Consistency in a Stream Warehouse

Lukasz Golab and Theodore Johnson
AT&T Labs – Research
180 Park Avenue, Florham Park, NJ, USA 07932
lgolab@research.att.com, johnsont@research.att.com

ABSTRACT

A *stream warehouse* is a Data Stream Management System (DSMS) that stores a very long history, e.g. years or decades; or equivalently a data warehouse that is continuously loaded. A stream warehouse enables queries that seamlessly range from real-time alerting and diagnostics to long-term data mining. However, continuously loading data from many different and uncontrolled sources into a real-time stream warehouse introduces a new consistency problem: users want results in as timely a fashion as possible, but "stable" results often require lengthy synchronization delays. In this paper we develop a theory of temporal consistency for stream warehouses that allows for multiple consistency levels. We show how to restrict query answers to a given consistency level and we show how warehouse maintenance can be optimized using knowledge of the consistency levels required by materialized views.

1. INTRODUCTION

Many real-world enterprises generate streams of information about their operations and require real-time response for their maintenance. Examples include financial markets, communications networks, data center management, and vehicular road networks. Data Stream Management Systems (DSMSs) have been developed to provide real-time analysis and alerting of these and other data streams, typically by processing events in-memory and over a short time window. However, users often want to perform longer-term analyses over large time windows on the data streams, e.g. to determine the conditions that should raise alerts.

While it is possible to build separate systems for either real-time or long-term data analysis, a system which provides both capabilities is more useful. The window of data used for queries can seamlessly range from short term to very long term, making it difficult to decide where to divide the systems. Furthermore, historical data can provide a context for interpreting new data [2]. A *stream warehouse* bridges the short-term vs. long-term gap by loading data continuously in a streaming fashion and warehousing them over a long time period (e.g. years). Stream warehouse systems, such as Moirae [2], latte [22], DataDepot [11], Everest [1], and Truviso [10], have been applied to monitoring applications such as data centers [2], RFID [23], web complexes [1], highway traffic [22], and wide-scale networks [14].

A DSMS normally monitors a nearly-instantaneous and ordered data feed of, e.g., network packets [8], financial tickers or sensor measurements. However, a stream warehouse operates on longer time scales, and, instead of processing data from a localized source, it receives a wide range of data feeds from disparate, far-flung, and uncontrolled sources. For example, the Darkstar network management system [14] (built using DataDepot) receives more than 100 distinct data feeds, each of which collects data from a worldwide communications network using many different dissemination mechanisms. These distinct feeds need to be cross-correlated and analyzed into higher level data products for use by network analysts. In such a widely distributed and heterogeneous environment, one can no longer assume that data within a stream arrive in time-order (or nearly so), or that streams are synchronized with each other. This leads to new *temporal consistency* problems: we want to load new data (and propagate changes to the materialized views maintained by the warehouse) as quickly as possible, but "stable" results may require significant synchronization delays. (Note that the temporal consistency issues studied in this paper are orthogonal to transactional consistency issues that arise from multiple data writers and/or readers.)

Consider a network monitoring system that collects performance measurements, such as router CPU utilization or the number of packets forwarded, and various system logs. Suppose that an alerting application generates an alarm whenever the CPU usage of a router exceeds a supplied threshold. If a high-CPU-usage record arrives, the application should not have to wait until all temporally preceding data have arrived before taking action. Similarly, a view containing all the routers that have generated at least ten critical system log messages in any one-minute window can be updated whenever the message count for a particular router, call it *r*, reaches ten; we do not need to see data from other routers, nor do we need to wait and see if any more messages from *r* arrive in this window. On the other hand, suppose that we want to maintain aggregated statistics for each time window. It may be better to wait until all the expected measurements have arrived before updating the statistics over the latest window, both in terms of interpretability (aggregates computed on incomplete data may not be accurate) and update efficiency (we want to avoid re-computing expensive aggregates while data are still trickling in).

These types of problems become even more challenging in production stream warehouses that correlate a wide variety of highly disordered and asynchronous feeds and maintain complex

5th *Biennial Conference on Innovative Data Systems Research (CIDR '11)* January 9-12, 2011, Asilomar, California, USA.

materialized view hierarchies. Such warehouses often support critical applications; examples from the networking domain include real-time network troubleshooting and anomaly detection [14]. However, without an understanding of temporal data consistency, we may not know how to trust the answers.

Motivated by our experiences with production stream warehouses, we present *temporal consistency* models for a stream warehouse that range from very weak to very strong, and we show how they can be tracked and used simultaneously. Given that warehouse tables are typically partitioned by time, the key technical novelty is to reason about and to propagate consistency information at the granularity of partitions. Since a significant part of the value of a stream warehouse is its ability to correlate disparate data sources for the users, our models describe the state of the data in an intuitive way that allows users to interpret real-time query results. For instance, a partition that is guaranteed not to change is marked "closed", while one that may be updated with new data, but whose existing data are guaranteed not to change, is marked "append-only". Since warehouse maintenance involves propagating changes across view hierarchies, we also discuss disseminating consistency level information from base tables to materialized views and vice versa. We show that stronger consistency levels not only provide assurances for query results, but they can also be used to avoid unnecessary computations. Finally, we discuss applications of our models to monitoring *data stream quality*.

2. BACKGROUND AND MOTIVATION

A DSMS continuously ingests data from one or more *data feeds*, and processes a collection of long-running queries over these feeds. Many sources can produce a data feed: a stream of measurements, log files delivered from an external source, a log of updates to a transactional store, and so on. The feed regularly presents a *package* of records for ingest into the stream system. The records in a package are stamped with the time of the observation (or observation time interval), and also the package itself is often timestamped. The set of timestamps in a package are generally highly correlated with the package timestamp and delivery time.

Data feeds are usually append-only; i.e., records that have arrived in the past are not deleted or modified in the future. For example, a feed of network measurements may have a schema of the form *(timestamp, router_id, avg_cpu_usage)*, with each record corresponding to the average CPU usage of the router with the given *router_id* recorded at the given time(stamp). We may receive a new package every five minutes, containing new CPU usage measurements for each router. Here, data from old packages (old measurements) are never deleted or modified. However, in some applications, old packages may be revised and retransmitted.

When a package arrives in a DSMS, the conventional behavior is to fully process the new records (modulo operator scheduling policies [5]). Some exceptions occur: a *sort* operator might reorder slightly disordered streams, and blocking operators such as aggregation and outer join might delay some or all of their output until a punctuation [21] indicates end-of-window. However, these mechanisms assume that streams are mostly-synchronized and mostly-ordered, so that buffering costs and processing delay times are small (the discussion of punctuation generation in [13] implicitly assumes that streams are synchronized).

As mentioned, a stream warehouse faces more challenging problems of disorder in its input streams. We have found the following disorder problems within the Darkstar warehouse:

Data arrive in a smear over time

In the course of operating several DataDepot warehouses, we noticed that any given package of data contains records with a range of timestamps. This behavior is not unexpected since data are gathered from world-wide network elements. We investigated this phenomenon by examining the data arrivals of several Darkstar tables.

We first examined arrivals for table C, which contains 5-minute statistics about router performance – a package normally arrives once every 5 minutes. We found that 23 percent of the packages (covering a 10-day period) contain some data for a previous 5-minute period, and sometimes for data up to an hour old (the packages frequently arrive late also). In another table, T, loaded at 1-minute intervals, every package except one contained records for a previous time period (observed over a 7-day interval). A third table, S (loaded at 1-minute intervals), showed the greatest disorder: each package contained data for an average of 4.5 previous time periods. The degree of disorder changes over time, as illustrated in Figure 1 which plots the number of time periods with at least one record in any given package. We hypothesize that the degree of disorder within a package is related to load on the data delivery system.

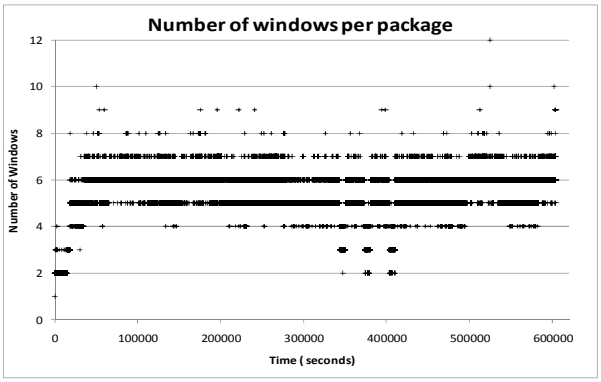

Figure 1. Number of time periods in one package for S

Data sources are unsynchronized

Different data feeds use different collection and delivery mechanisms, and therefore they tend to have different degrees of currency. We considered three feeds, the previously mentioned C and T (containing router alerts), and a third feed WD (packet loss and delay measurements), and sampled the lateness of the most recent data in each of these tables. On average, T was 6 minutes behind, C was 17 minutes behind, and WD was 47 minutes behind. Again, we believe that the currency of these feeds changes according to the load on their data delivery system.

Late arrivals are common

Significantly late arrivals are not common enough to readily measure, but in our experience they occur often enough to be an

operational concern – corroborated by another recent study [15]. Often the problem is a temporary failure of a component in the data delivery system. Occasionally, a portion of the source data is discovered to be corrupt and needs re-acquisition and reloading.

Given the large data volumes and high disorder in the source streams of a stream warehouse, conventional in-memory buffering techniques are prohibitively expensive [10]. Compounding the problem are complex view hierarchies. For example, Figure 2 shows a fragment of a real-time network monitoring application which searches for misbehaving routers, involving WD and other data (the full application has another 21 tables). The octagons are the base tables, while boxes identify tables that are often queried. These types of applications are too large to manage using conventional means and too complex to be understood without consistency assurances.

Another problem is that there can be multiple notions of consistency that users desire. For example, some Darkstar users (or applications) require access to router alerts (e.g., T) as soon as possible, and need to correlate them with the most recent possible router performance reports (e.g., C). Other users (or other materialized views) might need stable answers to queries based on these streams, even at the cost of a moderate synchronization delay.

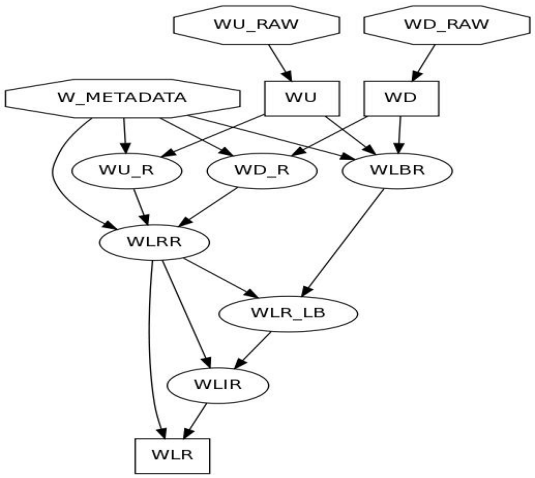

Figure 2. Data flow in an application fragment

3. SYSTEM MODEL

This work was motivated by the practical problems encountered by users of our DataDepot stream warehouse. We phrase the system model in DataDepot terms, but the model applies to all of the stream warehouses we have seen (perhaps with a change of phrasing).

A stream warehouse is characterized by streaming inputs, by a strong emphasis on the temporal nature of the data, and by multiple levels of materialized views. To manage a long-term store of a data stream, the stream is split into temporal *partitions* (or panes [16], windows [4][10], etc.). Each temporal partition stores data within a contiguous time range. The collection of temporal partitions of a stored stream comprises a complete and non-overlapping range of the stored window of the data stream. A feed package may contain data for multiple partitions, as shown in Figure 1. The storage of a high-volume stream may require additional partitioning dimensions, but we will not be concerned with this complication in this paper.

A data warehouse maintains a collection of *materialized views* computed from the raw inputs to the warehouse. Materialized views are used to accelerate user queries by pre-computing their answers and to simplify data access by cleaning and de-normalizing tables. A stream warehouse typically has a large collection of materialized views arranged as a Directed Acyclic Graph (DAG). The DAG tracks data dependencies, e.g. that view V is computed from streams A and B (Figure 2 shows a data flow DAG, the reverse of a dependency DAG). A stream warehouse also tracks temporal dependencies, e.g. that data in V from 1:00 to 1:15 are computed from data in stream A from 1:00 to 1:15 and from data in B from 12:30 to 1:15 (as in Figure 3).

Let V be a warehouse table. We assume that V has a *timestamp field*, **V.ts** which tends to increase over time. Further, we assume that every table V is temporally partitioned, and that the partitions are identified by integer values so that **V(t)** is the t^{th} partition of V. Associated with V is a strictly increasing *partitioning function* $\boldsymbol{pt_V(t)}$. Partition *t* of V contains all and only those data in V such that

$$pt_V(t) \leq V.ts < pt_V(t+1).$$

Base tables are loaded directly from a source stream (for example, WU_RAW in Figure 2). Derived tables (materialized views) are defined by a query over other base and derived tables (for example, WU_R in Figure 2). We define **S(V)** to be the set of source tables of V, e.g. S(WU_R) = {WU, W_METADATA}. We assume that all derived-table-defining queries exhibit temporal locality (e.g., they may be defined over a sliding window).

Let S be a table in S(V). Then **Dep(V(t),S)** is the set of partitions in S that supply data to V(t), and **Dep(V(t))** is the set of all partitions that supply data to V(t) regardless of the source table. For example, suppose that in Figure 3, each partition represents 15 minutes of data, and that partition 20 corresponds to 1:00 through 1:15. Then Dep(V(20), B) = {B(20)} and Dep(V(20)) = {A(18), A(19), A(20), B(20)}. When any of the partitions in Dep(V(20)) are updated, V(20) must also be updated (incrementally, if possible, or by being re-computed from scratch).

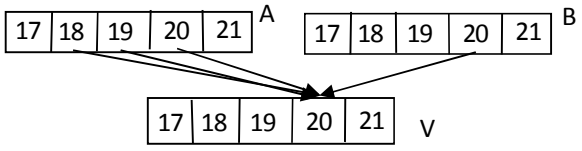

Figure 3. Partition dependencies

4. CONSISTENCY MODELS

Our basic notion of temporal consistency assigns one or more markers to each temporal partition in a table. Consistency markers can be thought of as a generalization of punctuations, since multiple consistency levels would be used in general. Below, we propose two related but different notions of

consistency. The first, *query consistency* defines properties of data in a partition that determine if those data can be used to answer a query with a desired consistency level. The second, *update consistency*, propagates table consistency requirements and is used to optimize the processing of updates to a stream warehouse.

4.1 Query Consistency

Our definition of query consistency starts at the base tables. For the purposes of this discussion, we use a minimal set of three levels of consistency, but many more are desirable in practice. We choose this particular set of three levels because they are natural and they form a simple hierarchy, but they also illustrate some interesting aspects of query consistency. However, an actual implementation of a warehouse would likely use a more refined set of consistency levels, as we will discuss in Section 5.

Let B be a base table and let B(d) be one of its partitions. Then:

- **Open(B(d))** if data exist or might exist in B(d).
- **Closed(B(d))** if we do not expect any more updates to B(d) according to a supplied definition of expectation; e.g., that data can be at most 15 minutes late.
- **Complete(B(d))** if Closed(B(d)) and all expected data have arrived (i.e., no data are permanently lost).

The notions of Open and Closed consistency are the natural minimal and maximal definitions. Complete consistency is stronger, and it is motivated by DataDepot user requirements: only perform analysis on complete data partitions because otherwise one may get misleading results (however, Closed partitions are often acceptable to users). Of course, the vagaries of the raw data sources may make it difficult to precisely establish when a partition has achieved one of these levels of consistency; this is similar to the problem of generating punctuations. However, several types of inference are possible:

- If there is at least one record in a partition, we mark it as Open. However, a partition might have Open consistency even though it is empty: no data might ever be generated for it. We might mark an empty base table partition as Open if we can infer that some data could have arrived, e.g. if a temporally later partition is non-empty.
- We might know that exactly five packages provide data for a partition and that packages rarely arrive more than one hour late. If so, we can mark a partition as both Closed and Complete if all five packages have arrived. If only four have arrived, but an hour has passed since the expected arrival time of the fifth one, we would only mark the partition as Closed. If the fifth package never arrives, this partition never becomes Complete.

The consistency of a partition of a derived table is determined by the consistency of its source partitions. Each level of consistency has its own inference rules, and inference is performed for each consistency level separately. The most basic inference rule is as follows: *for consistency level C, infer C(V(t)) if C(S(d)) for each S(d) in Dep(V(t))*. However, by analyzing the query that defines a materialized view we can sometimes create a more accurate inference rule.

Let us consider an example set of inference rules using our set of three consistency levels. Let V be a derived table and let V(t) be one of its partitions.

Query Consistency Inference

- Let RQD(V), a subset of S(V), be the non-empty set of tables referenced by "required" range variables, i.e., those used for inner-join or intersection.
  - If RQD(V) is non-empty, then **Open(V(t))** if for each S in RQD(V), there is a S(d) in Dep(V(t),S) such that Open(S(d)).
  - If RQD(V) is empty, then **Open(V(t))** if there is a S(d) in Dep(V(t)) such that Open(S(d)).
- **Closed(V(t))** if Closed(S(d)) for each S(d) in Dep(V(t)).
- **Complete(V(t))** if Complete(S(d)) for each S(d) in Dep(V(t)).

The Closed and Complete consistency levels use the basic inference rule, but by analyzing the query that defines materialized view V we can avoid labeling a partition V(t) as Open when no data can be in it. Section 5 contains additional examples of query-dependent consistency inference rules.

The inference that a partition of a derived table has a particular consistency level is computed top-down (from source to dependent tables). Normally, this inference would be performed at view maintenance time by comparing source with destination consistency metadata. This maintenance can be performed globally, as with, e.g., Oracle [9], or piecemeal, as with DataDepot [11]. Note that the consistency of a partition can change even though the partition does not need to be updated, e.g., due to a base table partition becoming Closed as well as Open.

For example, consider table V computed by an inner join of A and B as shown in Figure 4. In this figure, we represent Open, Closed, and Complete consistency markers by O, Cl, and CM, respectively, and we omit an O marker if a Cl marker exists. Partition 1 of V can be inferred to have Closed consistency, since both sources are Closed, but not Complete consistency; however partition 2 can be inferred to be Complete. Partition 3 is Open because both A and B can contribute an Open (or Closed) partition. Partition 4 cannot even be inferred to be Open.

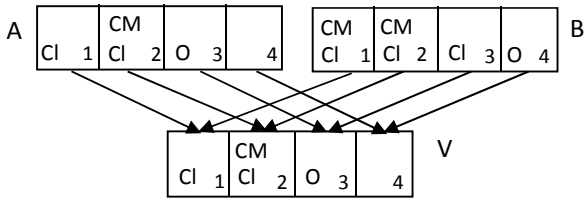

Figure 4. Query consistency inference

Query consistency markers ensure the consistency of query results. In Darkstar applications, ensuring temporal consistency is critical, but very difficult without warehouse support. Applications such as RouterMiner and G-RCA [14] enable real-time network troubleshooting by correlating data from feeds including C, S, T, WD and many others; however, each of these

feeds produces base tables with widely varying timeliness (recall Section 2).

An outline of the procedure for ensuring the consistency of a query is to treat the query as a derived table and determine its partition dependencies. A query can be answered with a given level of consistency if that consistency level can be inferred from the set of all source partitions accessed by the query. A query that cannot be answered with the desired consistency can have its temporal range trimmed (or its consistency relaxed). For example, if we are performing a selection on table V in Figure 4 and we require Complete consistency, then the inference rules state that the query can only be run on the data in partition 2.

While the proposed mechanism for ensuring query consistency is general, it can be confusing to users. A convenient way to summarize the state of a (base or derived) table is a *consistency line*. The C-consistency line of table V is the maximum value of *pt* such that all partitions V(t), $t \leq pt$, have C(V(t)). A query that references tables S_1 through S_n can be answered with C-consistency if it is restricted to accessing partitions of S_i at or below the C-consistency line of S_i for each i=1,..,n. In previous literature, we have referred to the Open-consistency line as the *leading edge* of a table, and the Closed-consistency line as the *trailing edge* [11]. A Complete-consistency line is likely of little value since some partitions might permanently fail to become Complete.

For example, the Open-line (leading edge) of table V in Figure 4 is partition 3, while the Closed-line (trailing edge) of V is partition 2. We cannot define a Complete line since partition 1 is not Complete.

4.1.1 Case Study

We now give an example of how applications can choose and exploit query consistency guarantees. A fragment of one of the Darkstar applications was shown in Figure 2. This application processes packet delay and packet loss measurements to come up with network alarm events. These measurements are taken roughly every five minutes, one measurement for each link in the network. A loss or delay alarm record is produced for a given link if there are four or more consecutive loss or delay measurements, respectively, that exceed a specified threshold. If a measurement for a given link is missing in a 5-minute window, it is considered to have exceeded the threshold for the purposes of alarm generation. In Figure 2, WLR is the materialized view that contains loss alarm records, each record containing a link id, the start and end times of the alarm, and the average packet loss and delay during the alarm interval. The size of each WLR partition is five minutes, which corresponds to the frequency of the underlying data feeds. The ovals in Figure 2 correspond to intermediate views that implement the application logic (e.g., selecting measurements that exceed the threshold, computing the starting point of each alarm event, computing alarm statistics, etc.). To complete the application, a Web-based front end displays the current and historical alarms by periodically querying the WLR table.

Since this is a real-time alerting application, one may argue that WLR should have Open consistency; i.e., it should be loaded with all the available data at all times. However, the problem is that missing measurements are assumed to have exceeded the threshold. Thus, if we attempt to update WLR before the latest measurements arrive, we will incorrectly assume that all of these measurements are missing and we may generate false alarms. Instead, it is more appropriate to use Closed consistency for WLR, with partitions closing at each 5-minute boundary. Note that Complete consistency may not be appropriate for this application since we do not want to delay the generation of network alarms for the data that have already arrived, even if a partition is not yet complete.

4.2 Update Consistency

In addition to understanding data semantics and query results, another use for consistency is to minimize the number of base table and view updates in a warehouse. For an example drawn from experience, consider a derived table V defined by an aggregation query which summarizes a real-time table S with once-per-5-minutes updates (with 5-minute partitions) into a daily grand-total summary (with per-day partitions). If V is updated every time S is updated, V would be updated about 288 times (1440 minutes in a day / 5) before the day is closed out. If we are interested in the grand total rather than the running sum, this procedure for updating V is wasteful. Here, a partition of V is only useful if it has Closed consistency, so we should only compute it when it can be safely Closed.

The *update consistency* of a table is the minimal consistency required by queries on it or its dependent tables, and determines when to refresh its partition(s). A partition of a table is computed only when it can be inferred to have a query consistency matching the desired update consistency.

Naively, we might require the warehouse administrator to mark each table with its desired update consistency. However, any given table may supply data to many derived tables, each with differing types of update consistency. We need an algorithm for determining what kind of update consistency table S should enforce.

Furthermore, not every view is primarily intended for output. A table might be materialized to simplify or accelerate the materialization of another table, or it might be a partial result shared by several tables (see, e.g., the application fragment in Figure 2). We assume that output tables are marked as such (all leaf-level materialized views are output tables). A table can be marked with one of the following labels:

- **Prefer_Open**: a table that does not have to reflect the most recent data, but one whose partitions can be easily updated (in an incremental manner) if necessary; e.g., *monotonic* views such as selections and transformations of one other table.
- **Require_Open**: a real-time table in which any possible data must be provided as soon as possible.
- **Prefer_Closed**: Tables whose partitions are expensive to re-compute, such as joins and complex aggregation (depending on the incremental maintenance strategy).
- **Prefer_Complete**: a table whose output is only meaningful if the input is complete.

All output tables need to be marked with these initial labels, which may be more effort than the warehouse administrator cares to expend. By default, selection and union views may be marked Prefer_Open because they can be very easily updated. Join and

aggregation views may be marked Prefer_Closed since it is more efficient to perform batch updates to them rather than continuously updating them whenever new data are available (or because users may not be interested in partial aggregates). We note that Prefer_Open is a "don't care" type of condition.

The algorithm for determining update consistency works in a reverse breadth-first search (BFS) of the data flow DAG, starting from the leaf-level views and working to the roots (base tables). When a table T is selected for processing, all of its dependent tables have received their final marking. To mark table T, we follow a resolution procedure. Let M be the set of dependent table markings, along with the marking of table T, if any (internal-use tables might not be marked).

The three consistency levels we are using form a hierarchy: Complete implies Closed, and Closed implies Open. The general resolution procedure is to choose the lowest level of consistency in M, with the "don't care" consistency level as a fallback. Therefore our update consistency resolution procedure is simple and produces a single result. There is one complication: it is likely that not all base table partitions will ever be labeled Complete, and therefore we should use Complete update consistency only if all dependent tables use Complete update consistency.

Update Consistency Resolution:

1. If Require_Open is in M, mark T as Require_Open
2. Else, if some label in M is Prefer_Closed, mark T as Prefer_Closed
3. Else, if all labels in M are Prefer_Complete, mark T as Prefer_Complete
4. Else, mark T as Prefer_Open.

Tables marked Require_Open or Prefer_Open use Open update consistency, while tables marked Prefer_Closed (resp. Prefer_Complete) use Closed (resp. Complete) update consistency.

Consider the example illustrated in Figure 5. The leaf tables (V, W, X, Y) are all output tables, indicated by a rectangle, with pre-assigned update consistency levels of (Require_Open, Prefer_Complete, Prefer_Closed, Prefer_Open) respectively. These tables are considered first in the reverse BFS search of the DAG. When one of these tables is processed, its own label is the only entry in M, so each table in (V, W, X, Y) is assigned its preferred update consistency. Non-output tables (A, B, C) are processed next. When one of these tables is processed, the markings of its successor tables are the entries in M. For example, when B is processed the entries in M are (Prefer_Complete, Prefer_Closed) so the resolution procedure marks B as Prefer_Closed.

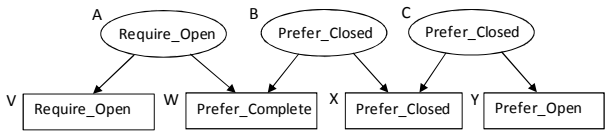

Figure 5. Update consistency inference

4.2.1 Experimental Evaluation

To see the potential performance benefit of using update consistency, we collected the number of updates performed on tables WU_RAW, WD_RAW, and WLR in Figure 1, over a 10 day, 18+ hour period. WU_RAW and WD_RAW (are supposed to) receive updates every 15 minutes; therefore we expect 1033 updates to these tables during the observation period. W_METADATA is another input, but it receives updates only once per day, so we will ignore it in our analysis.

The implementation of DataDepot that we measured did not incorporate update consistency. The only update scheduling options available were *immediate* (update a table whenever one of its sources has been updated) and *periodic* (e.g. every 15 minutes). Since the W application is real-time critical, we used immediate scheduling to minimize data latency. The extensive use of joins in this application suggests that immediate updates are likely to be inefficient, since one will often perform an (inner) join update when data from only one range variable is available. Without an update consistency analysis, however, the warehouse has no basis for *not* performing an update when data from only one range variable is available, since the join might be an outer join, and which source table supplies the outer join range variable is not clear.

During the observation period, there were 1364 updates to WU_RAW and 1359 to WD_RAW. The excess over the expected 1033 updates are due to late arrivals of some of the packages that comprise the data in a partition (recall the discussion in Section 2). Under Open update consistency, the number of updates to WLR should be 1033 plus one for each excess update to one of the RAW tables, for a total of 1690 updates. We observed 4702 updates to WLR: 3012 unnecessary updates. The use of update consistency clearly has the potential to be a significant optimization since we could reduce the number of updates to WLR by 64 percent. If we used Closed consistency, we could reduce the number of updates by 78 percent.

5. EXTENSIONS

Our framework for computing and using query and update consistency is general and can be extended to additional consistency levels, as long as one can produce safe inference rules.

As we discussed in Section 4, determining the consistency level of a base table partition is a matter of guesswork. Closed partitions are generally not really closed since very late data might arrive, sources might provide revisions to previously loaded data ("Sorry, we sent you garbage"), and so on. Thus, it may be useful to provide different levels of closed-ness according to different definitions. For example, WeakClosed(V(t)) might mean that the data are probably loaded and stable enough for queries, while StrongClosed(V(t)) might mean that we are certain enough that no more data will arrive and that we will refuse to process revisions. The definition of Closed in Section 4.1 corresponds to WeakClosed here.

If we have reasonably accurate and stable statistics about late arrivals, we can associate specific time-out values with various levels of closed-ness. For instance, we may know that revisions and updates mostly occur within five minutes of the expected partition closing time, and very few occur an hour later. A similar

example is X-Percent-Closed(V(t)), which indicates that X percent of the data will not change in the future. This consistency marker is motivated by nearly-append-only data feeds that we have observed in the Darkstar warehouse, which are mostly stable except for occasional revisions. Finally, different levels of completeness, such as X-Percent-Full(V(t)) may also be useful --- the warehouse may maintain summary views whose results are acceptable as long as they summarize a sufficient fraction of the input.

The above types of consistency levels may be used to quantify and monitor data quality in a stream warehouse. For example, if the number of packages per partition is a fixed constant, we can track multiple X-Percent Closed and Full consistency lines for various values of X in order to understand the extent of missing and delayed data. Such consistency lines may also be useful for monitoring and debugging the warehouse update propagation algorithm. For example, if all the base tables are full, but recent derived table partitions are only 25 percent full, or just open, then perhaps the warehouse is spending too much time trying to keep up with the raw inputs rather than propagating updates through materialized views. We hope to report on a visual data quality tool based on consistency lines in future work.

Conventionally defined punctuations allow for group-wise processing, i.e., an assurance that all data within a group have arrived. Analogously, in some cases we might be able to provide group-wise consistency guarantees if we know that, e.g., all data from routers in the European region has arrived while we are still waiting for data from Southeast Asia routers. Propagating group-wise punctuation would require a more sophisticated analysis of the queries that define materialized views, e.g. that an aggregation query groups on the region column.

If we are willing to make increasingly detailed analyses of the queries that define tables, we can obtain a more refined and less restrictive set of consistency levels. Three additional types of consistency are:

- **NoNewRecords**: no records will be added to the partition in the future, but some existing records may be removed (this may happen in views with negation).
- **NoFieldChange**(K,F): If a record with key K exists in the partition, the value of fields K union F will not change in the future.
- **NoDeletedRecords**: no records will be deleted from this partition in the future, but new records may be added (this occurs in monotonic views).

A full description of how to analyze queries to apply these consistency levels is lengthy and detailed. However, we outline one type of query as an example. Suppose that table V is computed by outer-joining B to A, and the join predicate is from a foreign key on A to a primary key on B. Then NoNewRecords(V) depends on NoNewRecords(A) and NoFieldChange(A) only, not on table B.

5.1 Update Consistency in the Presence of Multiple Hierarchies

The discussion of update consistency resolution in Section 4.2 assumes that the collection of consistency levels form a hierarchy, e.g., Complete(V(t)) => Closed(V(t)) => Open(V(t)). However, a complex collection of consistency levels is likely to have many incomparable definitions. For example, from WeakClosed(V(t)) we cannot infer 100-Percent-Full(V(t)), nor vice versa. In this section we show how to resolve the update consistency of a table in a general setting.

Let **Cn** be the set of consistency classes available to the warehouse. We define a predicate **Stronger(C1,C2)**, C1 and C2 in Cn, if C1(V(t)) => C2(V(t)). We assume that the pair (Cn, Stronger) forms a directed acyclic graph, Gc, that includes all transitive edges.

Consistency classes such as Closed, in which some partitions might never reach the specified level of consistency, lead us to make additional definitions. For consistency level C in Cn, we define **Linear(C)** if C(V(t)) => C(V(t-k)) for $0 < k < t$. We also choose a default consistency level, C*default* in Cn, to be the update consistency level to be used if the update consistency resolution procedure returns an empty result.

As in Section 4.2, let M be the set of dependent table markings, along with the marking of table V. Let **Dep** the set of children of T in the data flow DAG. Then:

Update Consistency Resolution with hierarchies

1. Mark a node C in Cn if
   a. Linear(C), and C in M.
   b. Not Linear(C), C in M, and for each D in Dep, the update consistency of D is C.
2. Let U be the marked nodes {C1} in Cn such that there is no C2 in Cn such that Stronger(C1,C2).
3. If U is non-empty
   a. Return U
   b. Else return {C*default*}

Let U(V) be the set of update consistency levels returned by the update consistency resolution procedure. Then a partition V(t) is updated if we can infer consistency level Cu(V(T)) for some Cu in U(V).

We now present examples to illustrate update consistency inference with multiple hierarchies. Suppose that Cn={C1,…,C7}, and neither C5 nor C7 are linear (which we note with the double lined circle). Edges imply the Stronger relation (e.g. Stronger(C5,C3)), and we have removed transitive edges for clarity. In Figure 6, M = {C2, C3, C6}, so the result is that U = {C3, C3}. In Figure 7, M = {C5, C4}. However, not every S in Dep has update consistency C5 (as witnessed by the C4 marking}, and therefore we do not use C5 in the resolution procedure. Therefore U = {C4}.

6. RELATED WORK

The type of consistency we discuss in this paper relates to *temporal consistency* rather than transactional consistency. Data warehouses often use locking [9] or multi-version concurrency control [11][19] for the latter. However the method for implementing transactional consistency is orthogonal to the concerns of this paper.

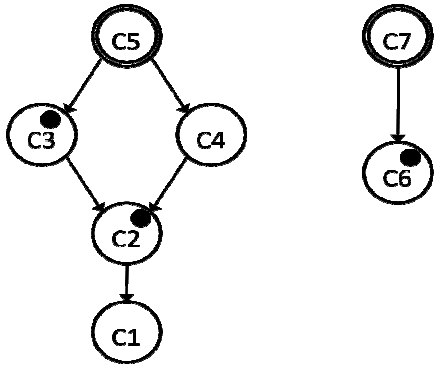

Figure 6. Update consistency resolution (a)

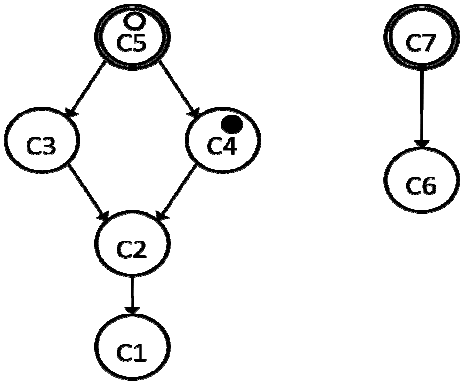

Figure 7. Update consistency resolution (b)

Materialized view maintenance in a data warehouse has an extensive literature; we summarize some key points below. The notion of temporal consistency in a data warehouse is generally taken to mean some type of *strong* consistency [25], e.g., that all materialized views are sourced from the same data, generally meaning that all views are updated in a single global pass. While some work has been done to allow for multiple consistency zones [6][24] using different consistency policies (e.g., immediate vs. deferred updates), any table belongs to a single zone and all tables in a zone are updated together. Even modern data warehousing systems are oriented towards batch updates [9]. The Real-Time community often defines a database as being consistent if it contains data representing a recent time interval, and mutually consistent if tables represent the same time intervals [12].

Temporal databases often use the *bitemporal model* [20]. Each record in a bitemporal database has a *valid time*, which refers to the time interval during which an event occurred, and a *transaction time*, which refers to the system clock time when a record is the most recent description of an event. However, the bitemporal model is not useful for much of the data in a stream warehouse (temporal metadata tables are a notable exception): the analyst is not concerned about transaction time, records in event feeds generally have many often conflicting timestamps and sequence numbers, and bitemporal databases do not enable update consistency.

Conventional DSMSs usually assume that data are in-order or nearly so, and they manage disorder by sorting or by punctuations and a limited degree of "out-of-order" processing [17]. Query operators can be assumed to have the most recent data, and consistency becomes a non-issue (but see [3], which manages stream consistency using revisions). We refer the interested reader to the more detailed discussion in Section 2 of [15].

Concepts similar to Closed consistency were discussed in [3], but that work assumed a specific tri-temporal data model and focused on handling revisions. Our consistency models only assume that each record has a timestamp whose value tends to increase over time, and they are oriented towards users' trust in query answers and efficient warehouse maintenance.

Another interesting comparison is between the two stream warehouse systems whose consistency management has been most fully described: DataDepot (in this paper and in [11]), and Truviso [10][15]. These two systems have approached stream warehousing from different angles: DataDepot adds stream processing to a conventional data warehouse, while Truviso adds warehousing capabilities to a stream system.

Truviso allows stream queries to reference conventional database tables, which can be updated during stream processing. Truviso uses *window consistency* [7] for these types of queries: the processing of stream S on window w has read-consistency on table T during the processing of w. To handle late-arriving data, Truviso computes window revisions [15], which can be thought of as the increments for self-maintaining views [18].

The consistency mechanisms described in this paper and those described for Truviso are orthogonal. DataDepot could benefit from window consistency (currently it uses temporal metadata tables such as W_METADATA in Figure 2). The window revisions are an optimized method for performing view maintenance, as compared to DataDepot's default of recomputing partitions affected by new data (Truviso falls back to recomputing affected windows for views that are non self-maintaining [15]). Thus, the consistency mechanism described in this paper applies to both systems.

7. CONCLUSIONS AND FUTURE WORK

We proposed mechanisms for managing and exploiting the consistency of materialized views in a stream warehouse. The first, *query consistency*, propagates consistency properties from base tables to materialized views, and provides consistency guarantees of query results. The second, *update consistency*, propagates table consistency requirements from materialized views to base tables, and is used to optimize the management of a stream warehouse. We focused on a most basic three types of consistency: Open, Closed, and Complete; however, as discussed in Section 5, many more useful consistency definitions can fit within our framework.

There are several issues not fully addressed in this paper. One issue is the handling of very late data, e.g. data that arrive long after the base table partitions have been marked Closed. These partitions, and all dependent partitions in dependent tables, need to be recomputed, but what is the best way to handle the revisions to the consistency markings? We have proposed the "trailing-edge line" as a convenient way to summarize stable data, but late arrivals poke holes in this line.

Another issue which is not fully addressed is the processing of query-specific consistency properties. Our basic models make

some use of query-specific handling, e.g. inner-join vs. outer-join range variables. However, query-specific consistency inference can become arbitrarily complex; experience will determine whether the complexity produces a tangible benefit.

All the examples in this paper assumed that recent data may be suspect, but they eventually stabilize over time. We are also interested in applying our consistency framework to data that begin as "exact" when loaded into the warehouse and then "decay" or lose accuracy over time. Examples include location data periodically collected from moving objects, and sensor measurements.

8. REFERENCES

[1] M. Ahuja, C. C. Chen, R.Gottapu, J. Hallmann, W. Hasan, R. Johnson, M. Kozyrczak, R. Pabbati, N. Pandit, S. Pokuri, and K. Uppala, Peta-scale data warehousing at Yahoo!, *Proc. of SIGMOD 2009*, 855-862.

[2] M. Balazinska, Y. Kwon, N. Kuchta, and D. Lee, Moirae: History-Enhanced Monitoring, *Proc. of CIDR 2007*, 275-286.

[3] R. Barga, J. Goldstein, M. Ali, and M. Hong, Consistent Streaming Through Time: A Vision for Event Stream Processing. *Proc. of CIDR 2007*, 363-374.

[4] I. Botan, R. Derekhshan, N. Dindar, L. Haas, R. Miller, and N. Tatbul. SECRET: A Model for Analysis of the Execution Semantics of Stream Processing Systems. *PVLDB* 3(1):232-243 (2010).

[5] R. Carney, U. Cetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker, Operator Scheduling in a Data Stream Manager, *Proc. of VLDB 2003*, 838-849.

[6] L. Colby, A. Kawaguchi, D. Lieuwen, I. S. Mumick, and K. Ross, Supporting Multiple View Maintenance Policies, *Proc. of SIGMOD 1997*, 405-416.

[7] N. Conway, Transactions and Data Stream Processing, http://neilconway.org/docs/thesis/pdf, April 2008.

[8] C. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk, Gigascope: A Stream Database for Network Applications, *Proc. of SIGMOD 2003*, 647-651.

[9] 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, *Proc. of VLDB 2005*, 1043-1054.

[10] M. Franklin, S. Krishnamurthy, N. Conway, A. Li, A. Russakovsky, and N. Thombre, Continuous Analytics: Rethinking Query Processing in a Network-Effect World, *Proc. of CIDR 2009*.

[11] L. Golab, T. Johnson, J. Seidel, and V. Shkapenyuk, Stream warehousing with DataDepot, *Proc. of SIGMOD 2009*, 847-854.

[12] A.K. Jha, M. Xiong, and K. Ramamritham. Mutual Consistency in Real-Time Databases. *Proc. of the 27th IEEE Real Time Systems Symposium (RTSS) 2006*, 335-343.

[13] T. Johnson, S. Muthukrishnan, V. Shkapenyuk, and O. Spatscheck, A Heartbeat Mechanism and Its Application in Gigascope, *Proc. of VLDB 2005*, 1079-1088.

[14] C. Kalmanek, Z. Ge, S. Lee, C. Lund, D. Pei, J. S. Seidel, K. Van der Merwe, and J. Yates, Darkstar: Using Exploratory Data Mining to Raise the Bar on Network Reliability and Performance, *Proc. of DRCN 2009*.

[15] S. Krishnamurthy, M. Franklin, J. Davis, D. Farina, P. Golovko, A. Li, and N. Thombre, Continuous analytics over discontinuous streams, *Proc. of SIGMOD 2010*, 1081-1092.

[16] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. Tucker, No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Record* 34(1): 39-44 (2005).

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

[18] I. Mumick, D. Quass, and B. Mumick, Maintenance of Data Cubes and Summary Tables in a Warehouse, *Proc. of SIGMOD 1997*, 100-111.

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

[20] R.T. Snodgrass. The TSQL2 Temporal Query Language, Kluwer 1995.

[21] P. Tucker, D. Maier, T. Sheard, and L. Fegaras, Exploiting Punctuation Semantics in Continuous Data Streams, *TKDE* 15(3): 555-568 (2003).

[22] K. Tufte, J. Li, D. Maier, V. Papadimos, R. Bertini, and J. Rucker, Travel time estimation using NiagaraST and latte, *Proc. of SIGMOD 2007*, 1091-1093.

[23] E. Welbourne, N. Khoussainova, J. Letchner, Y. Li, M. Balazinska, G. Borriello, and D. Suciu, Cascadia: a system for specifying, detecting, and managing RFID events, *Proc. of MobiSys 2008*, 281-294.

[24] Y. Zhuge, H. Garcia-Molina, and J. Wiener, Multiple View Consistency for Data Warehousing, *Proc. of ICDE 1997*, 289-300.

[25] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom, View Maintenance in a Warehousing Environment, *Proc. of SIGMOD 1995*: 316-327.