Stream Warehousing with DataDepot

Lukasz Golab, Theodore Johnson, J. Spencer Seidel, and Vladislav Shkapenyuk
AT&T Labs – Research
180 Park Avenue, Florham Park, NJ, USA 07950
{lgolab,johnsont,spence,vshkap}@research.att.com

ABSTRACT

We describe DataDepot, a tool for generating warehouses from streaming data feeds, such as network-traffic traces, router alerts, financial tickers, transaction logs, and so on. DataDepot is a streaming data warehouse designed to automate the ingestion of streaming data from a wide variety of sources and to maintain complex materialized views over these sources. As a streaming warehouse, DataDepot is similar to Data Stream Management Systems (DSMSs) with its emphasis on temporal data, best-effort consistency, and real-time response. However, as a data warehouse, DataDepot is designed to store tens to hundreds of terabytes of historical data, allow time windows measured in years or decades, and allow both real-time queries on recent data and deep analyses on historical data. In this paper we discuss the DataDepot architecture, with an emphasis on several of its novel and critical features. DataDepot is currently being used for five very large warehousing projects within AT&T; one of these warehouses ingests 500 Mbytes per minute (and is growing). We use these installations to illustrate streaming warehouse use and behavior, and design choices made in developing DataDepot. We conclude with a discussion of DataDepot applications and the efficacy of some optimizations.

Categories and Subject Descriptors

H.2.7 [**Database Management**]: Database Administration – *Data warehouse and repository.*

General Terms

Algorithms, Design.

Keywords

Data stream warehousing, Real-time data warehousing.

1. INTRODUCTION

Data warehouses are used for complex off-line analysis of data periodically collected from operational databases. On the other hand, Data Stream Management Systems (DSMSs) process relatively simple queries on "live" data feeds. Streaming warehouses (also known as active warehouses [10]) combine these two technologies for applications that perform data mining and analysis in near real-time. For example, an Internet Service Provider (ISP) may collect various data streams from its network, including system logs and router CPU-usage measurements. By capturing these streams in a warehouse, the ISP may compare new results with historical patterns, correlate events across streams, etc. Other examples of streaming data feeds include financial tickers, point-of-sale purchase transactions, sensor data, and results of scientific experiments.

Data streams are usually append-only. For example, a typical stream of router CPU-usage polls may consist of three fields, *timestamp, router,* and *CPU_usage*, with new records generated by each router every minute and existing records never modified. In that sense, a streaming warehouse is similar to a DSMS, with its emphasis on append-only temporal (timestamped) data, best-effort consistency (data may arrive out-of-order), and near real-time response to newly arrived data. However, a streaming warehouse must also manage tens to hundreds of terabytes of historical data, allow time windows measured in years or decades, and allow both real-time queries on recent data and deep analyses on historical data.

In this paper, we describe DataDepot, a tool for generating warehouses from streaming data feeds. DataDepot automates the ingestion of streaming data and maintenance of complex materialized views in an underlying relational database. We discuss the architecture of DataDepot, its novel features, and the issues which drove its design choices. Notable aspects include:

- An Extract-Transform-Load (ETL) process that treats raw data as a non-materialized view—a useful flexibility for understanding properties of data feeds before an expensive ingest.
- DataDepot uses a timestamp-based horizontal partitioning scheme for storing very large tables. However, most tables have several timestamps (start time, end time, arrival time, transaction time, etc.). We discuss a flexible timestamp correlation mechanism which allows any of these timestamps (contained in a user- or table-defining query) to constrain the partitions used in answering the query.
- A flexible method for maintaining multiple layers of derived tables (materialized views) using partition dependencies.
- Support for real-time tables, including a hierarchy-aware update manager, which incorporates a novel real-time scheduling algorithm [5], and architectural features for performing real-time updates on historical tables (having time windows measured in months or years).
- Query and table consistency issues that arise in a streaming warehouse: Do we answer a query with the most recent possible data, or with the most recent stable data? How do we know when data are stable?

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'09, June 29–July 2, 2009, Providence, Rhode Island USA.
Copyright 2009 ACM 978-1-60558-551-2/09/06...$5.00.

- A Warehouse Dashboard allowing the DBA to track the current status of hundreds of streaming tables, monitor data feed errors and data quality problems, track resource usage, and raise alerts about looming problems.
- Streaming data-quality monitoring tools, including conditional integrity constraints [6], sampling very large tables to estimate data quality [2], and detecting missing and extraneous data.

DataDepot is currently being used for five very large warehousing projects within AT&T; one of these warehouses ingests 500 Mbytes per minute (and is growing), another contains 400+ (and growing) tables derived from complex streaming feeds and is managed by a single DBA. We use these installations to illustrate streaming warehouse use and behavior, and design choices made in developing DataDepot. We conclude with some performance measurements to illustrate the efficacy of some DataDepot optimizations.

2. OVERVIEW

DataDepot is a stream-data warehouse, and its data-management policies resemble those of a DSMS: pervasively temporal data, an emphasis on materialized views (i.e., continuous queries), and best-effort data consistency. In this section, we present an overview of DataDepot warehouse definition and view maintenance.

A DataDepot warehouse is a set of table definitions and database management scripts. The system architecture of a typical DataDepot warehouse is shown in Figure 1. We assume that an external feed-management process delivers raw data files from which the warehouse is (mostly) derived. Raw files may be generated by any of a large number of external sources and may arrive with widely varying frequencies. In our applications, the data feeds include Netflow (*http://www.cisco.com/go/netflow*) records, network-performance measurements (e.g., SNMP (*http://tools.ietf.org/html/rfc1157*) polls), and various system logs and alerts. In order to generate a DataDepot warehouse, the user writes a set of configuration files containing the definitions of the raw and derived tables, which will be discussed below. From the user-supplied configuration files, DataDepot generates table definitions as well as warehouse-maintenance scripts that are executed by the update manager.

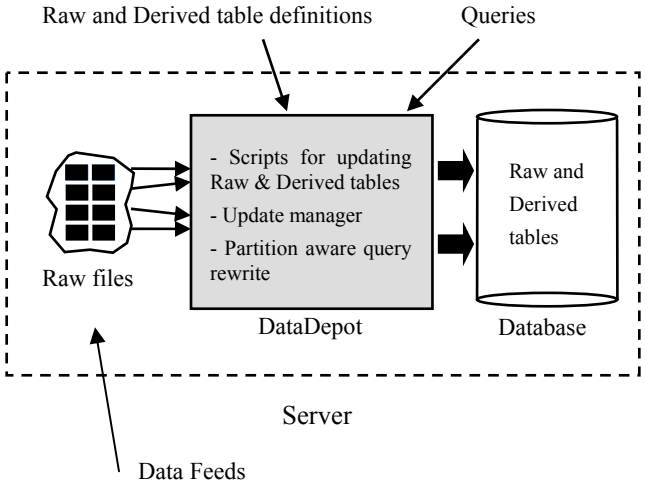

Figure 1. DataDepot warehouse model.

Each table maintained by DataDepot is horizontally partitioned on a timestamp attribute, forming a contiguous temporal window. As newer data arrive, new partitions are created as is necessary to store the data. Correspondingly, the oldest partitions are dropped to maintain a constant window size. While a DSMS might maintain a view window of seconds to hours, DataDepot maintains windows of months to years.

Currently, DataDepot uses Daytona [7] as its underlying database but can be adapted to other relational databases. Users write queries in EMF-SQL [8], which is an extension of SQL that supports complex grouping conditions. DataDepot exploits correlations between the partitioning attributes and other temporal fields (as specified by users in the configuration files) to rewrite queries in a way that minimizes the number of partitions that need to be accessed (see Section 2.3.5). The rewritten queries are processed by Daytona.

Daytona has two advantages that make it an attractive DBMS for large-scale warehousing. First, it translates all of its queries to efficient C code, making it a very fast database. Second, it supports large-scale horizontal partitioning using a readily accessible partition directory. Partition directories contain a variety of useful metadata, such as the temporal range of the data in each partition, time of last update, etc. DataDepot makes extensive use of these partition directories.

2.1 Raw Tables

Data sources and their ETL processes are encapsulated as Raw tables. In its most basic form, a Raw table is a non-materialized view of the source data files that comprise a table. Creating a non-materialized view of the raw data files comprises four tasks: finding the raw data files (especially new data files), classifying the files, extracting their contents, and organizing the contents as a relational record. The last task is similar to a standard "create table" statement that specifies attribute names, types, and various constraints (not null, etc.). The first three tasks are described below.

File finding: The source files for a table are generally placed in a particular directory and have particular names. When updating a Raw table, DataDepot calls a file finder that searches the specified directory for all files matching the specified pattern. New files (those not already in the Raw-table partition directory) comprise the update delta to the Raw table. While DataDepot provides default file finders, custom file-finding programs can be used instead—e.g., search for only the most recent files (a significant performance gain when the collection of raw files is large), or ignore the newest file, which may be in the process of being transferred by the feed management process.

File classification: The raw-file directory structure and naming convention often contains useful metadata, which might not exist in the data-file contents. For example, the data for the Raw table WebHits might be stored in a file whose path has the pattern */data/feeds/YYYY/MM/DD/Hits_source_hhmmss.dat.gz*, where *YYYY-MM-DD*, *hh:mm:ss* indicates the time at which the data was gathered to be transmitted to the warehouse, and *source* indicates the server farm where the data was gathered. DataDepot may be instructed to extract this metadata, include it in the schema of the raw table, and use the temporal information for partitioning.

Data extraction: A Raw-table configuration file must specify a program to extract data from the raw files. Usually, the program

is as simple as gzcat but in some cases involves data transformation (e.g., convert timestamps to GMT).

Useful warehouses have many levels of dependent materialized views; therefore, the Raw tables must supply enough information to efficiently propagate updates. The DataDepot update propagation process depends on two information structures: 1) partitioning a table based on the value of a timestamp field, and 2) marking the partitions of a table with a last-update timestamp.

In a streaming warehouse, each table contains one or more timestamp fields. The values of these timestamp fields (e.g., a datestamp and a clockstamp) should generally be larger in the newer data than in the older data. Each table must specify a monotonic increasing function by which one or more timestamp fields may be mapped to an integer (e.g., to partition a table by hour, convert the timestamp to Unix time and divide by 3600). The value returned by this function is the *temporal partitioning attribute* of a table.

As elaborated in Section 2.2, Derived tables must specify the relationship between their partitions and the partitions of their sources; when a source partition is updated, DataDepot propagates changes to the appropriate partition(s) of the derived table. Thus, when a Raw table is updated, we need to know which of its partitions have changed.

First, suppose that the partitioning attribute is computed from file paths and names during the file classification process; we call this type of Raw table a *direct* Raw table. In this case, DataDepot maintains the assignment of data files to partitions in the partition directory. When the update manager executes an update of a Raw table and the file finder identifies new files, the names of these files are added to the metadata of the corresponding partitions, and the last-update times of partitions that received new files are set to the current time. A direct Raw table is not materialized (records are extracted from new files only when updating a derived table).

Now suppose that the partitioning attribute references timestamps inside individual records. This necessitates a different kind of table, what we call an *indirect* Raw table. In this case, DataDepot must scan the contents of the raw files to determine the updated partitions. Extracting data from raw files is generally very expensive; we would prefer to perform this step only once. Therefore, we extract the contents of the new files in an indirect Raw table into a hidden "Loaded-to" table. The extraction query is a "Select *" query, which has a simple and efficient algorithm for an incremental materialized-view update: append the contents of the Raw-table delta to the (partition-wise) end of the Loaded-to table. As before, the partition directory of the Loaded-to table must maintain the last-update times of each partition.

2.2 Derived Tables

While Raw tables are non-materialized views (aside from Loaded-to derived products) representing data sources, Derived tables are materialized views and are therefore the tables that users normally query. A Derived table is defined by an EMF-SQL query over one or more source tables, which may themselves be raw or derived. The dependencies between tables may be of arbitrary complexity and depth (system resources permitting) as long as the dependency graph is acyclic (except for special cases). A configuration file for a Derived table must specify the defining query, the set of indices to build, the data and index directories, the partitioning function, and the priority of the table..

The queries used to define Derived tables may be of arbitrary complexity, and therefore might not be an efficient incremental view maintenance algorithm for the table. Therefore, DataDepot completely recomputes some Derived table partitions during the table update process. To ensure efficient updates, we rely on partition sizes being set such that partitions rarely get recomputed, and on precisely identifying which partitions need to be recomputed. To do so, the configuration must specify, for each source table, a lower bound and an upper bound of range of source partitions that affect data in a destination partition of the derived table. These bounds are expressed as source lower bound (*SLB*) and source upper bound (*SUB*) functions. Three examples are shown in Figure 2. Suppose the source table S has eight partitions of size one hour each. In the first example, the Derived table D also has eight one-hour partitions. In the second, D has four 2-hour partitions, and in the third, D is a five-hour moving average over the data in S.

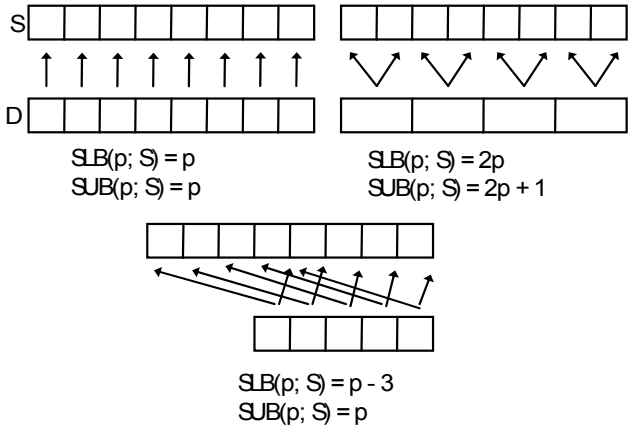

Figure 2. Examples of Source Dependencies

When a source table S of Derived table D is updated, we need to determine which partitions of D need to be recomputed. In principle, we can do this using the following algorithm:

```
Update_Set = {}
For each partition p in D's partition directory
  For each partition p' between SLB(p) and SUB(p)
    If p' in S's partition directory has a larger
    last-update timestamp than that of p in D's
    partition directory
      Add p to the Update_Set
```

In practice, we compute the inverse of *SLB* and *SUB* (denoted SLB^{-1} and SUB^{-1}) and use them to compute Update_Set as follows:

```
Update_Set = {}
For each partition p in S's partition directory
  For each partition p' between SLB^-1(p) and
  SUB^-1(p)
    If p in S's partition directory has a larger
    last-update timestamp than that of p' in D's
    partition directory, or if p' does not exist
    in D's partition directory
      Add p' to the Update_Set
```

Using SLB^{-1} and SUB^{-1} to compute Update_Set has the advantage of being a simpler procedure for determining when D's window advances in response to S's window advancing. The disadvantage

is that *SLB* and *SUB* must be invertible. We have not yet encountered a case in which *SLB* and *SUB* were not invertible.

2.3 Issues and Features

In this section, we summarize a variety of issues and features in DataDepot and the concerns which drove them.

2.3.1 Data Loading Via Files and Queries

In DataDepot, all Derived tables are defined by queries over other tables, ultimately over Raw tables, which are derived from raw data files. This architecture incurs some limitations—all data are generated externally, and there are no user inserts, deletes or updates. However, the benefits outweigh these limitations. Since all processing after the arrival of raw data is defined by explicit queries, the provenance of the data in all Derived tables is documented. This makes table definitions easier to understand, and makes the warehouse self-documenting. Furthermore, the completely declarative specification of the warehouse enables optimizations, especially multi-query optimizations. Finally, the loading-by-query architecture enables inexpensive but effective concurrency control and recovery, as discussed below.

2.3.2 Late Data

DataDepot is designed with the assumption that data generally arrive in temporal order, i.e., that the new data for a table will be placed in partitions whose temporal partitioning attribute has a value larger than those of existing partitions. Ideally, partitions should be sized so that their contents are computed only once. However, one cannot in general count on data arriving in order—our experience is that delays of hours to days are common. The problem of late data is the motivation for indirect Raw tables, where individual record timestamps may be different from the file timestamp.

2.3.3 Multi-version Concurrency Control (MVCC)

A data warehouse must ensure atomicity and recoverability. The simple update model adopted by DataDepot enables MVCC via partition directory swapping. When a table is being updated, a temporary partition directory is created that points to temporary copies of all the partitions that must be added or recomputed. At the end of the update, the temporary partition directory is swapped in place of the old directory in an atomic step. When a query accesses a table, it locks the partition directory via a shared lock, copies the locations of the necessary partitions, and releases the lock immediately. Thus, queries that started before an update continue to use the old directory. This model is similar to previous work on concurrency control in data warehouses, which describes maintaining multiple logical versions of the warehouse so that updates do not interfere with long-running queries (see, e.g., [11]). However, DataDepot implements partition-level versioning, not tuple-level versioning.

Since all of the data products in DataDepot are derived via a chain of one or more queries over raw data sources, recovery ultimately depends on the raw data. Late data are handled by indirect Raw tables. If a file needs to be deleted or replaced, the corresponding Raw table partition must be recomputed, and the normal update process propagates the revised data to all affected Derived tables.

2.3.4 Temporal Metadata

The "star schema" layout of a data warehouse assumes that while the fact table might be temporal, the dimension tables are generally static. We have found that this assumption generally does not hold, though the metadata tables might change slowly. For example, SNMP data feeds from various routers may not contain router names but rather the IP addresses of the router interfaces. A separate table contains the interface IP address-to-router name mapping, which may change from day to day. Thus, the only way to reliably model the state of the network in the past is to maintain this dimension table as another streaming table.

2.3.5 Temporal Correlation

Streaming data typically have more than one timestamp. A record might represent an interval of time (e.g., Netflow data, customer interaction) and therefore has both a start and an end timestamp. Timestamps may also be assigned by several entities: the local server, the data poller, and so on. A data record might have a Unix timestamp as well as a date stamp; users typically prefer to query using easily understood datestamps rather than cryptic Unix timestamps. Each of these timestamps tends to be highly correlated, e.g., start and end times may be at most 30 minutes apart. A Derived-table configuration file allows the user to specify which temporal fields are correlated. Using this information, DataDepot rewrites queries with range predicates on the correlated timestamps by adding range predicates on the partitioning fields, which reduces the number of partitions that need to be accessed.

3. REAL TIME AND HISTORICAL DATA

A fundamental feature of a streaming warehouse is that it allows queries on both real-time (very recent) as well as historical (aged) data. Real-time data warehouses require specializations that can conflict with the needs of historical warehouses. In this section, we describe how DataDepot supports both real-time and historical data.

3.1 Variable Partitioning

A major component of the cost of adding data to a warehouse is index rebuilding. We can overcome this cost by using small partitions; e.g. if we update a table every five minutes, we use five-minute partitions. However, very small partitions impose a significant overhead for historical tables, for which large partitions are preferable. If we want to store a 2-year window on a table using 5-minute partitions, we would need 210,240 partitions total. Such a large number of partitions slows down queries, e.g., to retrieve data for a particular day the warehouse would need to scan 288 separate indices. By using daily partitions, we can reduce the number of partitions to an easily manageable 730.

DataDepot offers a combined real-time/historical table, which uses variable partitioning to manage a historical (e.g. 2-year) table with real-time (e.g. 5 minute) updates. The most recent portion of the table is finely partitioned (say, 5-minutes), while the older part is coarsely partitioned (say, 4 hours or one day). When a collection of fine-grained partitions ages out of the real-time window (say, the most recent 3 days), it is rolled up into a coarse partition. The DataDepot update propagation algorithm (Section 2.2) and the temporal attribute correlations (Section 2.3.5) all work with variable partitioning.

3.2 Real-Time Scheduling

A typical streaming warehouse ingests dozens to hundreds of data feeds and supports several times that many Derived tables. The stream sources can be highly bursty, at times providing several

times more data than average. While the warehouse server might be a large parallel and/or clustered server, we still need to enforce resource control to avoid overwhelming resources and incurring (memory, disk-arm, CPU cache) thrashing. The real-time aspect of DataDepot warehouses requires that the resource controller be a real-time scheduler to ensure that the important tables (as determined via user-defined priorities specified in the configuration files) are kept as fresh as possible.

DataDepot faces an unusual real-time scheduling problem. For one, update tasks (increments to a table) can be deferred but cannot be dropped. The scheduler must account for frequent overload conditions during which low-priority tables can be deferred, but important tables must be kept fresh. The natural periodicity of the table updates in a typical warehouse can range from one minute to one day, with corresponding variations in update execution times. Finally, Derived tables form a loads-from hierarchy; therefore, the table update tasks also form a hierarchy as a Derived table can be updated only when all of its source tables have been updated.

We have developed new real-time scheduling algorithms [5] that minimize the weighted *staleness* (difference between the current time and the most recent data in a table) of a streaming warehouse. We have incorporated the best algorithm into the Update Manager, which also ensures that only one update of a table runs at a time, understands table dependencies so that a table update is invoked only when beneficial, and provides informative logs of update status.

4. DATA CONSISTENCY

By design, a streaming data warehouse attempts to load new data as frequently as possible. Thus, it is not clear which subset of the data is "consistent" or "stable". DataDepot determines data stability using the concepts of a *trailing edge* and a *leading edge*.

A reasonable definition of data consistency is as follows: a query on any table T must return the same answer as an equivalent query directly on T's sources at some point in the past or present. Suppose that T is derived from a sliding window join of tables T1 and T2, which have been updated at times 10:00 and 10:05, respectively, and that these updates have been propagated to T immediately. If T1 and T2 can incur arbitrary updates (inserts, deletes, or modifications), it may not be possible to "consistently" answer a query on T such that the result reflects the state of T1 and T2 as of, say, 10:00 (unless we stored each version of every record and were able to "roll back" the state of T2 to time 10:00). Fortunately, the append-only nature of data streams simplifies consistency issues in a streaming warehouse. For now, suppose that all data arrive in timestamp order. When evaluating a query on T, we can "extract" the state of T2 as of time 10:00 by ignoring records with T2-timestamps greater than 10:00 (we assume that all timestamps are retained when creating derived tables from multiple sources).

Formally, let $F_i(\tau)$ be the *freshness* of (a raw or derived) table *i* at time τ, defined as the maximum timestamp of any record in table *i* at that time. We define the *leading edge* of a set of tables *T* at time τ as the maximum timestamp of any record in any of the tables, i.e., $\max_{i \text{ in } T}(F_i(\tau))$. We also define the *trailing edge* of *T* at time τ as $\min_{i \text{ in } T}(F_i(\tau))$, i.e., the freshness of the least-fresh table in the set. Note that DataDepot loads all the available data (i.e., up to the leading edge). However, a consistent value for the trailing edge point (with respect to the above definition) may be computed by inserting timestamp predicates into the query.

Now, suppose that data may arrive out of order. Typically, there is a limit on the degree of disorder, e.g., SNMP data arrive at most one hour late (and are discarded if they arrive more than an hour late). Some sources may insert *punctuations* [12] into the data stream to explicitly inform the warehouse that, e.g., no more records with timestamps older than *t* will arrive in the future. We define the *safe trailing edge* of a set of tables *T* as $\min_{i \text{ in } T}(F'_i(\tau))$, where F'_i is the "safe" freshness of table *i*, i.e., the maximum time value such that no record with a smaller timestamp can arrive in the future. Returning to the above example, suppose that T1 has been updated at time 10:00 and we know that no more records with timestamps smaller than 9:58 will arrive. Further, T2 has been updated at time 10:05, and no more records with timestamps smaller than 10:01 will arrive. Answering queries with respect to the safe trailing edge of 9:58, rather than the trailing edge of 10:00, provides a consistent answer even with out-of-order arrivals.

5. DATA QUALITY MONITORING

DataDepot benefits from a number of data-quality tools for very large data sets that have been developed at AT&T. Typically, data semantics are expressed using some integrity constraints such as functional dependencies (FDs), inclusion dependencies, or aggregate constraints. Data-quality issues manifest themselves as violations of these constraints. For instance, we may expect each interface on each router to report its traffic at least five times an hour. The corresponding aggregate constraint may be expressed as a group-by query on router, interface and hour of day, having a count of at least five. Conversely, some measurements should arrive at most every fifteen minutes (to avoid overloading the routers), in which case router, interface, and Unix_time/(15*60) should be a key. In either case, we may tolerate a small number of violations, i.e., the constraints are *approximate*.

Streaming data are naturally heterogeneous—they are generated from multiple sources and may change over time. Thus, a constraint may not hold on an entire table and/or over a long period of time. The idea of a pattern tableau was proposed in [3] to concisely represent the subset of a table on which an FD applies. For each attribute participating in the constraint, a pattern consists of a value from the attribute's domain or a wildcard pattern, represented as a dash. We have developed an algorithm for mining pattern tableaux from the data in order to determine which subsets satisfy and *fail* an FD or an aggregation constraint [6]. For instance, recall the above aggregation constraint on traffic reports. A possible tableau indicating subsets that *fail* the constraint (i.e., those interfaces which are sending out fewer than five reports per hour) is shown in Table 1. The first pattern matches all interfaces on all routers of type *backbone* as reporting fewer than five traffic polls per hour at all times. The second pattern identifies all interfaces on a particular edge router, namely *Edge_router_5*, as being problematic. The third pattern matches all measurements, regardless of the router or interface, received between 10:00 and 10:59 on January 5, 2009. This concise representation of offending subsets is easier to understand and analyze than a raw stream of report counts for each router, interface and hour.

Table 1. Example of a Pattern Tableau for a Conditional Integrity Constraint

Router	Interface	Router type	Timestamp Hour
-	-	backbone	-
Edge_router_5	-	edge	-
-	-	-	2009-01-05, 10:00

In addition to data heterogeneity, the large volume of data in a streaming warehouse makes it difficult to characterize data quality. Many data warehouses store random samples for (approximately) answering simple queries [1]. While random samples are sufficient to estimate simple data quality checks, such as counts of records belonging to particular routers or interfaces, they are inadequate for reliably estimating the degree to which FDs and Conditional FDs (CFDs) hold. In [2], we have developed new sampling algorithms that produce compact summaries for estimating the *confidence* of FDs and CFDs with bounded error.

5.1 Warehouse Dashboard

A perpetual problem in warehouse management is ensuring the continued availability, quality, and freshness of the warehouse. While the scheduling and data-quality monitoring tools described in Sections 3.2 and 5 are of invaluable help, many problems remain. Ensuring the quality and delivery of a single data feed from a single source can be problematic; ensuring the quality and delivery of hundreds of data feeds collected globally is generally quite difficult. Other problems arise: table updates encounter problems for a variety of reasons, the warehouse server becomes overloaded and cannot update all tables, and so on.

We have developed a Warehouse Dashboard to provide an at-a-glance view of the current state of the warehouse. The most critical view is the freshness of each of the tables—essential information for the DBA and the database user alike. Additional views show error logs of the raw feed ingest processes, and the volume of data in a table over time. The Warehouse Dashboard is also a metadata repository, much of which is automatically collected from DataDepot configurations and user-defined function archives. We are continuing to add informative views of the warehouse contents, with additional data statistics and data quality reports.

6. APPLICATIONS

We have used DataDepot to build five very large data warehouses thus far. Four of these are production data warehouses ingesting upwards of 500 Mbytes/minute and growing. The most complex warehouse is maintained by AT&T Labs-Research to correlate many network performance, configuration, and status feeds that were previously maintained separately.

The network warehouse was developed to enable research on a variety of issues that cross the usual "silos" of network-data repositories. Currently, this warehouse loads more than 140 data feeds from 25 distinct silos, giving rise to over 140 Raw and 260 Derived tables. The shallow hierarchy reflects the research nature of the warehouse: the emphasis is on providing researchers with access to the data rather than on generating specific reports. Tables of special interest tend to have deeper layers of Derived tables—a continuing process as we gain expertise with the data. One-third of the tables are loaded at about 15 minute intervals; the others are loaded at about 8 hour intervals. Based on the many requests we have received, we plan to move many tables to 5-minute or 1-minute update schedules. The temporal windows on the tables range from 1 month to 2 years.

Being a project in Research, the network warehouse is run with limited resources. There is only one DBA for the project. Many of the tables have been defined by the researchers who are familiar with the feed and need the data. The Do-It-Yourself, low-budget aspect of this project was a strong impetus for the Warehouse Dashboard (Section 5.1), as auto-documentation and a clear readout of the warehouse status are critical.

The production applications of DataDepot tend to be better planned and supported. A typical silo'ed application will have perhaps a dozen raw feeds, ingested at 5 minute intervals, and deep levels of Derived tables corresponding to the variety of reports required for the application.

Although this paper is about the architecture of DataDepot rather than specific optimizations, we can report of the effectiveness of the temporal correlation optimization described in Section 2.3.5. One table, storing SNMP measurements, contains 410 Gbytes of data and is partitioned on the Unix timestamp. However, users typically query on the datestamp, so we added a correlation between datestamp and Unix timestamp in the warehouse configuration. Using the correlation, a query for a single day's worth of data takes 125 seconds; without the correlation the same query takes 7401 seconds.

7. CONCLUSIONS

We have described DataDepot—a tool for generating and maintaining streaming data warehouses in an underlying relational database. We have highlighted several issues that must be handled by a streaming warehouse, including loading new data in near real-time, combined queries against recent and historical data, maintenance of complex derived tables, dealing with multiple timestamps and with out-of-order data, data consistency, and streaming data quality.

We are continuing to develop DataDepot in order to improve its efficiency and meet the needs of our users. For instance, we are interested in distributing DataDepot across multiple machines to speed up data ingestion (DataDepot already allows very large tables to be distributed across file systems). We are also interested in optimization techniques for updating materialized views. The simple update propagation technique currently employed by DataDepot always recomputes partitions. However, there has been a great deal of work on incremental view maintenance that is applicable to our problem, e.g., [4]. Similarly, multi-query optimization strategies can help improve the efficiency of update propagation if multiple derived tables are defined on a single source table, e.g., [9]. Other research issues include streaming data consistency, streaming data quality, and refinements to the real-time infrastructure.

8. REFERENCES

[1] P. Brown and P. Haas. Techniques for Warehousing of Sample Data. *ICDE 2006*, 6.

[2] G. Cormode, L. Golab, F. Korn, A. MacGregor, D. Srivastava, and X. Zhang. Estimating the Confidence of Conditional Functional Dependencies. *SIGMOD 2009*, to appear.

[3] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *TODS* 33(2) (2008).

[4] N. Folkert, A. Gupta, A. Witkowski, S. Subramanian, S. Bellamkonda, S. Shankar, T. Bozkaya, and L. Sheng. Optimizing refresh of a set of materialized views. *VLDB 2005*, 1043-1054.

[5] L. Golab, T. Johnson, and V. Shkapenyuk. Scheduling Updates in a Real-Time Stream Warehouse. *ICDE* 2009, to appear.

[6] L. Golab, H. Karloff, F. Korn, D. Srivastava, and B. Yu. On generating near-optimal tableaux for conditional functional dependencies. *PVLDB*, 1(1): 376-390 (2008).

[7] R. Greer. Daytona and the fourth-generation language Cymbal. *SIGMOD 1999*, 525-526.

[8] T. Johnson and D. Chatziantoniou. Extending complex ad-hoc OLAP. *CIKM 1999*, 170-179.

[9] W. Lehner, B. Cochrane, H. Pirahesh, and M. Zaharioudakis. fAST refresh using mass query optimization. *ICDE 2001*, 391-398.

[10] N. Polyzotis, S. Skiadopoulos, P. Vassiliadis, A. Simitsis, and N.-E. Frantzell. Supporting Streaming Updates in an Active Data Warehouse. *ICDE 2007*, 476-485.

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

[12] P. Tucker *Punctuated Data Streams*. Ph.D. Thesis, Oregon Health & Science University, 2005.