The Strobe Algorithms for Multi-Source Warehouse Consistency*

Yue Zhuge, Hector Garcia-Molina, and Janet L. Wiener

Computer Science Department
Stanford University
Stanford, CA 94305-2140, USA
{zhuge,hector,wiener}@cs.stanford.edu
http://db.stanford.edu/warehouse

Abstract

A warehouse is a data repository containing integrated information for efficient querying and analysis. Maintaining the consistency of warehouse data is challenging, especially if the data sources are autonomous and views of the data at the warehouse span multiple sources. Transactions containing multiple updates at one or more sources, e.g., batch updates, complicate the consistency problem. In this paper we identify and discuss three fundamental transaction processing scenarios for data warehousing. We define four levels of consistency for warehouse data and present a new family of algorithms, the Strobe family, that maintain consistency as the warehouse is updated, under the various warehousing scenarios. All of the algorithms are incremental and can handle a continuous and overlapping stream of updates from the sources. Our implementation shows that the algorithms are practical and realistic choices for a wide variety of update scenarios.

1 Introduction

A *data warehouse* is a repository of integrated information from distributed, autonomous, and possibly heterogeneous, sources. Figure 1 illustrates the basic warehouse architecture. At each source, a monitor collects the data of interest and sends it to the warehouse. The monitors are responsible for identifying changes in the source data, and notifying the warehouse. At the warehouse, the integrator receives the source data, performs any necessary data integration or translation, adds any extra desired information, such as timestamps for historical analysis, and tells the warehouse to store the data. In effect, the warehouse caches a materialized view of the source data[13]. The data is then readily available to user applications for querying and analysis.

Most current commercial warehousing systems (e.g., Prism, Redbrick) focus on storing the data for efficient access, and on providing extensive querying facilities at the warehouse. They ignore the complementary problem of consistently integrating new data, assuming that this happens "off line" while queries are not being run. Of course, they are discovering that many customers have international operations in multiple time zones, so there is no convenient down time, a "night" or "weekend" when all of the recent updates can be batched and processed together, and materialized views can be recomputed. Furthermore, as more and more updates occur, the down time window may no longer be sufficient to process all of the updates [7].

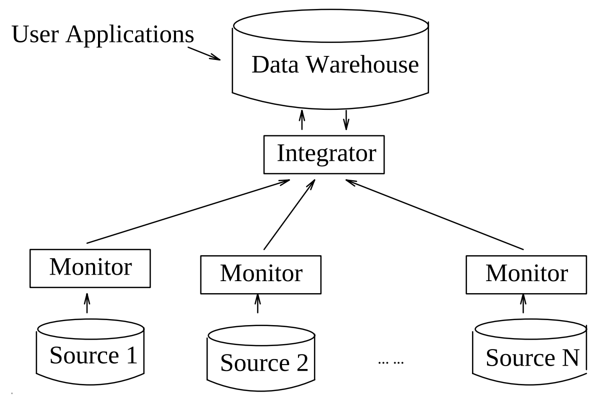

Figure 1: Data warehouse architecture

Thus, there is substantial interest in warehouses that can absorb incoming updates and incrementally modify the materialized views at the warehouse, without halting query processing. In this paper we focus on this process and on how to ensure that queries see consistent data. The crux of the problem is that each arriving update may need to be integrated with data from other sources before being stored at the warehouse. During this processing, more updates may arrive at the warehouse, causing the warehouse to become inconsistent.

The following example illustrates some of the inconsistencies that may arise. For simplicity, we assume that both the warehouse and the sources use the relational model, and that the materialized view kept at the warehouse contains the key for each par-

*This work was partially supported by Rome Laboratories under Air Force Contract F30602-94-C-0237 and by an equipment grant from Digital Equipment Corporation.

ticipating relation. In this example, each update is a separate transaction at one of the sources. We also assume that the integrator is tightly coupled with the warehouse. Therefore, although the view maintenance computation is done by the integrator, and the actual view operation is done by the warehouse, we use the term *warehouse* (*WH*) to denote the combination of the integrator and the warehouse in Figure 1.

Example 1: View maintenance anomaly over multiple sources

Let view V be defined as $V = r_1 \bowtie r_2 \bowtie r_3$, where r_1, r_2, r_3 are three relations residing on sources x, y and z, respectively. Initially, the relations are

$$r_1 : \begin{array}{cc} \text{A} & \text{B} \\ \hline 1 & 2 \end{array} \qquad r_2 : \begin{array}{cc} \text{B} & \text{C} \\ \hline - & - \end{array} \qquad r_3 : \begin{array}{cc} \text{C} & \text{D} \\ \hline 3 & 4 \end{array}$$

The materialized view at the warehouse is $MV = \emptyset$. We consider two source updates: $U_1 = insert(r_2, [2,3])$ and $U_2 = delete(r_1, [1,2])$. Using a conventional incremental view maintenance algorithm [2], the following events may occur at the *WH*.

1. The *WH* receives $U_1 = insert(r_2, [2,3])$ from source y. It generates query $Q_1 = r_1 \bowtie [2,3] \bowtie r_3$. To evaluate Q_1, the *WH* first sends query $Q_1^1 = r_1 \bowtie [2,3]$ to source x.
2. The *WH* receives $A_1^1 = [1,2,3]$ from source x. Query $Q_1^2 = [1,2,3] \bowtie r_3$ is sent to source z for evaluation.
3. The *WH* receives $U_2 = delete(r_1, [1,2])$ from source x. Since the current view is empty, no action is taken for this deletion.
4. The *WH* receives $A_1^2 = [1,2,3,4]$ from source z, which is the final answer for Q_1. Since there are no pending queries or updates, the answer is inserted into MV and $MV = [1,2,3,4]$. This final view is incorrect. □

In this example, the interleaving of query Q_1 with updates arriving from the sources causes the incorrect view. Note that even if the warehouse view is updated by completely recomputing the view — an approach taken by several commercial systems, such as Bull and Pyramid — the warehouse is subject to the same anomalies caused by the interleaving of updates with recomputation.

There are two straightforward ways to avoid this type of inconsistency, but we will argue that in general, neither one is desirable. The first way is to store copies of all relations at the warehouse. In our example, Q_1 could then be atomically evaluated at the warehouse, causing tuple $[1,2,3,4]$ to be added to MV. When U_2 arrives, the tuple is deleted from MV, yielding a correct final warehouse state. While this solution may be adequate for some applications, we believe it has several disadvantages. First, the storage requirement at the warehouse may be very high. For instance, suppose that r_3 contains data on companies, e.g., their name, stock price, and profit history. If we copy all of r_3 at the warehouse, we need to keep tuples for *all* companies that exist anywhere in the world, not just those we are currently interested in tracking. (If we do not keep data for all companies, in the future we may not be able to answer a query that refers to a new company, or a company we did not previously track, and be unable to atomically update the warehouse.) Second, the warehouse must integrate updates for all of the source data, not just the data of interest. In our company example, we would need to update the stock prices of all companies, as the prices change. This can represent a very high update load [4], much of it to data we may never need. Third, due to cost, copyright, or security, storing copies of all of the source data may not be feasible. For example, the source access charges may be proportional to the amount of data we track at the warehouse.

The second straightforward way to avoid inconsistencies is to run each update and all of the actions needed to incrementally integrate it into the warehouse as a distributed transaction spanning the warehouse and all the sources involved. In our example, if Q_1 runs as part of a distributed transaction, then it can read a consistent snapshot and properly update the warehouse. However, distributed transactions require a global concurrency control mechanism spanning all the sources, which may not exist. And even if it does, the sources may be unwilling to tolerate the delays that come with global concurrency control.

Instead, our approach is to make queries *appear* atomic by processing them intelligently at the warehouse (and without requiring warehouse copies of all relations). In our example, the warehouse notes that deletion U_2 arrived at the warehouse while it was processing query Q_1. Therefore, answer A_1 may contain some tuples that reflect the deleted r_1 tuple. Indeed, A_1 contains $[1,2,3,4]$, which should not exist after $[1,2]$ was deleted from r_1. Thus, the warehouse removes this tuple, leaving an empty answer. The materialized view is then left empty, which is the correct state after both updates take place. The above example gives the "flavor" of our solution; we will present more details as we explain our algorithms.

Note that the intelligent processing of updates at the warehouse depends on how and if sources run transactions. If some sources run transactions, then we need to treat their updates, whether they came from one source or multiple sources, as atomic units. Combining updates into atomic warehouse actions introduces additional complexities that will be handled by our algorithms. Since we do not wish to assume a particular transaction scenario, in this paper we cover the three main possibilities: sources run no transactions, some sources run local (but not global) transactions, and some sources run global transactions.

Although we are fairly broad in the transaction scenarios we consider, we do make two key simplifying assumptions: we assume that warehouse views are defined by relational project, select, join (PSJ) operations, and we assume that these views include the keys of all of the relations involved. We believe that PSJ views are the most common and therefore, it is a good subproblem on which to focus initially. We believe that requiring keys is a reasonable assumption, since keys make it easier for the applications to interpret and handle the warehouse data. Furthermore, if a user-specified view does not contain sufficient key information, the warehouse can simply add the key attributes to the view definition. (We have developed view maintenance algorithms for the case where some key data is not present, but they are not discussed here. They are substantially more complex than the ones presented here — another reason for including keys in the view.)

In our previous work [17] we considered a very restricted scenario: all warehouse data arrived from a single source. Even in that simple case, there are consistency problems, and we developed algorithms for solving them. However, in the more realistic multi-source scenario, it becomes *significantly* more complex to maintain consistent views. (For instance, the ECA and ECA-Key algorithms of [17] do not provide consistency in Example 1; they lead to the same incorrect execution shown.) In particular, the complexities not covered in our earlier work are as follows.

- An update from one source may need to be integrated with data from several other sources. However, gathering the data corresponding to one view update is not an atomic operation. No matter how fast the warehouse generates the appropriate query and sends it to the sources, receiving the answer is not atomic, because parts of it come from different, autonomous sources. Nonetheless, the view should be updated as if all of the sources were queried atomically.

- Individual sources may batch several updates into a single, source-local, transaction. For example, the warehouse may receive an entire day's updates in one transaction. These updates — after integration with data from other sources — should appear atomically at the warehouse. Furthermore, updates from several sources may together comprise one, global, transaction, which again must be handled atomically.

These complexities lead to substantially different solutions. In particular, the main contributions of this paper are:

1. We define and discuss all of the above update and transaction scenarios, which require increasingly complex algorithms.

2. We identify four levels of consistency for warehouse views defined on multiple sources, in increasing order of difficulty to guarantee. Note that as concurrent query and update processing at warehouses becomes more common, and as warehouse applications grow beyond "statistical analysis," there will be more concern from users about the consistency of the data they are accessing [7]. Thus, we believe it is important to offer customers a variety of consistency options and ways to enforce them.

3. We develop the Strobe family of algorithms to provide consistency for each of the transaction scenarios. We have implemented each of the Strobe algorithms in our warehouse prototype [16], demonstrating that the algorithms are practical and efficient.

4. We map out the space of warehouse maintenance algorithms (Figure 2). The algorithms we present in this paper provide a wide number of options for this consistency and distribution space.

The remainder of the paper is organized as follows. We discuss related work in Section 2. In Section 3, we define the three transaction scenarios and specify our assumptions about the order of messages and events in a warehouse environment. In Section 4 we define four levels of consistency and correctness, and discuss when each might be desirable. Then we describe our new algorithms in Section 5 and apply the algorithms to examples. We also demonstrate the levels of consistency that each algorithm achieves for the different transaction scenarios. In Section 6, we adapt the algorithms so that the warehouse can reflect every update individually, and show that the algorithms will terminate. We conclude in Section 7 by outlining optimizations to our algorithms and our future work.

2 Related research

The work we describe in this paper is closely related to research in three fields: data warehousing, data consistency and incremental maintenance of materialized views. We discuss each in turn.

Data warehouses are large repositories for analytical data, and have recently generated tremendous interest in industry. A general description of the data warehousing idea may be found in [11]. Companies such as Red Brick and Prism have built specialized data warehousing software, while almost all other database vendors, such as Sybase, Oracle and IBM, are targeting their existing products to data warehousing applications.

A warehouse holds a copy of the source data, so essentially we have a distributed database system with replicated data. However, because of the autonomy of the sources, traditional concurrency mechanisms are often not applicable [3]. A variety of concurrency control schemes have been suggested over the years for such environments. They either provide weaker notions of consistency, e.g., [6], or exploit the semantics of applications. The algorithms we present in this paper exploit the semantics of materialized view maintenance to obtain consistency without traditional distributed concurrency control. Furthermore, they offer a variety of consistency levels that are useful in the context of warehousing.

Many incremental view maintenance algorithms have been developed for centralized database systems, e.g., [2, 9, 5] and a good overview of materialized views and their maintenance can be found in [8]. Most of these solutions assume that a single system controls all of the base relations and understands the views and hence can intelligently monitor activities and compute all of the information that is needed for updating the views. As we showed in Example 1, when a centralized algorithm is applied to the warehouse, the warehouse user may see inconsistent views of the source data. These inconsistent views arise regardless of whether the centralized algorithm computes changes using the old base relations, as in [2], or using the new base relations, as in [5]. The crux of the warehouse problem is that the exact state of the base relations (old or new) when the incremental changes are computed at the sources is unknown, and our algorithms filter out or add in recent modifications dynamically.

Previous distributed algorithms for view maintenance, such as those in [14, 12], rely on timestamping the updated tuples. For a warehousing environment, sources can be legacy systems so we cannot assume that they will help by transmitting all necessary data or by attaching timestamps.

Hull and Zhou [10] provide a framework for supporting distributed data integration using materialized views. However, their approach first materializes each base relation (or relevant portion), then computes the view from the materialized copies; on the other hand, we propose algorithms to maintain joined views directly, without storing any auxiliary data. We compare our definition of consistency with theirs in Section 4. Another recent paper by Baralis, et al. [1] also uses timestamps to maintain materialized views at a warehouse. However, they assume that the warehouse never needs to query the sources for more data, hence circumventing all of the consistency problems that we address.

A warehouse often processes updates (from one or more transactions) in batch mode. Conventional algorithms have no way to ensure that an entire transaction is reflected in the view at the same time, or that a batch representing an entire day (or hour, or week, or minute) of updates is propagated to the view simultaneously. In this paper we present view maintenance algorithms that address these problems.

Finally, as we mentioned in Section 1, in [17] we showed how to provide consistency in a restricted single-source environment. Here we study the more general case of multiple sources and transactions that may span sources.

3 Warehouse transaction environment

The complexity of designing consistent warehouse algorithms is closely related to the scope of transactions at the sources. The larger the scope of a transaction, the more complex the algorithm becomes. In this section, we define three common transaction scenarios, in increasing order of complexity, and spell out our assumptions about the warehouse environment. In particular, we address the ordering of messages between sources and the warehouse, and define a source *event*. We use the relational model for simplicity; each update therefore consists of a single tuple action such as inserting or deleting a tuple.

3.1 Update transaction scenarios

The three transaction scenarios we consider in this paper are:

1. *Single update transactions.* Single update transactions are the simplest; each update comprises its own transaction and is reported to the warehouse separately. Actions of legacy systems that do not have transactions fall in this category: as each change is detected by the source monitor, it is sent to the warehouse as a single update transaction.

2. *Source-local transactions.* A source-local transaction is a sequence of actions performed at the same source that together comprise one transaction. The goal is therefore to reflect all of these actions atomically at the warehouse. We assume that each source has a local serialization schedule of all of its source-local transactions. Single update transactions are special cases of source-local transactions. Database sources, for example, are likely to have source-local transactions. We also consider batches of updates that are reported together to be a single, source-local, transaction.

3. *Global transactions.* In this scenario there are global transactions that contain actions performed at multiple sources. We assume that there is a global serialization order of the global transactions. (If there is not, it does not matter how we order the transactions at the warehouse.) The goal is therefore to reflect the global transactions atomically at the warehouse. Depending on how much information the warehouse receives about the transaction, this goal is more or less achievable. For example, unless there are global transaction identifiers, or the entire transaction is reported by a single source, the warehouse cannot tell which source-local transactions together comprise a global transaction.

For each transaction scenario, we make slightly different assumptions about the content of messages.

3.2 Messages

There are two types of messages from the sources to the warehouse: reporting an update and returning the answer to a query. There is only one type of message in the other direction; the warehouse may send queries to the sources.

We assume that each single update transaction and source-local transaction is reported in one message, at the time that the transaction commits. For example, a relational database source might trigger sending a message on transaction commit [15]. However, batching multiple transactions into the same message does not affect the algorithms of Section 5. For global transactions, updates can be delivered in a variety of ways. For example, the site that commits the transaction may collect all of the updates and send them to the warehouse at the commit point. As an alternative, each site may send its own updates, once it knows the global transaction has committed. In Section 5.4 we discuss the implications of the different schemes.

3.3 Event Ordering

Each source action, plus the resulting message sent to the warehouse, is considered one event. For example, evaluating a query at a source and sending the answer back to the warehouse is considered one event. We assume events are atomic, and are ordered by the sequence of the corresponding actions. (In [18] we discuss what to do when this assumption does not hold.) We also assume that any two messages sent from one source to the warehouse are delivered in the same order as they were sent. (This can be enforced by numbering messages.) We place no restrictions on the order in which messages sent from different sources to the warehouse are delivered.

3.4 Discussion

In practice, the update transaction scenario seen at the warehouse depends primarily on the capabilities of the underlying sources. For example, it is currently common practice to report updates from a source periodically. Instead of reporting each change, a monitor might send all of the changes that occurred over the last hour or day to the warehouse, as a single batch transaction. Periodic snapshots may be the only way for the monitor of an unsophisticated legacy source to report changes, or a monitor might choose to report updates lazily when the warehouse does not need to be kept strictly up to date.

In general, smarter monitors (those which help to group or classify updates or those which coordinate global transactions) save the warehouse processing and may enable the warehouse to achieve a higher level of consistency, as we will see in Section 5.4. We believe that today most warehouse transaction environments will support either single-update transactions or source-local transactions (or both), but will not have any communication or coordination between sources. Still, for completeness, we believe it is important to understand the global transaction scenario, which may be more likely in the future.

4 Correctness and consistency

Before describing our algorithms, we first define what it means for an algorithm to be correct in an environment where activity at the sources is decoupled from the view at the warehouse. In particular, we are concerned with what it means for a warehouse view to be consistent with the original source data. Since each source update may involve fetching data from multiple sources in order to update the warehouse view, we first define *states* at the sources and at the warehouse.

4.1 Source and warehouse states

Each warehouse state ws represents the contents of the warehouse. The warehouse state changes whenever the view is updated. Let the warehouse states be $ws_0, ws_1, ws_2, \ldots, ws_f$. (We assume there is a final warehouse state after all activity ceases.) We consider one view V at the warehouse, which is defined over a set of base relations at one or more sources. The view at state ws_j is $V(ws_j)$.

Let there be u sources, where each source has a unique id x ($1 \leq x \leq u$). A source state ss is a vector that contains u elements and represents the (visible) state of each source at a given instant in time. The x^{th} component, $ss[x]$, is the state of source x. Source states represent the contents of source base relations. We assume that source updates are executed is a serializable fashion across all sources, i.e., there is some serial schedule S that represents execution of the updates. (However, what constitutes a transaction varies according to the scenario.) We assume that ss_q is the final state after S completes. $V(ss)$ is the result of computing the view V over the source state ss. That is, for each relation r at source x that contributes to the view, $V(ss)$ is evaluated over r at the state $ss[x]$.

Each source transaction is guaranteed to bring the sources from one consistent state to another. For any serial schedule R, we use $\mathit{result}(R)$ to refer to the source state vector that results from its execution.

4.2 Levels of consistency

Assume that the view at the warehouse is initially synchronized with the source data, i.e., $V(ss_0) = V(ws_0)$. We define four levels of consistency for warehouse views. Each level subsumes all prior levels. These definitions are a generalization of the ones in [17] for a multi-source warehouse environment.

1. **Convergence:** For all finite executions, $V(ws_f) = V(ss_q)$. That is, after the last update and after all activity has ceased, the view is consistent with the source data.

2. **Weak consistency:** Convergence holds and, for all ws_i, there exists a source state vector ss_j such that $V(ws_i) = V(ss_j)$. Furthermore, for each source x, there exists a serial schedule $R = T_1, \ldots, T_k$ of (a subset of all) transactions such that $\mathit{result}(R)[x] = ss_j[x]$. That is, each warehouse state reflects a valid state at each source, and there is a locally serializable schedule at each source that achieves that state. However, each source may reflect a different serializable schedule and the warehouse may reflect a different set of committed transactions at each source.

3. **Strong consistency:** Convergence holds and there exists a serial schedule R and a mapping m, from warehouse states into source states, with the following properties: (i) Serial schedule R is equivalent to the actual execution of transactions at the sources. It defines a sequence of source states $ss_1, ss_2, \ldots$ where ss_j reflects the first j transactions (i.e., $ss_j = \mathit{result}(R')$ where R' is the R prefix with j transactions). (ii) For all ws_i, $m(ws_i) = ss_j$ for some j and $V[ws_i] = V[ss_j]$. (iii) If $ws_i < ws_k$, then $m(ws_i) < m(ws_k)$. That is, each warehouse state reflects a set of valid source states, reflecting the *same* globally serializable schedule, and the order of the warehouse states matches the order of source actions.

4. **Completeness:** In addition to strong consistency, for every ss_j defined by R, there exists a ws_i such that $m(ws_i) = ss_j$. That is, there is a complete order-preserving mapping between the states of the view and the states of the sources.

Hull and Zhou's definition of consistency for replicated data [10] is similar to our strong consistency, except that they also require global timestamps across sources, which we do not. Also, our strong consistency is less restrictive than theirs in that we do not require any fixed order between two non-conflicting actions. Our definition is compatible with standard serializability theory. In fact, our consistency definition can be rephrased in terms of serializability theory, by treating the warehouse view evaluation as a read only transaction at the sources [18].

Although completeness is a nice property since it states that the view "tracks" the base data exactly, we believe it may be too strong a requirement and unnecessary in most practical warehousing scenarios. In some cases, convergence may be sufficient, i.e., knowing that "eventually" the warehouse will have a valid state, even if it passes through intermediate states that are invalid. In most cases, strong consistency is desirable, i.e., knowing that every warehouse state is valid with respect to a source state. In the next section, we show that an algorithm may achieve different levels of consistency depending on the update transaction scenario to which it is applied.

5 Algorithms

In this section, we present the Strobe family of algorithms. The Strobe algorithms are named after strobe lights, because they periodically "freeze" the constantly changing sources into a consistent view at the warehouse. Each algorithm was designed to achieve a specific level of correctness for one of the three transaction processing scenarios. We discuss the algorithms in increasing level of complexity: the Strobe algorithm, which is the simplest, achieves strong consistency for single update transactions. The Transaction-Strobe algorithm achieves strong consistency for source-local transactions, and the Global-Strobe algorithm achieves strong consistency for global transactions. In Section 6 we present modifications to these algorithms that attain completeness for their respective transaction scenarios.

5.1 Terminology

First, we introduce the terminology that we use to describe the algorithms.

Definition: A view V at the warehouse over n relations is defined by a Project-Select-Join (PSJ) expression $V = \Pi_{proj}(\sigma_{cond}(r_1 \bowtie r_2 \bowtie \ldots \bowtie r_n))$. □

Any two relations may reside at the same or at different sources, and any relational algebra expression constructed with project, select, and join operations can be transformed into an equivalent expression of this form. Moreover, although we describe our algorithms for PSJ views, our ideas can be used to adapt any existing centralized view maintenance algorithm to a warehousing environment.

As we mentioned in the introduction, we assume that the projection list contains the key attributes for each relation. We expect most applications to require keys anyway, and if not, they can be added to the view by the warehouse.

When a view is defined over multiple sources, an update at one source is likely to initiate a multi-source query Q at the warehouse. Since we cannot assume that the sources will cooperate to answer Q, the warehouse must therefore decide where to send the query first.

Definition: Suppose we are given a query Q that needs to be evaluated. The function $next\_source(Q)$ returns the pair (x, Q^i) where x is the next source to contact, and Q^i is the portion of Q that can be evaluated at x. If Q does not need to be evaluated further, then x is nil. A^i is the answer received at the warehouse in response to subquery Q^i. Query $Q\langle A^i\rangle$ denotes the remaining query after answer A^i has been incorporated into query Q. □

For PSJ queries, *next_source* will always choose a source containing a relation that can be joined with the known part of the query, rather than requiring the source to ship the entire base relation to the warehouse (which may not even be possible). As we will see later, queries generated by an algorithm can also be unions of PSJ expressions. For such queries, *next_source* simply selects one of the expressions for evaluation. An improvement would be to find common subexpressions.

Example 2: Using *next_source*

Let relations r_1, r_2, r_3 reside at sources x, y, z, respectively, let $V = r_1 \bowtie r_2 \bowtie r_3$, and let U_2 be an update to relation r_2 received at the warehouse. Therefore, query $Q = (r_1 \bowtie U_2 \bowtie r_3)$, and $next\_source(Q) = (x, Q^1 = r_1 \bowtie U_2)$. When the warehouse receives answer A^1 from x, $Q\langle A^1\rangle = A^1 \bowtie r_3$. Then $next\_source(A^1 \bowtie r_3) = (z, Q^2 = A^1 \bowtie r_3)$, since there is only one relation left to join in the query. A^2 is the final answer. □

In the above example, the query was sent to source x first. Alternatively, $next\_source(Q) = (z, U_2 \bowtie r_3)$. When there is more than one possible relation to join with the intermediate result, *next_source* may use statistics (such as those used by query optimizers) to decide which part of the query to evaluate next.

We are now ready to define the procedure *source_evaluate*, which loops to compute the next portion of query Q until the final result answer A is received. In the procedure, WQ is the "working query" portion of query Q, i.e., the part of Q that has not yet been evaluated.

Procedure $source\_evaluate(Q)$
  i = 0; $WQ = Q$; $A^0 = Q$;
  $(x, Q^1) \leftarrow next\_source(WQ)$;
  While x is not nil do
    — Let i = i+1;
    — Send Q^i to source x;
    — When x returns A^i, let $WQ = WQ\langle A^i\rangle$;
    — Let $(x, Q^{i+1}) \leftarrow next\_source(WQ)$;
  Return(A^i).
End Procedure

The procedure $source\_evaluate(Q)$ may return an incorrect answer when there are concurrent transactions at the sources that interfere with the query evaluation. For example, in example 1, we saw that a delete that occurs at a source after a subquery has been evaluated there, but before the final answer is computed, may be skipped in the final query result. More subtle problems result when two subqueries of the same query are sent to the same source for evaluation at different times (to join with different relations) and use different source states, or when two subqueries are evaluated at two different sources in states that are inconsistent with each other. The key idea behind the Strobe algorithms is to keep track of the updates that occur during query evaluation, and to later compensate. We introduce the Strobe family with the basic Strobe algorithm.

For simplicity, here we only consider insertions and deletions in our algorithms. Conceptually, modifications of tuples (updates sent to the warehouse) can be treated at the warehouse simply as a deletion of the old tuple followed by an insertion of the new tuple. However, for consistency and performance, the delete and the insert should be handled "at the same time." Our algorithms can be easily extended for this type of processing, but we do not do it here. Further discussion of how to treat a modification as an insert and a delete may be found in [8].

5.2 Strobe

The Strobe algorithm processes updates as they arrive, sending queries to the sources when necessary. However, the updates are not performed immediately on the materialized view MV; instead, we generate a list of actions AL to be performed on the view. We update MV only when we are sure that applying all of the actions in AL (as a single transaction at the warehouse) will bring the view to a consistent state. This occurs when there are no outstanding queries and all received updates have been processed.

When the warehouse receives a deletion, it generates a delete action for the corresponding tuples (with matching key values) in MV. When an insert arrives, the warehouse may need to generate and process a query, using procedure $source\_evaluate()$. While a Q query is being answered by the sources, updates may arrive at the warehouse, and the answer obtained may have missed their effects. To compensate, we keep a set $pending(Q)$ of the updates that occur while Q is being processed. After Q's answer is fully compensated, an insert action for MV is generated and placed on the action list AL.

Definition: The unanswered query set UQS is the set of all queries that the warehouse has sent to some source but for which it has not yet received an answer. □

Definition: The operation $key\_delete(R, U_i)$ deletes from relation R the tuples whose key attributes have the same values as U_i. □

Definition: $V\langle U\rangle$ denotes the view expression V with the tuple U substituted for U's relation. □

Algorithm 1: Strobe algorithm

At each source:

- ▷ After executing update U_i, send U_i to the warehouse.
- ▷ Upon receipt of query Q_i, compute the answer A_i over $ss[x]$ (the current source state), and send A_i to the warehouse.

At the warehouse:

- ▷ Initially, AL is set to empty $\langle\ \rangle$.
- ▷ Upon receipt of update U_i:
  - ∘ If U_i is a deletion
    - — $\forall Q_j \in UQS$ add U_i to $pending(Q_j)$;
    - — Add $key\_delete(MV, U_i)$ to AL.
  - ∘ If U_i is an insertion
    - — Let $Q_i = V\langle U_i\rangle$ and set $pending(Q_i) = \emptyset$;
    - — Let $A_i = source\_evaluate(Q_i)$;
    - — $\forall U_j \in pending(Q_i)$, apply $key\_delete(A_i, U_j)$;
    - — Add insert(MV,A_i) to AL.
- ▷ When $UQS = \emptyset$, apply AL to MV as a single transaction, without adding duplicate tuples to MV. Reset $AL = \langle\ \rangle$.

End Algorithm 1

The following example applies the Strobe algorithm to the warehouse scenario in Example 1 in the introduction. Specifically, it shows why a deletion needs to be applied to the answer of a previous query, when the previous query's answer arrives at the warehouse later than the deletion.

Example 3: Strobe avoids deletion anomaly

As in example 1, let view V be defined as $V = r_1 \bowtie r_2 \bowtie r_3$, where r_1, r_2, r_3 are three relations residing on sources x, y and z, respectively. Initially, the relations are

r_1:

A	B
1	2

r_2:

B	C
-	-

r_3:

C	D
3	4

The materialized view $MV = \emptyset$. We again consider two source updates: $U_1 = insert(r_2, [2,3])$ and $U_2 = delete(r_1, [1,2])$, and apply the Strobe algorithm.

1. $AL = \langle\ \rangle$. The WH receives $U_1 = insert(r_2, [2,3])$ from source y. It generates query $Q_1 = r_1 \bowtie [2,3] \bowtie r_3$. To evaluate Q_1, the WH first sends query $Q_1^1 = r_1 \bowtie [2,3]$ to source x.
2. The WH receives $A_1^1 = [1,2,3]$ from source x. Query $Q_1^2 = [1,2,3] \bowtie r_3$ is sent to source z for evaluation.
3. The WH receives $U_2 = delete(r_1, [1,2])$ from source x. It first adds U_2 to $pending(Q_1)$ and then adds $key\_delete(MV, U_2)$ to AL. The resulting $AL = \langle key\_delete(MV, U_2)\rangle$.
4. The WH receives $A_1^2 = [1,2,3,4]$ from source z. Since $pending(Q)$ is not empty, the WH applies

$key\_delete(A_1^2, U_2)$ and the resulting answer $A_2 = \emptyset$. Therefore, nothing is added to AL. There are no pending queries, so the WH updates MV by applying $AL = \langle key\_delete(MV, U_2)\rangle$. The resulting $MV = \emptyset$. The final view is correct and strongly consistent with the source relations. □

This example demonstrates how Strobe avoids the anomaly that caused both ECA-key and conventional view maintenance algorithms to be incorrect: by remembering the delete until the end of the query, Strobe is able to correctly apply it to the query result *before* updating the view MV. If the deletion U_2 were received before Q_1^1 had been sent to source x, then A_1^1 would have been empty and no extra action would have been necessary.

The Strobe algorithm provides strong consistency for all single-update transaction environments. A correctness proof is given in [18]. The intuition is that each time MV is modified, updates have quiesced and the view contents can be obtained by evaluating the view expression at the current source states. Therefore, although not all source states will be reflected in the view, the view always reflects a consistent set of source states.

5.3 Transaction-Strobe

The Transaction-Strobe (T-Strobe) algorithm adapts the Strobe algorithm to provide strong consistency for source-local transactions. T-Strobe collects all of the updates performed by one transaction and processes these updates as a single unit. Batching the updates of a transaction not only makes it easier to enforce consistency, but also reduces the number of query messages that must be sent to and from the sources.

Definition: $UL(T)$ is the *update list* of a transaction T. $UL(T)$ contains the inserts and deletes performed by T, in order. $IL(T) \subseteq UL(T)$ is the *insertion list* of T; it contains all of the insertions performed by T. □

Definition: $key(U_i)$ denotes the key attributes of the inserted or deleted tuple U_i. If $key(U_i) = key(U_j)$ then U_i and U_j denote the same tuple (although other attributes may have been modified). □

The source actions in T-Strobe are the same as in Strobe; we therefore present only the warehouse actions. First, the WH removes all pairs of insertions and deletions such that the same tuple was first inserted and then deleted. This removal is an optimization that avoids sending out a query for the insertion, only to later delete the answer. Next the WH adds all remaining deletions to the action list AL. Finally, the WH generates one query for all of the insertions. As before, deletions which arrive at the WH after the query is generated are subtracted from the query result.

The following example demonstrates that the Strobe algorithm may only achieve convergence, while the T-Strobe algorithm guarantees strong consistency for source-local transactions. Because the Strobe algorithm does not understand transactions, it may provide a view which corresponds to the "middle" of a transaction at a source state. However, Strobe will eventually provide the correct view, once the transaction commits, and is therefore convergent.

Algorithm 2: Transaction-Strobe algorithm
At the warehouse:
- ▷ Initially, $AL = \langle\ \rangle$.
- ▷ Upon receipt of $UL(T_i)$ for a transaction T_i:
  - ∘ For each $U_j, U_k \in UL(T_i)$ such that U_j is an insertion, U_k is a deletion, $U_j < U_k$ and $key(U_j) = key(U_k)$, remove both U_j and U_k from $UL(T_i)$.
  - ∘ For every deletion $U \in UL(T_i)$:
    - — $\forall Q_j \in UQS$, add U to $pending(Q_j)$.
    - — Add $key\_delete(MV, U)$ to AL.
  - ∘ Let $Q_i = \bigcup_{U_j \in IL(T)} V\langle U_j\rangle$, and set $pending(Q_i) = \emptyset$;
  - ∘ Let $A_i = source\_evaluate(Q_i)$;
  - ∘ $\forall U \in pending(Q_i)$, apply $key\_delete(A_i, U)$;
  - ∘ Add $insert(MV, A_i)$ to AL.
- ▷ When $UQS = \emptyset$, apply AL to MV, without adding duplicate tuples to MV. Reset $AL = \langle\ \rangle$.

End Algorithm 2

Example 4: T-Strobe provides stronger consistency than Strobe

Consider a simple view over one source defined as $V = r_1$. Assume attribute A is the key of relation r_1. Originally, the relation is: $r_1 = \frac{\text{A} \quad \text{B}}{1 \quad 2}$.

Initially $MV = ([1,2])$. We consider one source transaction: $T_1 = \langle delete(r_1, [1,2]), insert(r_1, [3,4])\rangle$.

When the Strobe algorithm is applied to this scenario, the warehouse firsts adds the deletion to AL. Since there are no pending updates, AL is applied to MV and MV is updated to $MV = \emptyset$, which is not consistent with r_1 either before or after T_1. Then the warehouse processes the insertion and updates MV again, to the correct view $MV = ([3,4])$.

The T-Strobe algorithm, on the other hand, only updates MV after both updates in the transaction have been processed. Therefore, MV is updated directly to the correct view, $MV = ([3,4])$. □

The T-Strobe algorithm is inherently strongly consistent with respect to the source states defined after each source-local transaction.[1] T-Strobe can also pro-

[1]Note incidentally that if modifications are treated as a delete-insert pair, then T-Strobe can process the pair within a single transaction, easily avoiding inconsistencies. However, for performance reasons we may still want to modify T-Strobe to handle modifications as a third type of action processed at the

cess batched updates, not necessarily generated by the same transaction, but which were sent to the warehouse at the same time from the same source. In this case, T-Strobe also guarantees strong consistency if we define consistent source states to be those corresponding to the batching points at sources. Since it is common practice today to send updates from the sources periodically in batches, we believe that T-Strobe is probably the most useful algorithm. On single-update transactions, T-Strobe reduces to the Strobe algorithm.

5.4 Global-strobe

While the T-Strobe algorithm is strongly consistent for source-local transactions, it is only weakly consistent if global transactions are present. In [18] we present an example that illustrates this and develop a new algorithm, Global-Strobe (G-Strobe), that guarantees strong consistency for global transactions. G-Strobe is the same as T-Strobe except that it only updates MV (with the actions in AL) when the following three conditions have all been met. (T-Strobe only requires condition 1). Let TT be the set of transaction identifiers that the warehouse has received since it last updated MV.

1. $UQS = \emptyset$;
2. For each transaction T_i in TT that depends (in the concurrency control sense) on another transaction T_j, T_j is also in TT; and
3. All of the updates of the transactions in TT have been received and processed.

Due to space limitations, we do not present G-Strobe here.

6 Completeness and termination of the algorithms

A problem with Strobe, T-Strobe, and G-Strobe is that if there are continuous source updates, the algorithms may not reach a quiescent state where UQS is empty and the materialized view MV can be updated. To address this problem, in this section we present an algorithm, Complete Strobe (C-Strobe) that can update MV after any source update. For example, C-strobe can propagate updates to MV after a particular batch of updates has been received, or after some long period of time has gone by without a natural quiescent point. For simplicity, we will describe C-strobe enforcing an update to MV after each update; in this case, C-strobe achieves completeness. The extension to update MV after an arbitrary number of updates is straightforward and enforces strong consistency.

To force an update to MV after update U_i arrives at the warehouse, we need to compute the resulting view. However, other concurrent updates at the sources complicate the problem. In particular, consider the case where U_i is an insertion. To compute the next MV state, the warehouse sends a query Q_i to the sources. By the time the answer A_i arrives, the warehouse may have received (but not processed) updates $U_{i+1}...U_k$. Answer A_i may reflect the effects of these later updates, so before it can use A_i to update MV, the warehouse must "subtract out" the effects of later updates from A_i, or else it will not get a consistent state. If one of the later updates, say U_j, is an insert, then it can just remove the corresponding tuples from A_i. However, if U_j is a delete, the warehouse may need to *add* tuples to A_i, but to compute these missing tuples, it must send additional queries to the sources! When the answers to these additional queries arrive at the warehouse, they may also have to be adjusted for updates they saw but which should not be reflected in MV. Fortunately, as we show below, the process does converge, and eventually the warehouse is able to compute the consistent MV state that follows U_i. After it updates MV, the warehouse then processes U_{i+1} in the same fashion.

Before presenting the algorithm, we need a few definitions.

Definition: $Q_{i,\_,\_}$ denotes the set of queries sent by the warehouse to compute the view after insertion update U_i. $Q_{i,j,\_}$ are the queries sent in response to update U_j that occurred while computing the answer for a query in $Q_{i,\_,\_}$. A unique integer k is used to distinguish each query in $Q_{i,j,\_}$ as $Q_{i,j,k}$. □

In the scenario above, for insert U_i we first generate $Q_{i,i,0}$. When its answer $A_{i,i,0}$ arrives, a deletion U_j received before $A_{i,i,0}$ requires us to send out another query, identified as Q_{i,j,new_j}. In the algorithm, new_j is used to generate the next unique integer for queries caused by U_j in the context of processing U_i.

When processing each update U_i separately, no action list AL is necessary. In the Strobe and T-strobe algorithms, AL keeps track of multiple updates whose processing overlaps. In the C-strobe algorithm outlined below, each update is compensated for subsequent, "held," updates so that it can be applied directly to the view. If C-strobe is extended (not shown here) to only force updates to MV periodically, after a batch of overlapping updates, then an action list AL is again necessary to remember the actions that should be applied for the entire batch.

Definition: $Q\langle U_i\rangle$ is the resulting query after the updated tuple in U_i replaces its base relation in Q. If the base relation of U_i does not appear in Q, then $Q\langle U_i\rangle = \emptyset$. □

Definition: $Delta$ is the set of changes that need to be applied to MV for one insertion update. Note that $Delta$, when computed, would correspond to a single $insert(MV, Delta)$ action on AL if we kept an action list. (Deletion updates can be applied directly to MV, but insertions must be compensated first. $Delta$ collects the compensations.) □

We also use a slightly different version of key_delete: $key\_delete^*(Delta, U_k)$ only deletes from $Delta$ those tuples that match with U_k on both key and non-key

warehouse. As stated earlier, we do not describe this straightforward extension here.

attributes (not just on key attributes). Finally, when we add tuples to *Delta*, we allow tuples with the same key values but different non-key values to be added. These tuples violate the key condition, but only appear in *Delta* temporarily. However, it is important to keep them in *Delta* for the algorithm to work correctly. (The reason for these changes is that when we "subtract out" the updates seen by $Q_{i,i,0}$, we first compensate for deletes, and then for all inserts. In between, we may have two tuples with the same key, one added from the compensation of a delete, and the other to be deleted when we compensate for inserts.)

In algorithm C-Strobe, the source behavior remains the same as for the Strobe algorithm, so we only describe the actions at the warehouse. C-Strobe is complete because MV is updated once after each update, and the resulting warehouse state corresponds to the source state after the same update. We prove the correctness of C-Strobe in [18].

Algorithm 3: Complete Strobe

At the warehouse:

- ▷ Initially, $Delta = \emptyset$.
- ▷ As updates arrive, they are placed in a holding queue.
- ▷ Process each update U_i in order of arrival:
  - ∘ If U_i is a deletion
    - — Apply $key\_delete(MV, U_i)$.
  - ∘ If U_i is an insertion
    - — Let $Q_{i,i,0} = V\langle U_i \rangle$;
    - — Let $A_{i,i,0} = source\_evaluate(Q_{i,i,0})$;
    - — Repeat for each $A_{i,j,k}$ until $UQS = \emptyset$:
      - ∘ Add $A_{i,j,k}$ to *Delta* (without adding duplicate tuples).
      - ∘ For all deletions U_p received between U_j and $A_{i,j,k}$:
        - — Let $Q_{i,p,new_p} = Q_{i,j,k}\langle U_p \rangle$;
        - — Let $A_{i,p,new_p} = source\_evaluate(Q_{i,p,new_p})$; When answer arrives, process starting 4 lines above.
    - — For all insertions U_k received between U_i and the last answer, if $\neg\exists U_j < U_k$ such that U_j is a deletion and U_j, U_k refer to the same tuple, then apply $key\_delete^*(Delta, U_k)$.
    - — Let $MV = MV + Delta$ and $Delta = \emptyset$.

End Algorithm 3

The compensating process (the loop in the algorithm) always terminates because any expression $Q_{i,j,k}\langle U_p \rangle$ always has one fewer base relation than $Q_{i,j,k}$. Let us assume that there are at most K updates that can arrive between the time a query is sent out and its answer is received, and that there are n base relations. When we process insertion U_i we send out query $Q_{i,i,0}$; when we get its answer we may have to send out at most K compensating queries with $n-2$ base relations each. For each of those queries, at most K queries with $n-3$ base relations may be sent, and so on. Thus, the total number of queries sent in the loop is no more than K^{n-2}, and the algorithm eventually finishes processing U_i and updates MV.

The number of compensating queries may be significantly reduced by combining related queries. For example, when we compensate for $Q_{i,i,0}$, the above algorithm sends out up to K queries. However, since there are only n base relations, we can group these queries into $n-1$ queries, where each combined query groups all of the queries generated by an update to the same base relation. If we continue to group queries by base relation, we see that the total number of compensating queries cannot exceed $(n-1)\times(n-2)\times\ldots\times 1 = (n-1)!$. That is, C-Strobe will update MV after at most $(n-1)!$ queries are evaluated. If the view involves a small number of relations, then this bound will be relatively small. Of course, this maximum number of queries only occurs under extreme conditions where there is a continuous stream of updates.

We now apply the C-Strobe algorithm to the warehouse scenario in Example 1, and show how C-Strobe processes this scenario differently from the Strobe algorithm (shown in Example 3).

Example 5: Complete Strobe

As in examples 1 and 3, let view V be defined as $V = r_1 \bowtie r_2 \bowtie r_3$, where r_1, r_2, r_3 are three relations residing on sources x, y and z, respectively. Initially, the relations are

r_1:

A	B
1	2

r_2:

B	C
-	-

r_3:

C	D
3	4

The materialized view $MV = \emptyset$. We again consider two source updates: $U_1 = insert(r_2, [2,3])$ and $U_2 = delete(r_1, [1,2])$, and apply the C-Strobe algorithm. There are two possible orderings of events at the warehouse. Here we consider one, and in the next example we discuss the other.

1. $Delta = \emptyset$. The *WH* receives from source y $U_1 = insert(r_2, [2,3])$. It generates query $Q_{1,1,0} = r_1 \bowtie [2,3] \bowtie r_3$. To evaluate $Q_{1,1,0}$, the *WH* first sends query $Q^1_{1,1,0} = r_1 \bowtie [2,3]$ to source x.
2. The *WH* receives $A^1_{1,1,0} = [1,2,3]$ from source x. Query $Q^2_{1,1,0} = [1,2,3] \bowtie r_3$ is sent to source z for evaluation.
3. The *WH* receives $U_2 = delete(r_1, [1,2])$ from source x. It saves this update in a queue.
4. The *WH* receives $A_{1,1,0} = A^2_{1,1,0} = ([1,2,3,4])$ from source z, which is the final answer to $Q_{1,1,0}$. Since U_2 was received between $Q_{1,1,0}$ and $A_{1,1,0}$ and it is a deletion, the *WH* generates a query $Q_{1,2,1} = [1,2] \bowtie [2,3] \bowtie r_3$ and sends it to source z. Also, it adds $A_{1,1,0}$ to *Delta*, so $Delta = ([1,2,3,4])$.
5. The *WH* receives $A_{1,2,1} = ([1,2,3,4])$ and tries to add it to *Delta*. Since it is a duplicate tuple, *Delta* remains the same.

6. $UQS = \emptyset$, so the *WH* updates the view to $MV = MV + Delta = ([1,2,3,4])$.
7. Next the *WH* processes U_2 which is next in the update queue. Since U_2 is a deletion, it applies $key\_delete^*(MV, U_2)$ and $MV = \emptyset$. □

In this example, MV is updated twice, in steps 6 and 7. After step 6, MV is equal to the result of evaluating V after U_1 but before U_2 occurs. Similarly, after step 7, MV corresponds to evaluating V after U_2, but before any further updates occur, which is the final source state in this example. In the next example we consider the case where U_2 occurs before the evaluation of the query corresponding to U_1, and we show that compensating queries are necessary.

Example 6: C-Strobe applied again, with different timing of the updates

Let the view definition, initial base relations and source updates be the same as in example 5. We now consider a different set of events at the *WH*.

1. $Delta = \emptyset$. The *WH* receives from source y $U_1 = insert(r_2, [2,3])$. It generates query $Q_{1,1,0} = r_1 \bowtie [2,3] \bowtie r_3$. To evaluate $Q_{1,1,0}$, the *WH* first sends query $Q^1_{1,1,0} = r_1 \bowtie [2,3]$ to source x.
2. The *WH* receives $U_2 = delete(r_1, [1,2])$ from source x. It saves this update in a queue.
3. The *WH* receives $A^1_{1,1,0} = \emptyset$ from source x. This implies that $A_{1,1,0} = \emptyset$. Since U_2 was received between $Q_{1,1,0}$ and $A_{1,1,0}$, the *WH* generates the compensating query $Q_{1,2,1} = [1,2] \bowtie [2,3] \bowtie r_3$ and sends it to source z. Also, it adds $A_{1,1,0}$ to *Delta* and *Delta* is still empty.
4. The *WH* receives $A_{1,2,1} = ([1,2,3,4])$ and adds it to *Delta*. $Delta = ([1,2,3,4])$.
5. Since $UQS = \emptyset$, the *WH* updates the view to $MV = MV + Delta = ([1,2,3,4])$.
6. The *WH* processes U_2. Since U_2 is a deletion, it applies $key\_delete^*(MV, U_2)$ and $MV = \emptyset$. □

As mentioned earlier, C-Strobe can be extended to update MV periodically, after processing every k updates. In this case, we periodically stop processing updates (placing them in a holding queue). We then process the answers to all queries that are in UQS as we did in C-Strobe, and then apply the action list AL to the view MV. The T-Strobe algorithm can also be made complete or periodic in a similar way. We call this algorithm C-TStrobe, but do not describe it here further.

7 Conclusions

In this paper, we identified three fundamental transaction processing scenarios for data warehousing and developed the Strobe family of algorithms to consistently maintain the warehouse data. Figure 2 summarizes the algorithms we discussed in this paper and their correctness. In the figure, "Conventional" refers to a conventional centralized view maintenance algorithm, while "ECA" and "ECA-Key" are algorithms from [17].

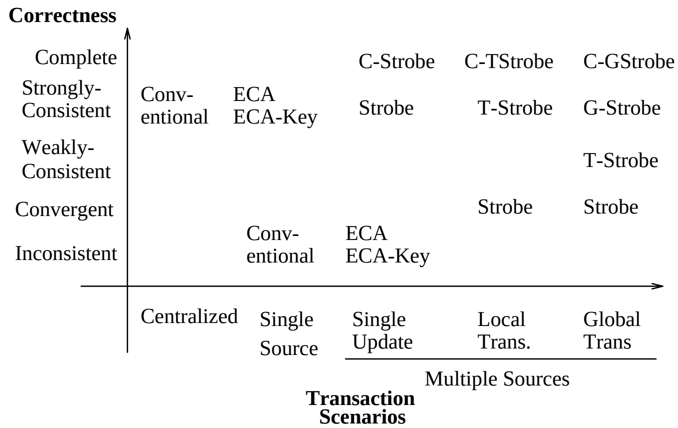

Figure 2: Consistency Spectrum

In Figure 2, an algorithm is shown in a particular scenario S and level of consistency L if it achieves L consistency in scenario S. Furthermore, the algorithm at (S, L) also achieves all lower levels of consistency for S, and achieves L consistency for scenarios that are less restrictive than S (scenarios to the left of S). For example, Strobe is strongly consistent for single update transactions at multiple sources. Therefore, it is weakly consistent and convergent (by definition) in that scenario. Similarly, Strobe is strongly consistent for centralized and single source scenarios.

Regarding the efficiency of the algorithms we have presented, there are four important points to make. First, there are a variety of enhancements that can improve efficiency substantially:

1. We can optimize global query evaluation. For example, in procedure $source\_evaluate()$, the warehouse can group all queries for one source into one, or can find an order of sources that minimizes data transfers. It can also use key information to avoid sending some queries to sources.
2. We can find the optimal batch size for processing. By batching together updates, we can reduce the message traffic to and from sources. However, delaying update processing means the warehouse view will not be as up to date, so there is a clear tradeoff that we would like to explore.
3. Although we argued against keeping copies of *all* base relations at the warehouse, it may make sense to copy the most frequently accessed ones (or portions thereof), if they are not too large or expensive to keep up to date. This also increases the number of queries that can be answered locally.

The second point regarding efficiency is that, even if someone determines that none of these algorithms is efficient enough for their application, it is still very important to understand the tradeoffs involved. The Strobe algorithms exemplify the inherent cost of keeping a warehouse consistent. Given these costs, users can now determine what is best for them, given their consistency requirements and their transactional scenario.

Third, when updates arrive infrequently at the warehouse, or only in periodic batches with large gaps in between, the Strobe algorithms are as efficient as conventional algorithms such as [2]. They only introduce extra complexity when updates must be processed while other updates are arriving at the warehouse, which is when conventional algorithms cannot guarantee a consistent view.

Fourth, the Strobe algorithms are relatively inexpensive to implement, and we have incorporated them into the Whips (WareHousing Information Prototype at Stanford) prototype [16]. In our implementation, the Strobe algorithm is only 50 more lines of C++ code than the conventional view maintenance algorithm, and C-strobe is only another 50 lines of code. The core of each of the algorithms is about 400 lines of C++ code (not including evaluating each query). The ability to guarantee correctness (Strobe), the ability to batch transactions, and the ability to update the view consistently, whenever desired and without quiescing updates (C-strobe) cost only approximately 100 lines of code, and one programmer day.

As part of our ongoing warehousing work, we are currently evaluating the performance of the Strobe and T-Strobe algorithms and considering some of the optimizations mentioned above. We are also extending the algorithms to handle more general type of views, for example, views with insufficient key information, and views defined by more complex relational algebra expressions. Our future work includes designing maintenance algorithms that coordinate updates to multiple warehouse views.

Acknowledgments

We would like to thank Jennifer Widom and Jose Blakely for discussions that led to some of the ideas in this paper.

References

[1] E. Baralis, S. Ceri, and S. Paraboschi. Conservative timestamp revised for materialized view maintenance in a data warehouse. In *The Workshop on Materialized Views*, pages 1–9, June 1996.

[2] J. Blakeley, P.-A. Larson, and F. Tompa. Efficiently updating materialized views. In *SIGMOD*, pages 61–71, June 1986.

[3] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of multidatabase transaction management. *VLDB Journal*, 1(2):181–239, Oct. 1992.

[4] M. Cochinwala and J. Bradley. A multidatabase system for tracking and retrieval of financial data. In *VLDB*, pages 714–721, 1994.

[5] L. Colby, T. Griffin, L. Libkin, I. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *SIGMOD*, pages 469–480, June 1996.

[6] R. Gallersdorfer and M. Nicola. Improving performance in replicated databases through relaxed coherency. In *VLDB*, pages 445–456, Sept. 1995.

[7] R. Goldring and B. Hamel, Jan. 1996. Personal correspondence about IBM's data warehouse customer needs.

[8] A. Gupta and I. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Engineering Bulletin*, 18(2):3–18, June 1995.

[9] A. Gupta, I. Mumick, and V. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, pages 157–166, May 1993.

[10] R. Hull and G. Zhou. A framework for supporting data integration using the materialized and virtual approaches. In *SIGMOD*, pages 481–492, June 1996.

[11] W. Inmon and C. Kelley. *Rdb/VMS: Developing the Data Warehouse.* QED Publishing Group, Boston, Massachusetts, 1993.

[12] B. Lindsay, L. Haas, C. Mohan, H. Pirahesh, and P. Wilms. A snapshot differential refresh algorithm. In *SIGMOD*, May 1986.

[13] D. Lomet and J. Widom, editors. *Special Issue on Materialized Views and Data Warehousing,* IEEE Data Engineering Bulletin 18(2), June 1995.

[14] A. Segev and J. Park. Updating distributed materialized views. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):173–184, June 1989.

[15] Sybase, Inc. *Command Reference Manual*, release 4.9 edition, 1992.

[16] J. Wiener, H. Gupta, W. Labio, Y. Zhuge, H. Garcia-Molina, and J. Widom. A system prototype for warehouse view maintenance. In *The Workshop on Materialized Views*, pages 26–33, June 1996.

[17] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *SIGMOD*, pages 316–327, May 1995.

[18] Y. Zhuge, H. Garcia-Molina, and J. Wiener. The Strobe algorithms for multi-source warehouse consistency. Technical report, Stanford University, Sept. 1996. Available via anonymous ftp from host `db.stanford.edu` as `pub/zhuge/1996/strobe-full.ps`.