compatibility with existing databases has become infeasible and the applications built on top of DataDepot are too large (2,000+ tables), complex, and mission-critical to risk re-writing them on the fly.

We also intend to use the opportunity provided by a fresh-slate approach to experiment with novel query processing techniques and optimizations. In this paper, we describe the organizing principles behind Tidalrace, how Tidalrace is structured, and its more significant features and optimizations.

1.2 Related work

In addition to DataDepot [13]and the works of Marz and Warren [30][31], several other projects have approached the problem of data stream warehouses.

One of the earliest descriptions of a stream warehouse is Moriae [3] which developed a history-enhanced event detection system. Moriae used a matching engine to determine when similar patterns occurred in the past to improve the system's ability to match current events. The Darkstar application Argus [43] uses stream history to identify network anomalies. An early proposal to support hybrid querying of live and archived streams is OSCAR [4] , built on top of TelegraphCQ. FastBit [35] takes a similar approach, using bitmap indices to accelerate queries. Related systems include latte [40], HYPE [33], and DejaVu [9] which has been implemented on top of MySQL and extensively tested.

Truviso [23] is a warehousing system that supports materialized views over continuously loaded data. We expand on their innovation of *revisions* into a more general mechanism.

The Tidalrace system described in this paper is based on our previous DataDepot [13] stream warehouse and our experiences with supporting Darkstar [22] and its applications.

These systems have been built on top of DBMS stores; however using a DBMS is not a requirement. Nova [32] is a system built on top of Hadoop and Pig to automate workflows. Nova will propagate delta updates from raw data sources to derived data products; the authors give an example of the workflow from RSS news feeds to a deduplicated set of articles. The workflow scheduling uses update triggers to propagate the deltas.

Our approach to data stream warehousing is similar in spirit to that of Marz and Warren [30][31]. They suggest an approach which uses cloud-friendly append-only files. Complex analytics can be supported by the equivalent of materialized views; new derived data product segments get computed when their source tables advance. Marz and Warren also argue that derived data products do not require extensive replication since their values can always be recomputed from the base data. These authors recognize the differences between *event* data and *condition* data

(as we discuss in Section 4.4), and propose an append-only mechanism for storing valid-time temporal tables.

Our approach to building a data stream warehouse has a similar philosophy, but with a greater emphasis on supplying semantics, especially those which enable users to make sense of temporarily inconsistent data, and to enable performance enhancing optimizations. Especially notable differences include:

- Explicit support and optimizations for late-arriving data, which is universal in large-scale data steam warehouses [13][23][28].
- Explicit support for *temporal consistency* (Section 5.3).
- A mechanism for streaming updates to valid-time temporal tables which supports efficient indexed access (Section 4.4).

2. Data Workflow

To motivate our approach, we describe a very simple network management example application, shown in Figure 2. Raw data flows from external sources to be loaded into base tables, which are then available for queries. In this case, the warehouse loads Twitter feeds, active round-trip-time measurements from probes in the network, and network link utilization data gathered via SNMP.

While this data is useful in its own right, usually we want to derive actionable information from the raw data. For example, from raw twitter feeds, we can apply textual analysis and derive a streaming table of customer complaints about network service. We can then combine customer complaints with the active measurement and link utilization feeds to derive a streaming table of service alerts. Troubleshooting the service alerts might require real-time data from the active measurement and link utilization tables. The service alerts can then be correlated with the customer complaint tables to derive knowledge about what types of network events have the most significant impact on customers. Other tables – the hourly and daily aggregates of link utilization – are similar to traditional OLAP tables and are used for long-range network planning. The tables which are populated with correlated and processed data are the *derived tables*, and are implemented as materialized views.

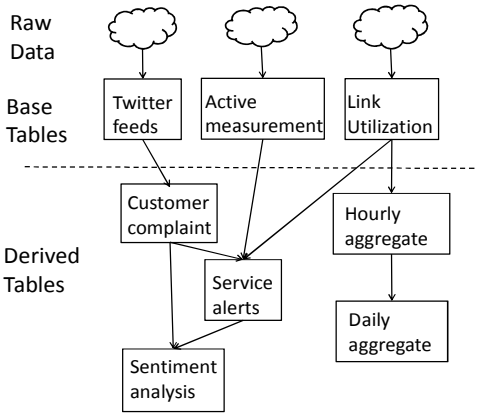

Figure 2. Simple example application.

This simple example of a streaming data warehouse illustrates several significant points. First, while real-time loading of data into base tables is important, the value of a data stream warehouse comes from its ability to correlate and process raw data into real-time actionable information. This principle can be seen in Nova [32], DataDepot [13], and the works of Marz and Warren [30][31]. Maintaining real-time derived information requires a mechanism that allows raw data to flow into the derived tables; the "update propagation" mechanism illustrated in Figure 1 has proven to be effective. Finally, some tables need to be updated as rapidly as possible (in this case the bases tables and the service alerts table), while others should be updated when the data sources are stable to enable long-term data mining (the Sentiment analysis and the aggregate tables).

3. Organizing Principles

Our approach to data stream warehousing is motivated by three main principles:

- Updates to existing data tables are prohibitively expensive. Even on centralized systems, random writes are far more expensive than sequential writes. Scalable (generally non-POSIX) distributed file systems generally support write-once or append-only files.
- In a large-scale distributed system, maintaining a current time-synchronized view of the system ranges from prohibitively expensive to infeasible[1]. Instead of requiring traditional strong consistency, we will just ensure that our tables continually make progress in the stream.
- A data stream warehouse must provide timely and eventually consistent data in order to be fully useful. However, in a large-scale stream warehouse, late-arriving data is universal [13][16][23][28].

To manage these conflicting concerns, one tends to be drawn to particular design decisions

Timestamp Partitioning: Since newly arriving data is the (generally) most recent data, a natural data organization is to use horizontal partitioning on a timestamp attribute. Ideally, all of the newly arrived data falls into a new partition. Even if old data arrives (i.e., having a value in the timestamp field that is less than the maximum of the timestamps of the existing records), the number of affected partitions is generally small. Furthermore, expiring obsolete data is simple, as the oldest data partitions can be simply dropped.

In general, every table is horizontally partitioned, using the timestamp field value for the primary partitioning predicate. Large tables may use additional partitioning predicates, and each horizontal partition may also be vertically partitioned (i.e. Section 4.1).

Write-once Files: In-place incremental updates are difficult to support in large-scale data systems for several reasons. For one, random access in disk storage (which is an order of magnitude cheaper and often more reliable than SSD storage [44]) is much slower than sequential access. For another, consistency in distributed storage is difficult and expensive to maintain. Distributed non-POSIX file systems such as HDFS only support write-once (or append-only) files. Finally, derived data products maintained as materialized views might not have a cheap and simple incremental update procedure – they might be e.g. machine learning models derived from an arbitrary R program.

[1] Spanner [5] uses a globally synchronized real-time clock for maintaining consistency, but is intended more for supporting sophisticated applications than for large-scale data mining.

If the data stream warehouse is structured to concentrate updates to a small collection of data partitions, then updates to base tables can be efficiently propagated to the derived data products which depend on these base tables. We have found (in the DataDepot project [13]) that the timestamp partitioning technique is very effective in localizing updates in most cases. DataDepot also uses write-once files for derived tables. Because of the efficiency and generality of write-once files, Tidalrace uses only write-once files for its data and indices (both base and derived tables).

Because of the prevalence of late-arriving data (and the frequent need for fast answers), small incremental updates to existing partitions are common. DataDepot would recompute these small partitions, which often introduces a significant inefficiency. In Section 5.2 we discuss a technique for overcoming this inefficiency that implements incremental updates using write-once files.

Loose Consistency: Traditional data warehouse operations place a high value on the internal consistency of its tables. However, the demands of real-time information from a data stream warehouse make the delays required for internal consistency untenable. Data stream management systems operate in real-time, but within very narrow time windows. A data stream warehouse loads disparate and widely-sourced data streams, with frequent late data, and cannot ensure operation within a narrow time window.

To ensure real-time response when needed, data must be loaded whenever it becomes available; high-priority derived data products might be computed well in advance of lower-priority tables. Transient errors in derived data products (due to inconsistency, incomplete data, or even incorrect source data) can be tolerated in many cases when the need for real-time information exceeds the need for consistency. However, these errors must be transient, labeled, and the user must understand and accept the implications of inconsistent data.

The temporal inconsistency inherent in the leading-edge of data stream warehouse tables generally makes the transactional commit of individual records an avoidable overhead. Because all correlated streams must be verified to be up-to-date, the batch-commit of update propagation is a better suited mechanism.

Update Propagation: We have argued that two traditional models for stream processing (data warehouse refresh and traditional data stream processing) are not suitable for data stream warehousing. Instead, we will simply try to advance the base tables (and transitively, the derived tables) to catch up to the source data streams. Each advancement step propagates updates in a target table's source(s) to the state of the target table, as illustrated in Figure 1. Updates can be performed in a localized, and therefore readily distributed, manner. We have found that data stream warehouse refresh via update propagation to be efficient and reliable [13]. While a naïve update propagation algorithm can suffer from the missing-update problem, we have published simple and provably correct update propagation algorithms in our previous work [21].

Multi-Version Concurrency Control: The need to ensure real-time response recommends the use of Multi-Version Concurrency Control (MVCC). Long-running queries do not block updates, updates do not block queries, and expensive updates do not block updates to their source tables. In previous work, we identified the source of state data in a real-time table to be the need to block updates to the real-time table to compute an hourly aggregate summary [21].

In a single-writer scenario, MVCC can be inexpensively implemented [34]. When using write-once files, the implementation becomes simpler still [13]. We note that single-writer does not mean that the update of a table must be single threaded, or even restricted to a single server, but rather that the computation have a single control and commit point.

Temporal Consistency: Continuous data loading generally entails a significant degree of uncertainty about whether or not all of the data for a given time period has arrived, or will arrive. The traditional data warehouse approach is to wait for a time interval to pass, after which it declares that all data has arrived and performs a batch load. However a) for many alerting and trouble-shooting applications, having the most up-to-date data is important enough that some data inconsistencies can be tolerated, and b) different streams have different arrival latencies, and need to be treated differently – one waiting interval does not fit all streams. A short delay time (e.g. one hour) is likely to return inconsistent results; while a long delay time (e.g. one day) does not provide real-time answers.

A data stream warehouse needs keep track of the "temporal consistency" of the streaming tables that it maintains [16]. Starting at the base tables, the system tracks the arrivals of new data and determines the completion status of each partition. The temporal consistency of the base table gets propagated to the dependent materialized views. By maintaining and supplying temporal consistency information of the tables in the stream warehouse, a data stream warehouse can maintain its tables using a loose consistency model and still provide consistency guarantees to users. We return to this topic in Section 5.3.

Temporal Description Tables: Streaming data is often thought of as consisting of a stream of *events* – measurements that occur at a specific time or during a short and well-defined time period. Event data generally needs *description* data to supply the necessary context for proper interpretation. Description data describes conditions that last for a long time and are of an uncertain duration. For example, an event data stream might consist of temperature measurements from sensors in a machine room. The temperature sensors themselves do not provide much meaningful information; they need to be correlated with a description table which specifies where each sensor is located.

The need to provide context for event streams has led many data stream management systems to allow joins to relational data. However, the relational tables are generally *snapshot* tables. A data stream warehouse cannot use snapshot tables for its description data because 1) the description tables change slowly but steadily over time, and 2) a data stream warehouse must often deal with in-the-past joins: late arriving data, catch-up on blocked streams, reloads of problem data. All description tables must be valid-time *temporal* tables [24]. Further, description tables receive streaming updates (e.g., a temperature sensor gets moved) and therefore must be maintained with timestamp-based partitioning in a manner similar to that of event tables. We return to this topic in Section 4.4

4. Tidalrace Architecture

A data stream warehouse generally does not need a specific underlying database architecture – systems have been layered on top of the Daytona [13] and Postgres [23] DBMSs and on top of Hadoop/Pig [32]. We decided to take the opportunity of the full system redesign to develop a data stream warehousing system which is specialized to our needs.

One issue that we needed to address was support for both distributed and non-distributed installations. Our experience with the GS Tool [7] showed that an efficient and well-tuned DSMS can processes petabytes per day in a single 2U server. A small-scale high-performance data stream warehouse system is essential for supporting network operations at the "edge" of the network. However, global network operations require very large scale warehouses, necessitating scalable storage and computation. We therefore needed to develop a system which works well both as a small-scale single-server installation and as a large-scale distributed system.

A second issue we faced is the need for multi-language support in the definition of materialized views. While a large fraction of the derived data products we needed to support are readily described by SQL, others are not. For example, many networking analyses require state-machine processing [16]. Other derived data products might be created using statistical analysis tools such R.

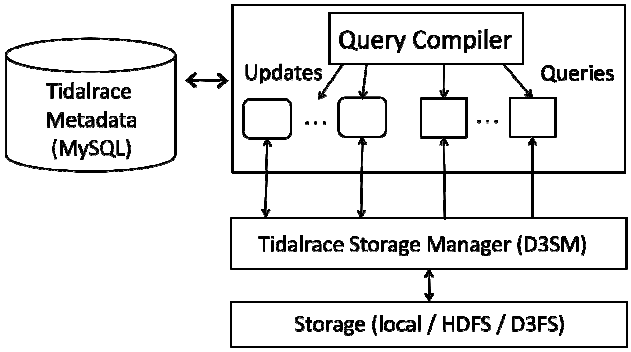

Figure 3. Tidalrace Architecture

Figure 3 shows a top-level view of the Tidalrace architecture. The state of the system – the data dictionary, the location and status of the data in index files, etc. – is stored in the Tidalrace metadata database. The metadata database is used to perform transactional commits for update programs, and therefore must be a transactional (ideally ACID) database. We use MySQL for this function, though other DBMSs can be used. The metadata database is the only transactional store in Tidalrace, and is small compared to the size of the warehoused streams. Our experience with managing very large data stream warehouses with DataDepot [13] has shown that a metadata database of a few megabytes can manage a warehouse of many terabytes and thousands of tables, with an update rate which rarely exceeds a few update transactions per second. The metadata database is the only transactional component of Tidalrace, so transactional storage and synchronization does not present a scaling bottleneck.

Tidalrace updates its tables via a batch *update propagation* mechanism. Data sources generally supply data in *packages* of new records; these packages arrive periodically and contain collections of records. At the base tables, an update determines which packages haven't yet been loaded into the base table, upacks the records in the packages, and loads them into the proper base table partitions. When the update is complete, its effects are committed by recording the progress in the source stream and the new data in the base table via a transactional commit to the metadata database. For derived tables, an update program determines what parts of the derived table require an update, computes the new value of the updated partitions, and records the new partitions of the derived table via a transactional commit to the metadata database. In both cases, partially executed updates have no effect on the state of the warehouse.

Queries (including the computation part of a derived table update) read the state of the tables that they access at the start of the query, providing read isolation. The effect is to implement a type of single-writer multiversion concurrency control [13]. Tables can be queried and updated concurrently, and a source table can be updated concurrently with an update of a dependent derived table.

An *update scheduler* component schedules updates to base and derived tables. Updates can be periodic or triggered (by the arrival of new packages or the update of a source table). Ensuring that high-priority tables are kept current while scheduling all updates and avoiding resource overutilization is a topic of continuing research [14].

Our experiences with the GS Tool [7], Daytona [17], and DataDepot [13] have shown that a compiled-query system offers very high performance while still enabling interactive ad-hoc queries. We have developed a SQL compiler for Tidalrace, which compiles queries in less than 2 seconds. In addition to enabling user access to Tidalrace-resident data, the query compiler is also the primary mechanism used to define materialized views. The result of a compiled view definition is a table definition in the metadata database, and also a program which updates the materialized view.

The query and the view update programs access Tidalrace-managed data using the Tidalrace storage manager, *D3SM*. The storage manager provides indexed and non-indexed access to data partitions, hiding details of the data format and storage location. Section 6.1 discusses D3SM in greater detail.

Tidalrace provides a collection of libraries that interface with the metadata database, and which perform functions that enable update propagation and transactional commit of updated or newly created data partitions. Therefore, there is no requirement that materialized view update programs be generated using our query compiler. As in DataDepot [13], we can instead generate a wrapper update program that fetches data from Tidalrace, presents the data to an external view computation program, then loads and commits the result into Tidalrace. Materialized view definition is much easier using a built-in view definition language, but the plug-in capability of an external-view wrapper provides what is sometimes a critical flexibility.

4.1 Data Model

Each table in Tidalrace contains a collection of records; each record contains a collection of fields. A field can be an atomic type (integer, date, string, etc.) or a structured type (range, list, map, etc.). Access to structured types depends on the presence of serialization/deserialization methods for transmission to/from D3SM, and the ability of the query language to process the type.

Every table in Tidalrace must have a *timestamp* field. The name and data type of the timestamp field is an immutable property of a Tidalrace table. The timestamp field is used for the primary (horizontal) partitioning predicate of a table, and refers to the Unix timestamp of the event occurrence time (or is a pair of Unix timestamps that define the valid time of a description – see Section 4.4). The timestamps allowed by Tidalrace are more restrictive than those allowed by DataDepot, but our experience with very large stream data warehouses has shown that every table among the thousands we have encountered can be readily modeled with Unix timestamps - with a considerable savings in logical complexity. The Unix timestamp can be represented with a variety of data types – integer, date_time, date, or float, but are converted to 64-bit integers for internal use.

A table's timestamp is used for its primary partitioning dimension. Each partition is labeled with a timestamp range [pl, ph). An event record r with timestamp ts is assigned to partition p if pl ≤ ts < ph (partitioning of description tables is discussed in Section 4.4). A large table can have additional orthogonal dimensions of partitioning.

4.2 Base Tables

Raw data continually arrives, most often as a collection of files (packages) transmitted from a data source. Raw data is periodically loaded into a *base* table – the period can be very small, e.g. 1 second. Base table loading performs two important functions. First, it extracts and transforms the raw data. For example, the source data might be compressed and in a csv format. Base tables apply simple transformations to the source data, converting fields into internal binary representations. Second, raw records get collated into base table partitions according to their timestamp field. Ideally, most new records are collated into new partitions; however some will collate into existing partitions. Base tables are select-project views over the raw data, and they have simple incremental update programs: append the new records. Tidalrace uses write-once files, so existing files can't be appended; instead we create add-on (*revision*) partitions to store the additional records. These revision partitions are linked to the original partition, as described further in Section 5.2.

4.3 Update Propagation

Tidalrace supports deep and complex DAGs of materialized views. An essential service of a data stream warehouse is to propagate updates from the base tables to all (immediately or transitively) dependent derived tables. Furthermore, we need to minimize the amount of recomputation at the dependent derived tables caused by an update to a base table. Tidalrace uses the *source vector* protocol [21] for update propagation, which we summarize below.

Each derived table is dependent on one or more *source* tables, which might be base or derived tables. Metadata associated with each derived table states the timestamp range of a source records that can affect a derived table record with a particular timestamp. Let D be the derived table, S be the source table, and D.ts, S.ts be their timestamps. Tidalrace uses this pair of bounding functions to determine the timestamp range:

$$D.ts \geq m\left\lfloor \frac{S.ts + a_l}{m} \right\rfloor + b_l$$

$$D.ts \leq m\left\lfloor \frac{S.ts + a_h}{m} \right\rfloor + b_h$$

The parameter *m* allows for the expression of discrete ranges (e.g. encountered in aggregation), while parameters *a* and *b* allow for timestamp variation and band joins. When a partition of S is updated, the affected partitions of D must be updated. These can be determined by inverting the bounding functions to determine the timestamp range, (and thus the collection of derived table partitions) which can be affected by data in the source table partition.

Another piece of table metadata is its *generation*, which increments each time the table is updated. Each table partition is labeled with the generation is which it was updated. Derived table partitions are also labeled with a compact representation of the generation(s) of the source table partitions they were computed from. The source-vector protocol uses this information to determine if the source data for a derived table partition is more recent (further along in the stream) than the derived table partition, indicating that the derived table partition should be updated.

By using the source-vector protocol and comparing all source partitions to the derived table partitions, the update propagation program can determine all derived table partitions that need updating. The decision of which derived table partitions to update (usually, all) is left to the scheduler.

4.4 Description Tables

As discussed in Section 3, a data stream warehouse ingests two types of streams: *event* streams, which refer to observations at a particular point in time or measurements over a small and well-defined time interval, and *description* streams, which refer to conditions that hold over long and indeterminate periods of time.

For example, a data stream warehouse for networking applications might receive a stream of SNMP measurements of the number of bytes transmitted over a network link during a 5-minute interval. While this information has value, to be truly useful to the analyst these records should be correlated with a database that describes the transmission speed of these links. Suppose that a link has transmitted 200 Gbits during the last 5 minutes. If the link speed is 1 Gbit/sec, then the link utilization is 67% (close to saturation), while if the link speed is 10 Gbit/sec, then the link utilization is 7% (low).

Description tables change slowly, so they are often modeled as snapshot tables, e.g. [1]. However we have observed in a sampling of description tables that a significant number of rows (1% to 5%) are modified each day. A data stream warehouse must often perform in-the-past correlations: ad-hoc queries, late-arriving data, catch-up processing on delayed streams, and loading new tables with an e.g. 2-week initial data load. Therefore description tables must be stored as *valid-time* temporal tables [24].

Conventionally, a valid-time temporal table is stored as a RDBMS table which receives continual updates which are processed in-place. Let description table D have fields (K, I, V) where K is an entity key (e.g. a customer ID), I is a valid time interval [tl, th), tl < th, and V is the value associated with key K during interval I.

Suppose that D is updated with (k,v,ts), where k is a new key, v its value, and t1 indicates that k received value v at time t1. Then (k, [t1,∞),v) is inserted into D. Next suppose that D is updated with (k,v',t2). Then the original record is modified to be (k,[t1,t2),v) and (k,[t2,∞),v') is inserted into D.

Modifying existing records does not fit well with our write-once architecture. We make use of a controlled degree of duplication to enable streaming updates to valid-time temporal tables.

Recall that each data partition in Tidalrace is marked with a timestamp range [pl, ph); records with a timestamp in this range are stored in this partition. Description tables have a time interval for their timestamp; a record with interval [tl, th) is stored in a partition with timestamp range [pl,ph) if [tl,th) intersects [pl,ph). A record in partition p with timestamp range [pl, ph) lies only [pl, ph). When performing an interval-stabbing join (i.e., given (k, t) return the v (if any) such that there is a record (k,[tl, th),v) where tl ≤ t < th), search the partition p with timestamp range [pl, ph) such that pl ≤ t < ph.

This type of data organization has several benefits: new updates do not affect old partitions, description tables are managed in the same way that event tables are, and old data is easily expired by dropping partitions. However, the benefit comes with a significant cost – a space blowup in the storage of the description

table. If we cut a new description table partition once per day (e.g. we might receive a new current-snapshot once a day), then storing a 2-year history requires a 730X space blowup.

We have found that description tables are generally much smaller than event tables, so some degree of duplication is acceptable. Furthermore, we can merge old and stable description table partitions into partitions which span a much larger interval. We have already implemented this type of partition merging in DataDepot [13]; its implementation in Tidalrace is part of our partition maintenance procedures, as described in Section 5.1. A 730X space blowup can easily be brought down to a manageable 10X blowup, depending on how fast the source data becomes stable (see Section 5.3). Finally, small and incremental updates can be handles with partition revisions (see Section 5.2).

4.5 Table Segments

Our experience in managing large data stream warehouses has shown that schema change is a continual headache. A common practice in data warehouse management is to include "dummy" fields in a large table. These fields can be renamed as new fields are added to the table, avoiding a massive restructuring operation for the historical portion of a large table. We make use of the highly temporal nature of a streaming table to enable dynamic schema change.

In Tidalrace, a table is partitioned into *segments*. The invariant properties of a segment are the fields (names and data types), indices, and defining query or program (for derived tables). The only invariant property of a table is its name and its timestamp field – which must exist in every segment. Each segment has a timestamp interval [sl, sh) which defines the region of validity for a segment: each partition in a segment has a timestamp interval [pl, ph) which is contained in [sl, sh), and the collection of segment timestamp intervals partitions the table's data window.

The primary purposes of a segment are to describe the available fields in the segment's partition, and to describe how to access missing fields (either treating them as NULL values or supplying default values). For example, suppose that in table CustomerBrowseHistory, in segment 1 the fields are (string Customer, string Url, int ts). After gathering data for an long period, the analysts want to add an additional field string Location. The location is added for segment 2, with a corresponding change in the defining query.

Next, the analyst poses an aggregation query, grouping by Customer and Location. This query can cross segment boundaries by supplying a default value (e.g. the empty string or "Not Available"), as is common practice.

Changes in indices can also cause expensive restructurings of existing tables. By specializing segments with the set of supported indices, indices can be added and dropped without restructuring, and queries can still seamlessly cross segment boundaries. The query plan will in general need to be specialized to the data segment – a access plan index available in segment 2 but not segment 1 is valid only for segment 2; a different plan is required for segment 1. Multiple access plans are supported by Tidalrace's partition set processing, as described in Section 5.5.

5. Optimizations

The efficient functioning of Tidalrace relies on a collection of optimizations, which are enabled by the basic Tidalrace architecture. We describe some of these optimizations in this section.

5.1 Partition Reorganization

Data warehouses that ingest real-time data feeds and archive them for long time windows face a tension in the methods used to store the data. Fast updates are best served by write-optimized storage: row-oriented, small uncompressed partitions. However the maintenance of a very large warehouse can be best served by read-optimized or storage-optimized formats. A technique that has been used is to transform the storage format of a data partition when enough time has passed that the data has become stable [13][26][36].

The use of file-oriented single-writer multi-version concurrency control (see Section 4) allows the reorganization of stable partitions to occur as a background task (a partition update conflicts with a partition reorganization, so the reorganized partition must be stable) with no impact on data loading or querying. Data reorganization tasks can be scheduled as low-priority tasks that execute as time permits and during off-peak hours. The types of table reorganization that can be performed include:

- Partition merging. Frequently updated tables are best served by small partitions to minimize partition rebuilding and index construction due to updates. However, opening many small partitions is inefficient for queries, and a very large number of partitions places a burden on the metadata database. The space overhead of description tables (see Section 4.4) can be greatly reduced by merging stable partitions.
- Column-oriented storage: While row-oriented storage allows for fast updates, column-oriented storage can offer many benefits, most notably reduced I/O cost when only a few of many fields are accessed, and the potential for aggressive compression.
- Compression: While fast updates are best supported by uncompressed storage, stable older data partitions should be compressed to reduce storage and I/O transfer costs. Old archival data partitions which are infrequently queried can be aggressively compressed.
- Storage hierarchy: New data is generally the most frequently accessed, and should be stored high in the memory hierarchy (RAM disk or SSD) while older data can be migrated to disk storage.

5.2 Revision Partitions

A key distinction between a DSMS and a data stream warehouse is the need to load and propagate late-arriving data [13]. The primary mechanism in Tidalrace and related systems [23] for dealing with late-arriving data is to recompute the affected partitions of derived tables. However, partition recomputation can become prohibitively expensive for large tables which experience frequent late arrivals.

A method for the efficient processing of late-arriving data was proposed for the Truviso system [23], which we call *partition revisions*. The idea is to process and summarize updates to the source data, to allow query-time correction of stored partitions.

With an abuse of notation, let D be a derived table defined by query Q over source(s) S, i.e. D=Q(S). Suppose that there are programs P and R such that

$$Q(S+\Delta S) = P(S,R(\Delta S)).$$

Then alongside S we can store R(ΔS) and combine S and R(ΔS) using P at query time. If R(ΔS) is small and P is fast, then storing the *revision* R(ΔS) is more efficient than recomputing D.

When phrased in terms of table partitions, an *anchor partition* is a regular partition, computed using query Q. A *revision partition* is computed using R. In the metadata database, partition revisions inherit the anchor partition's properties (e.g., timestamp range, source vector [21], etc.), but are marked as being revisions, and have their own generation metadata (see Section 4.3). The collection of revisions to an anchor can be thought of as being chained to the anchor, as illustrated in Figure 4.

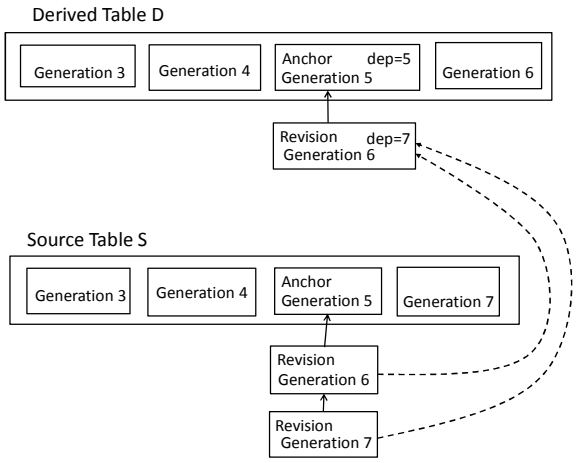

Figure 4. Partition anchors and revisions

In [23], the only tables that can have revision chains are aggregation tables which reference associative aggregates only. The programs P and R are derived using the well-known sub-aggregate and super-aggregate technique [27]. However, many defining queries have simple P and R programs – for example selection/projection queries. In Section 4.2, we describe how updates to existing base table partitions use the revision technique. Base tables are a projection view of the raw data: the program R is projection query Q and program P takes the union of the anchor and the revision.

By representing updates to existing base table partitions using the anchor/revision technique, we represent the delta to the base table in a readily accessible form. In Figure 4, the revisions in the source table S are explicitly tracked. After the 5th update to D (creating the anchor with generation 5), the source S has two additional updates, creating revisions with generations 6 and 7, respectively. The next time that D is updated, the update protocol can observe from the anchor partition's metadata that it has all data up to generation 5 of S. The corresponding anchor partition of S has generation 5, but also has two revisions with generation 6 and 7. Instead of recomputing the anchor partition of D, we can compute a revision using program R on the generation 6 and generation 7 revisions – creating a generation 6 revision to the anchor. The anchor and its revision now contain data up to revision 7 of the source S. This explicit representation of deltas allows up to chain the revision technique arbitrarily far up the derived table dependence DAG, subject only to the defining queries having suitable P and R programs.

In addition to selection/projection and aggregation queries, we can define inexpensive P and R programs for the queries which maintain the valid-time description tables, and for outer-join queries in which the outer join is a foreign-key to primary-key join – which are commonly used to join event data to description data. In fact, the need to support incremental updates to description tables has been a major motivation for developing revision partitions in Tidalrace, given their prevalence in data stream warehouses.

The need to maintain revisions and to apply program P at query time is an overhead which is acceptable for the active portion of a table, but can which can be eliminated for stable data. Revision flattening is one of the partition reorganization programs we run as a maintenance procedure (see Section 5.1).

5.3 Temporal Consistency

In our development of data stream warehouses, we argue that one cannot expect to obtain a consistent "now" snapshot in time of the state of the system. Ignoring data quality issues for the moment, one can only hope to obtain a snapshot view of the system up to sometime in the recent past. A major service of a data stream warehouse is to inform the users of what parts of the database (e.g. how recent) are suitable for their analyses. Furthermore, some analyses need data which is more solid than others. Troubleshooting queries will generally demand the most recent data possible, while data mining queries generally require high-quality time-synchronized data. Alerting queries fall in between.

In our discussions of warehouse maintenance procedures, we have often used phrases along the lines of, "when the partition is stable", for example in the context of partition reorganization in Section 5.1. The system needs a mechanism for determining or imputing stability – meaning that a partition is not expected to be updated ever again. The meaning of stability for data maintenance purposed is related to the meaning of suitability of data for data mining.

Some derived tables perform expensive computations to compute partition values – large-scale aggregation, data mining, etc. It is generally desirable to wait until the source data is stable before triggering updates to partitions in expensive derived tables. If the defining query does not have efficient P and R programs, one cannot use the partition revision technique (see Section 5.2), so the ad-hoc technique of "wait until 5 minutes after the hour to compute hourly summaries" still leads to frequent expensive computation due to late arriving source data.

In a large scale data stream warehouse that ingests hundreds of distinct data feeds obtained from worldwide sources, no single rule can be applied to judge the stability of each of the data sources. Data streams tend to have individual periodicities (e.g. 5-minute measurements vs. daily dumps), different latencies (data arrives 1 minute late vs. 1 hour late) and different degrees of disorder.

Contrary to the first paragraph, many analyses cannot ignore issues of data quality, especially issues of data completeness. In some cases one can estimate how much data *should* arrive during a time period, and mark a partition according to how complete the partition seems to be.

Determining stability and completeness at a base partition can be challenging; determining stability and completeness at derived table partitions computed from correlations over many tables is far more challenging. We have previously published a framework for tracking and imputing temporal consistency [16], which we summarize here.

In a data streaming system, *punctuations* [39] are used to track progress and enable out-of-order processing [28]. Our mechanism for tracking temporal consistency adapts punctuations to a data stream warehouse setting. Instead of applying to the stream as a whole, we annotate partitions with *consistency markers*. A simple collection of consistency markers is

- *Open* : source data exists for the partition.
- *Closed* : no new data will arrive for the partition.
- *Complete* : Closed, and all expected data has arrived.

Consistency markers are assigned to base table partitions based on imputation rules which must be tailored for the base table. This simple collection of markers has simple imputation rules for derived table partitions: Open if some source is Open, Closed (Complete) if all sources are Closed (Complete).

A far richer collection of markers is generally required. For example one might define *WeaklyClosed* and *StronglyClosed* markers to support different types of users. Our previous work [16] has more details on the topic.

Tidalrace metadata about partitions includes the collection of temporal consistency markers. We are currently engaged in researching imputation rules at the base tables using the data streams we ingest.

5.4 Distributed Storage and Queries

When used in a distributed setting, Tidalrace replicates and distributes partitions to participating servers in the cluster. When a user submits a query, data partitions can be fetched from the distributed file system to a local file system for local evaluation. However, it is easy to observe that a distributed query system can be made more efficient by shipping subqueries to the servers that contain the data partitions, which return query results that are combined for the return result. For example, an aggregation query can be broken into subaggregate queries sent to the remote servers, with a superaggregate query combining the results. The D3FS component of Tidalrace provides a facility for remote subquery execution.

Join-free queries can be efficiently implemented using remote subqueries, but join queries still require data transfers to bring together records correlated from different range variables. We can make two observations. First, in a data warehousing system, a large amount of the query workload is generated by the queries used to update derived tables. This workload is (relatively) constant, and can be analyzed and used to optimize update performance [11] – in particular, which tables are often joined together. Additional information can be collected from query logs. Second, we have observed that joins are almost always band joins; data from similar time periods are usually more relevant than data from distant time periods.

Since data from similar time periods tends to be correlated with each other, one natural way to distribute data is by hashing on the timestamp of the data. For example, data with a timestamp in [1:00, 1:05) is assigned to servers 1, 6, and 11; [1:05, 1:10) to server 2, 6, and 12, and so on. However, this assignment scheme concentrates work on a few servers during their active time period. Other issues arise: different data sets have different natural periodicities and partition sizes. A high volume event data stream of e.g. web clicks might be best partitioned into 1-minute segments, while a stream of 15-minute measurements has a natural periodicity of 15 minutes.

We make use of *co-location schemes* to co-locate oft-joined data. A co-location scheme is a combination of an identifier (e.g XYZ), and a sequence number for the identifier; so XYZ(5) is the 5th instance of co-location scheme XYZ. For every table T that uses co-location scheme XYZ, there is a mapping from sequence number *s* to a timestamp range [*tlo, thi*). This mapping is generally a function of the form *tlo(s) = a*s+b*, with a corresponding function for *thi*. A co-location identifier is used the key to hash to a collection of storage servers. If a partition with timestamp range [*plo, phi*) overlaps the timestamp range of its co-location scheme with sequence number *s*, the partition is stored at the corresponding set of servers.

A table can be associated with multiple co-location schemes, which increases its degree of replication but also its availability for local joins. We are developing co-location scheme optimization algorithms as part of our on-going research. Related projects include CoHadoop [10], which co-located HDFS-resident data; however it does not have the mechanism of co-location schemes.

5.5 Partition-wise Optimization

As we have discussed, joins in a data stream warehouse are almost always band joins on the table's timestamp. Since the primary partitioning dimension is on the timestamp, a natural way to process queries is to partition the work to be done on the table timestamps. This type of partitioning fits naturally with data distribution through co-location schemes, and with the variant processing that can be required by table segments (Section 4.5).

Suppose that query Q joins tables T_1 through T_n. A *partition set* is a collection of partitions PS = ($\{P_1\},\ldots,\{P_n\}$), where P_i is a set of partitions of table T_i. The partitions in a partition set are joined in one unit of processing. If the join has partition sets (PS_1, … PS_j), the result of the join is the union of the individual partition set joins.

Suppose that the optimizer has determined a left-deep join order, e.g. T_1, …, T_n. Suppose further that the optimizer has analyzed the band-join predicates in the query to compute a collection of bounding functions between the timestamps of each table (e.g. similar to the bounding functions in 4.3). A collection of partition sets can be found by

1. Determining the collection of partitions referenced in T1, P1.
2. Divide P1 into $\{P_{1,1},\ldots,P_{1,m}\}$.
3. For each j from 1 though m
   a. For each T_k from k=2 through n
      i. Use the bounding functions from T_k to T_1 .. T_{k-1} to determine timestamp band for T_k, and therefore $P_{j,k}$

There are a variety of interactions between the optimizer and the selection of partition sets – duplicate access to partitions, minimizing memory use of hash tables, etc. Recent work on incorporating horizontal partitioning into a cost based optimizer includes [19].

6. System Components

As shown in Figure 3, the principle components of Tidalrace are the metadata database, the query and update system, the storage manager D3SM, and the distributed file system D3FS. Aspects of the metadata database and the query system have been discussed elsewhere; in this section we discuss D3SM and D3FS.

6.1 D3SM

The Tidalrace storage manager, D3SM, provides an insulating layer between the query system and the actual data storage. A data partition might be in local or distributed storage; might be stored in a row-oriented or a column-oriented format, and might be compressed. The D3SM API requires that the query system specify the fields and indices to be accessed, the storage type (local or distributed) and the record layout (row-oriented or column-oriented). D3SM will fetch (in the case of distributed storage) and open the requisite files and thereafter present a uniform API for index and record access. By requiring that the

fields and indices be specified before partition access time, D3SM can fetch and open a minimum number of files.

6.2 D3FS

D3FS is a distributed object store, based on the architecture of Amazon Dynamo [8], tailored for storing large objects using co-location hints (see Section 5.4). Given its Dynamo heritage, D3FS is a pure peer-to-peer distributed system, which means that all nodes are created equal and that there is no single point of failure. Hence, D3FS does not use a meta-server for storing object placements. Instead, D3FS uses consistent hashing (i.e., a ring) to maintain node cluster membership and to efficiently decide object placement and repair strategies, in a decentralized fashion.

D3FS uses Apache Zookeeper as the foundation of cluster membership management. Each node that wishes to participate in a D3FS instance, first registers with Zookeeper. Zookeeper announces the new node among all existing nodes, and all nodes (including the new node) update their local instance of a structure called the cluster-map. The cluster-map is a ring structure, divided into a pre-defined number of disjoint partitions. The ring length is equal to the domain of a pre-defined hash function (e.g., the first four bytes of MD5SUM), that hashes strings to a number in [0, 2^32). Then, each partition of the ring is an interval from the domain of the chosen hash function. Each node is hashed a large number of times (e.g., 100 times) and added in the ring. So the ring is composed of a fixed number of partitions and a sequence of hash values, each hash value associated with a particular node.

When a new node is added to the cluster, a new set of hash values (corresponding to the new node) are added to the existing sequence of hash values. When an existing node is removed from the cluster (e.g., when a node fails) the set of hash values corresponding to that node is removed from the existing sequence of hash values. Notice that each time a change occurs in the cluster, each node is eventually notified by Zookeeper regarding which node entered or left the cluster, and can update its cluster-map accordingly. Given that we hash each node a large number of times, we expect the hash value sequence to be uniformly distributed across the ring.

D3FS uses data replication for reliability and availability purposes. By default, each object is replicated three times (this is a configurable, system wide parameter). Determining where to place a particular object is simple. Each object is associated with a key (e.g., the key can be a string) that uniquely identifies that object. D3FS hashes the key of the object and produces a hash value that is contained in one of the ring partitions. Let object O be contained in partition P. If we want to replicate O three times, we choose three nodes to be responsible for partition P. A simple way to choose three nodes is to consider the first three hash values in the cluster-map that come right after the hash value corresponding to the upper bound of partition P (if the first three hash values happen to correspond to duplicate nodes, we keep examining hash values in order until we get three distinct nodes). Assuming that all nodes have the same view of the cluster-map, each node will independently make the exact same placement decision for object O. Short-lived differences in how nodes view the state of the cluster are easy to take care of, since an incorrect initial placement can be made, and the nodes will eventually start repairing missing replicas, once the cluster-map reaches a steady-state.

D3FS is tailored for storing large objects because it does not use LSM-trees. Instead, objects are stored as plain binary files on the underlying file system of each node. When a client issues a request to store an object, D3FS initializes the file that will be storing the object on all the nodes responsible for that object. The initialization process is done using a two-phase commit. A coordinator node requests from all responsible nodes to initialize the file, and if enough successful replies are received, the coordinator signals the client to start streaming the data (D3FS supports the concept of success thresholds, where a store operation can succeed even if not all the nodes responsible for an object have successfully stored that object; inconsistencies are corrected in the repair phase). After the client has appended the whole object, the coordinator commits the object using one more round of two-phase commits with all destination nodes. Finally, the client is notified with success or failure. Two-phase commits happen in parallel for all nodes responsible for a particular store operation. Data appends happen in a pipeline (i.e., chain replication [41]), where the coordinator determines a random order for the destination nodes and then the data is appended in a chain.

D3FS does not support mutable objects. In fact, no deletions can happen in D3FS. If an object associated with a particular key has been stored in the system and a client tries to store a new version of that object using the same key, the store operation will fail (destination nodes will refuse to initialize a store request if they already store an object with the same key, locally). A failure scenario can occur with mutable objects if a client stores one version of object O, then all nodes responsible for O leave the cluster, the client stores a new version O' using the same key, and then some of the nodes storing O re-enter the cluster. Now D3FS would have two divergent versions of the same object.

D3FS has been designed so that clients can move computation where the data is actually stored. All objects are stored as regular binary files on the nodes. Given the key of an object, D3FS will return the location of that object and the corresponding canonical path on the local file system. Since files in D3FS are immutable, it is safe to allow clients to have read-only access to those files directly. For example, we can ship query fragments to a host which stores a particular data partition. This feature is particularly useful for executing complex joins between large tables, if all tables (or partitions thereof) are located on the same node.

D3FS allows clients to specify co-location hints (see Section 5.4) for storing related objects at exactly the same set of servers. This is accomplished by using a hierarchy of keys, i.e., a location key and object key. The location key is used for deciding the placement of an object and the object key is used as the file name of the file storing the object. In order to place two or more objects at the same location, we simply have to store them using the same location key. Then, given one of the nodes responsible for storing those objects we can read each object using the object keys. Clearly, an object can be associated with multiple location keys. In that case, the object might be replicated more times than the pre-defined replication factor. If an object is not part of a co-located placement, then the location key is made equal to the object key (which is a unique location key by definition).

Repair in D3FS happens every time there is a failed request to read an object from a particular node and every time the cluster-map is updated. In the first case, a client hashes the object key, determines the nodes responsible for storing the corresponding object, chooses one of those nodes, and the node replies that it does not actually store the given object. The client retries the request using another node, but the node acknowledging the failure tries to repair the missing replica. This scenario can occur if a node becomes responsible for the partition containing that

missing object after the object is written. This situation can occur if after the store operation, one of the nodes responsible for the object's partition left the cluster, or a new node entered the cluster and one of its corresponding hash values got injected in-between the hash values of previously responsible nodes, or during the store operation the node failed to store the object, but the store operation proceeded due to a reduced success threshold. Repair in that case is straightforward, since the node missing the object can simply request it and replicate it directly from the other nodes currently responsible for that object.

In the second case, repair is triggered by all nodes whose partition membership has changed, due to a node entering or leaving the cluster. Since the order of hash values in the cluster-map sequence changes, many nodes will become responsible for partitions they were not responsible before, and many nodes will have to relinquish partitions to other nodes. Clearly, finding new partition memberships can be done easily by taking the difference between the old and new hash value sequences. Once each node has determined the partitions it became responsible for, it contacts the other nodes responsible for those partition and request a list of objects that it needs to replicate, and starts replicating all missing objects. The node also becomes responsible for all new objects associated with that partition. When a node needs to relinquish a given partition, it waits to hear from the new node responsible for that partition, and then sends back a list of all objects contained therein. Once each object has been successfully replicated at the new node, the old node can delete the local replica.

Notice that a transient failure, where a node leaves the cluster and re-enters it a short amount of time later, does not pose a problem. During the time that the node has left the cluster, a new node will become responsible for all new objects stored in the affected partition and also start replicating all old objects. Once the failed node comes back, the new node will abort all pending replications. Then the old node will receive from all other nodes in the affected partition an updated list of objects belonging to that partition and it will start replicating all missing objects. Store operations that are in flight are first committed, and then repaired accordingly.

7. Implementation Status

We have implemented an initial version of Tidalrace with initial versions of the features described in this paper, except for remote query execution. Preliminary performance results are encouraging: data loading at 100k records per second per core, and aggregation at 500k records per second per core. D3FS exhibits performance that is as fast as HDFS. We are currently integrating Tidalrace into Darkstar for a next-generation data warehousing cluster, and also into other new network monitoring projects.

8. Conclusions

We are developing Tidalrace, a next-generation data stream warehousing system. Our efforts in data stream warehousing were driven by the need to develop a highly responsive data system to support real-time network monitoring for applications ranging from service quality management to network security to long-term networking research.

Data stream warehousing provides real-time data loading as well as long data histories and deep analytics. Real-world data provides many challenges: late-arriving data, highly heterogeneous data arrival patterns and latencies, frequent changes in data schemas. To address these challenges, we have been performing research into data stream warehousing technologies. These techniques include streaming description tables (Section 4.4), update propagation (Section 4.3), schema change (Section 4.5), temporal consistency (Section 5.3), incremental updates (Section 5.2), partition reorganization (Section 5.1), partition co-location (Section 5.4) and partition-wise query planning and optimization (Section 5.5). We launched the Tidalrace project to have a clean slate in which to implement these new techniques.

Data stream warehousing presents many interesting challenges that are open research areas: real-time scheduling, data quality, temporal consistency, and query optimization are a few. One motivation of developing Tidalrace as a new system is to have a platform with which to explore these research areas.

9. REFERENCES

[1] D. Abadi et al., *The Design of the Borealis Stream Processing Engine*, Proc. CIDR, 2005.

[2] Applied Materials. *Techedge Prism*, 2013. http://www.appliedmaterials.com/technologies/library/techedge-prizm

[3] M. Balazinska, Y.C. Kwon, N. Kuchta, D. Lee. Moriae: *History-Enhanced Monitoring*, Proc. Conf on Innovative Data Systems Research (CIDR) 2007.

[4] S. Chandrasekaran, M. Franklin. *Remembrance of Streams Past*. VLDB, 2004.

[5] J. C. Corbet et al. *Spanner: Google's Globally-Distributed Database*, Proc. OSDI, 2012,

[6] C. Cortes, K. Fisher, D. Pregibon, A. Rogers: *Hancock: a language for extracting signatures from data streams*. KDD 2000: 9-17

[7] C. D. Cranor, T. Johnson, O. Spatscheck V. Shkapenyuk*: Gigascope: A Stream Database for Network Applications*. SIGMOD Conference 2003: 647-651

[8] G. DeCandia et al. *Dynamo: Amazon's Highly Available Key-value Store* Proc. SOSP 2007.

[9] N. Dindar, P.M. Fischer, M. Soner, N. Tatbul. *Efficiently Correlating Complex Events over Live and Archived Streams*. DEBS, 2011.

[10] M. Y. Eltabakh, Y. Tian, F. Ozcan, R. Gemulla, A. Krettek, J. McPherson. *CoHadoop: Flexible Data Placement and its Exploitation in Hadoop*. Proc. VLDB 2011.

[11] N. Folkert et al. *Optimizing Refresh of a Set of Materialized Views*, Proc. VLDB 2005.

[12] Z. Ge, J. Yates, L. Breslau, D. Pei, H Yan, D. Massey. *G-RCA: A Generic Root Cause Analysis Platform for Service Quality Management in Large ISP Networks*. ACM ACM Conference on Emerging Networking Experiments and Technologies, 2010.

[13] L. Golab, T. Johnson, J. S. Seidel, V. Shkapenyuk: *Stream warehousing with DataDepot*. SIGMOD Conference 2009: 847-854.

[14] L. Golab, T. Johnson, V. Shkapenyuk*: Scalable Scheduling of Updates in Streaming Data Warehouses*. IEEE Trans. Knowl. Data Eng. 24(6): 1092-1105 (2012)

[15] L. Golab, T. Johnson, S. Sen, J. Yates: *A Sequence-Oriented Stream Warehouse Paradigm for Network Monitoring Applications*. PAM 2012: 53-63

[16] L. Golab, T. Johnson*: Consistency in a Stream Warehouse*. CIDR 2011: 114-122

[17] R. Greer: *Daytona And The Fourth-Generation Language Cymbal*. SIGMOD Conference 1999: 525-526

[18] Q. Hardy. *G.E.'s 'Industrial Internet' goes big*, New York Times 2013. http://bits.blogs.nytimes.com/2013/10/09/g-e-s-industrial-internet-goes-big/

[19] H. Herodotou, N. Borisov, S. Babu. Query Optimization Techniques for Partitioned Tables. Proc. SIGMOD 2011.

[20] J. Hruska. *Applied Materials designs tools to leverage big data and build better chips*, ExtremeTech, 2013. http://www.extremetech.com/extreme/155588-applied-materials-designs-tools-to-leverage-big-data-and-build-better-chips

[21] T. Johnson, V. Shkapenyuk: *Update Propagation in a Streaming Warehouse*. SSDBM 2011: 129-149

[22] C. Kalmanek et al., *Darkstar: Using Exploratory Data Mining to Raise the Bar on Network Reliability and Performance*, DRCN 2009

[23] S. Krishnamurthy, M.J Franklin, J. Davis, D. Farina, P. Golovko, A. Li, N. Thombre. *Analytics over Continuous and DisContinuous (ACDC) Streams: The Truviso Approach*. Proc. ACM Sigmod 2010.

[24] K. Kulkarni, J.-E. Michels. *Temporal features in SQL: 2011*. ACM SIGMOD Record 41.3 (2012): 34-43

[25] W. Lam, L. Liu, S. T. S. Prasad, A. Rajaraman, Z. Vacheri, A. H.i Doan: *Muppet: MapReduce-Style Processing of Fast Data*. PVLDB 5(12): 1814-1825 (2012)

[26] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandier, L. Doshi, C. Bear*: The Vertica Analytic Database: C-Store 7 Years Later* . PVLDB 5(12): 1790-1801 (2012)

[27] P. Larson. *Data Reduction by Partial Aggrgation*. Intl Conf. on Data Engineering, pg. 706-715, 2002.

[28] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, D. Maier*: Out-of-order processing: a new architecture for high-performance stream systems*. PVLDB 1(1): 274-288 (2008)

[29] A. Mahimkar et al.. *Rapid Detection of Maintenance Induced Changes*: Proc. ACM Conference on Emerging Networking Experiments and Technologies, 2011.

[30] N. Marz. *Runaway complexity in Big Data and a Plan to Stop It*. Slideshare, 2012. http://www.slideshare.net/nathanmarz/runaway-complexity-in-big-data-and-a-plan-to-stop-it

[31] N. Marz, J. Warren. *Big Data: Principles and best practices of scalable realtime data systems*. Maning Publications, ISBN 1617290343, 2014.

[32] C. Olston, et al. *Nova: continuous Pig/Hadoop workflows*. SIGMOD Conference 2011: 1081-1090

[33] S. Peng, Z. Li, Q. Li, Q. Chen. *Event Detection over Live and Archived Streams*. WAIM, 2011.

[34] D. Quass and J. Widom. *On-line warehouse view maintenance*. SIGMOD 1997, 393-404.

[35] F. Reiss, K. Stockinger, K.Wu, A. Shoshani, J.M. Hellerstein. *Enabling real-time querying of live and historical stream data.* SSDBM, 2007.

[36] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, B. Christof: *Efficient transaction processing in SAP HANA database: the end of a column store myth*. SIGMOD Conf. 2012: 731-742

[37] M. Stonebraker, U. Cetintemel, S. Zdonik, *The 8 Requirements of Real-Time Stream Processing*, ACM SIGMOD Record 34(4) pg. 42-47, 2005.

[38] Storm. http://storm-project.net/

[39] P. A. Tucker, D. Maier, T. Sheard, L. Fegaras: Exploiting *Punctuation Semantics in Continuous Data Streams*. IEEE Trans. Knowl. Data Eng. 15(3): 555-568 (2003)

[40] K. Tufte, J. Li, D. Maier, V. Papadimos, R.L. Bertini, J. Rucker. *Travel Time Estimation Using NiagaraST and latte*. Proc. ACM SIGMOD Conf., 2007.

[41] R. van Renesse, F. B. Schneider. *Chain Replication for Supporting High Throughput and Availability*. Proc. OSDI 2004.

[42] . J. Wang, Z. Ge, J. Yates, H. Song, A. Mahimkar, Y. Zhang *Analyzing IPTV Set-Top Box Crashes*.ACM SIGCOMM Workshop on Home Networks, 2011.

[43] H. Yan, et al.: *Argus: End-to-end service anomaly detection and localization from an ISP's point of view*. INFOCOM 2012:2756-2760

[44] M. Zheng, J. Tucek, F. Qin, M.Lillibridge. *Understanding the Robustness of SSDs under Power Fault*, Proc. Usenix File and Storage Technologies, 2013.