, , 1–36 ()

Consistency Algorithms for Multi-Source Warehouse View Maintenance* **

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://WWW-DB.STANFORD.EDU/WAREHOUSING

Editor:

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.

Keywords: data warehouse, data consistency, view maintenance

1. Introduction

A *data warehouse* is a repository of integrated information from distributed, autonomous, and possibly heterogeneous, sources. Figure 1.1 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

* This work was partially supported by Rome Laboratories under Air Force Contract F30602-94-C-0237; by an equipment grant from Digital Equipment Corporation; and by the Advanced Research and Development Committee of the Community Management Staff, as a project in the Massive Digital Data Systems Program.
** This paper is an extended version of a paper published in International Conference on Parallel and Distributed Information Systems, December 1996. The main additions are in Section 6.2 and Section 8. Also, we included the full proof of correctness of Strobe and C-Strobe algorithm in Appendix A.1 and A.2.

in the source data, and notifying the warehouse. If a source provides relational-style triggers, the monitor may simply pass on information. On the other hand, a monitor for a legacy source may need to compute the difference between successive snapshots of the source data. 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. The data is then readily available to user applications for querying and analysis.

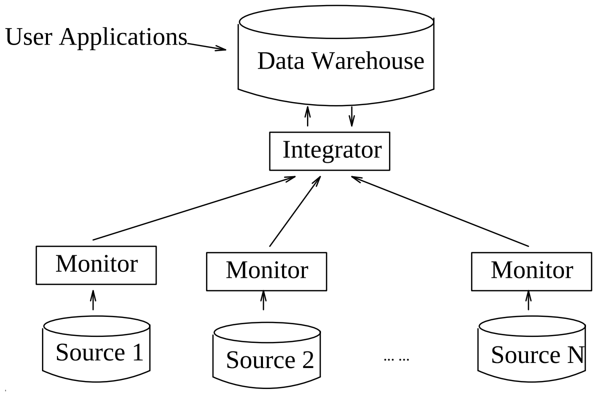

Figure 1.1. Data warehouse architecture

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 [12].

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 participating relation. In this example, each update is a separate transaction at one of the sources.

Example 0: View updating 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 [4], the following events may occur at the warehouse.

1. The warehouse 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 warehouse first sends query $Q_1^1 = r_1 \bowtie [2,3]$ to source x.

2. The warehouse 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 warehouse 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 warehouse 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 is 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 [7], 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 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 [28] 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 [28] 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 [12]. 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 [27], demonstrating that the algorithms are practical and efficient.
4. We map out the space of warehouse maintenance algorithms (Figure 9.3). 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 6. We also demonstrate the levels of consistency that each algorithm achieves for the different transaction scenarios. In Section 7, we adapt the algorithms so that the warehouse can reflect every update individually, and show that the algorithms will terminate. Section 8 gives a brief overview of the algorithm implementation status. We conclude in Section 9 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 [18]. 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. A good overview of the mechanisms for managing replicated data may be found in [3]. However, because of the autonomy of the sources, traditional concurrency mechanisms are often not applicable [5]. A variety of concurrency control schemes have been suggested over the years for such environments (for example, [1, 26, 10]). They either provide weaker notions of consistency, 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 [4, 14, 6, 16, 21, 24, 8] and a good overview of materialized views and their maintenance can be found in [13]. 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 [4], or using the new base relations, as in [8]. 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. Furthermore, all previous solutions require that the base relations be stable (e.g., locked) while computing the changes to the view. We allow modifications to the base relations to execute concurrently, and then compensate the proposed view changes for those modifications.

Previous distributed algorithms for view maintenance, such as those in [22, 23, 20], 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 [17] 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. As a result, they emphasize finding out which copies of base relations need to be kept consistent and how, while we propose algorithms to maintain joined views directly, without storing any auxiliary data. Furthermore, they require a notion of global time. We compare our definition of consistency with theirs in Section 4. Another recent paper by Baralis, et al. [2] 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 [28] 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 sep-

arately. 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 contents 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 [25]. 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 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 6.2 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. Events are atomic, and are ordered by the sequence of the corresponding actions.

We assume that any two messages sent from one source to the warehouse are delivered in the same order as they were sent. However, we place no restrictions on the order in which messages sent from different sources to the warehouse are delivered. That is, the sources are not coordinated. For example, even if a message from source x is sent prior to a message from source y, the warehouse may receive y's message first.

The second assumption (point-to-point message ordering) may not always hold, e.g., if two messages are sent by different routes; however, the crucial aspect is not that the messages arrive in order, but that they are processed in order. In our implementation, we add a sequence number to all messages sent by each source; the warehouse can then detect a missing message and wait for it before processing any later messages from that source.

The first assumption (atomic events at sources) is reasonable for sources that run transactions (either local or global), but may not be reasonable for legacy sources that have no transaction facilities. If such sources report their events in arbitrary order, then in some cases there is no way we can guarantee correctness. For example, if the source inserts a tuple and then deletes it, but report these events in reverse order, then there is no way for the warehouse to guess what actually happened. In other cases the warehouse may guarantee one of our weaker notions of consistency, for example, convergence. For ease of exposition we assume atomic events for all sources, and return to this issue in Section 9.

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 6.2. 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 $result(R)$ 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 definition are a generalization of the ones in [28] 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 $result(R)[x] = ss_j[x]$. 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., $ss_j = result(R')$ 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 [17] is similar to our strong consistency, except that they also require a 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 [11] at the sources.

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. Strobe Algorithms

In the following three sections, 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 7 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.

Definition. The materialized view MV of a view V is the current state of V at the warehouse, $V(ws)$.

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 1: 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.

$source\_evaluate(Q)$: returns answers to Q

Begin
    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

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.

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 [13].

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 occurs while Q is processed. After Q's answer is fully compensated, an insert action for MV is generated and placed on the action list AL.

Strobe algorithm

At each source:
 $\triangleright$ After executing update U_i, send U_i to the warehouse.
 $\triangleright$ 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:
 $\triangleright$ Initially, AL is set to empty $\langle \quad \rangle$.
 $\triangleright$ Upon receipt of update U_i:
  $\circ$ 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.
  $\circ$ 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.
 $\triangleright$ When $UQS = \emptyset$, apply AL to MV as a single transaction, without adding duplicate tuples to MV
 Reset $AL = \langle \quad \rangle$.
End Strobe algorithm.

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.

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 2: 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 warehouse 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 warehouse first sends query $Q_1^1 = r_1 \bowtie [2,3]$ to source x.

2. The warehouse 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 warehouse 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 warehouse receives $A_1^2 = [1,2,3,4]$ from source z. Since $pending(Q)$ is not empty, the warehouse 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 warehouse 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. We prove the correctness of Strobe algorithm in Appendix A.1.

6. Transaction-Strobe and Global-Strobe

In this section we present two variations of the Strobe algorithm: T-Strobe that handles source-local transactions, and G-Strobe that handles global transactions.

6.1. Transaction-Strobe algorithm

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).

Transaction-Strobe algorithm (T-Strobe)

1. At the warehouse:
2.   ▷ Initially, $AL = \langle \quad \rangle$.
3.   ▷ Upon receipt of $UL(T_i)$ for a transaction T_i:
4.   ∘ 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 $UL(T_i)$.
5.   ∘ For every deletion $U \in UL(T_i)$:
6.   — $\forall Q_j \in UQS$, add U to $pending(Q_j)$.
7.   — Add $key\_delete(MV, U)$ to AL.
8.   ∘ Let $Q_i = \bigcup_{U_j \in IL(T)} V\langle U_j \rangle$, and set $pending(Q_i) = \emptyset$;
9.   ∘ Let $A_i = source\_evaluate(Q_i)$;
10.   ∘ $\forall U \in pending(Q_i)$, apply $key\_delete(A_i, U)$;
11.   ∘ Add $insert(MV, A_i)$ to AL.
12.   ▷ When $UQS = \emptyset$, apply AL to MV, without adding duplicate tuples to MV. Reset $AL = \langle \quad \rangle$.
13. End algorithm

The source actions in T-Strobe are the same as in Strobe; we therefore present only the warehouse actions. First, the warehouse 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 warehouse adds all remaining deletions to the action list AL. Finally, the warehouse generates one query for all of the insertions. As before, deletions which arrive at the warehouse 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.

Example 3: 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, let the relation r_1 contain a single tuple $[1, 2]$. 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 process 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.

6.2. Global-Strobe

While the T-Strobe algorithm is strongly consistent for source-local transactions, we show in the next example that it is only weakly consistent if global transactions are present. We then devise a new algorithm, Global-Strobe, to guarantee strong consistency for global transactions. Since the capabilities of the sources in a warehousing system may vary, we discuss several possible ways to create the Global-Strobe algorithm, based on the cooperativeness of the sources.

Example 4: T-Strobe with global transactions

Let the warehouse view V be defined as $V = r_1 \bowtie r_2$, where r_1, r_2 reside at sources x and y, respectively. Assume A is the key for r_1 and C is the key for r_2. Initially, the relations are:

r_1 :

A	B
–	–

r_2 :

B	C
3	4
3	5

The materialized view $MV = \emptyset$. $AL = \langle\ \rangle$. We consider two source transactions: $T_1 = \langle U_1 = insert(r_1, [1,3])\rangle$ and $T_2 = \langle U_2 = delete(r_2, [3,4]), U_3 = insert(r_1, [2,3])\rangle$ and apply the T-Strobe algorithm.

1. The warehouse receives $U_1 = insert(r_1, [1,3])$ from source x. It generates query $Q_1 = [1,3] \bowtie r_2$ and sends Q_1 to source y for evaluation.

2. The warehouse receives $U_2 = delete(r_2, [3,4])$ from source y.[2] Since U_2 belongs to a global transaction, it arrives at the warehouse with a transaction id attached. The warehouse temporarily stores U_2 in a holding queue, and does not process it until the remaining T_2 updates arrive.

3. The warehouse receives $A_1 = ([1,3,5])$ from source y. This answer was evaluated after U_2 occurred at source y. $Insert(MV, A_1)$ is added to AL. Because $UQS = \emptyset$, the warehouse applies AL to MV and $MV = ([1,3,5])$, which is a globally inconsistent state. (It is a weakly consistent state: the warehouse sees source x after T_1 but before T_2 and source y after both T_1 and T_2.)

4. The warehouse receives U_3 from source x, with an attached transaction id for T_2. Now that the T_2 updates have been fully received, T-Strobe adds $key\_delete(WC, U_2)$ to AL and sends query $Q_2 = [2,3] \bowtie r_2$ to source y.

5. The warehouse receives $A_2 = ([2,3,5])$ from source y and adds $insert(MV, A_2)$ to AL. Since $UQS = \emptyset$, $AL = \langle key\_delete(MV, U_2), insert(MV, A_2)\rangle$ is applied to MV. The final MV is ([1,3,5], [2,3,5]), which is correct. □

In step 3, above, the view is updated to a globally inconsistent state: $MV = ([1,3,5])$. The inconsistent state occurs because the evaluation of query Q_1 interferes with global transaction T_2. If the two actions in T_2 were treated as separate local transactions, then the state $MV = ([1,3,5])$ would be consistent with the source states after U_1 and U_2 (but before U_3). Therefore, T-Strobe is weakly but not strongly consistent in the presence of global transactions.

Example 5 shows that to achieve strong consistency, the warehouse needs to ensure that no global source transactions like T_2 are "pending" before it modifies the view. We now modify T-Strobe for the global transaction scenario, and create a new algorithm, Global-Strobe (G-strobe). Let TT be the set of transaction identifiers that the warehouse has received since it last updated MV. 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:

1. $UQS = \emptyset$;
2. For each transaction T_i in TT that depends on (in the concurrency control sense) 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.

When we apply the G-Strobe algorithm to the scenario in example 5, we see that now the warehouse will not update MV after processing T_1. Although at this point there are no unanswered queries, one update belonging to transaction T_2 has been received, which may have affected the evaluation of Q_1. Therefore, the warehouse delays updating MV until after receiving and processing all of T_2.

Enforcing condition 3, above, is easy if the sources cooperate, even when there is no global concurrency control. If all of the updates of a global transaction are sent in a single message by the committing site, then the warehouse will always have the entire transactions. If the updates are sent in separate messages, then transactions identifiers are needed in each message, plus a count of how many updates are involved in the transaction. Together, the count and identifiers make it possible for the warehouse to collect all of the updates before processing them.

Enforcing condition 2, above, may be more problematic in practice. Sources typically do not report the transactions on which a committing transaction depends. This means that condition 2 must be enforced indirectly. To illustrate, suppose transaction T_1 performs update U_{1x} at site x and U_{1y} at site y. Similarly, T_2 performs U_{2y} at y and U_{2z} at z. Assume that T_2 commits second and depends on T_1. If the updates are sent by the sources individually and in commit order, then the warehouse must receive U_{1y} before U_{2y}. Therefore, it is not possible to receive a transaction (e.g., T_2) without first receiving at least one of the updates of every transaction on which it depends (e.g., T_1), in the sense of transaction dependencies [3]. That is, condition 2 is automatically enforced.

If, on the other hand, site z reports all of the updates of T_2, then these updates could arrive before the warehouse receives any of T_1's updates. To enforce condition 2 in this scenario, we need to add sequence numbers to individual updates, and wait for all prior updates from a source. In our example, when T_2 is received, U_{2y} would contain a sequence number, say, 33. Then the warehouse would delay processing T_2 until all updates from source y with sequence numbers less than 33 (such as U_{1y}) arrive. This strategy is very conservative but does ensure condition 2.

In summary, the mechanism for sending transactions to the warehouse will determine if G-Strobe can reasonably guarantee strongly consistent views at the warehouse. If G-Strobe is not feasible, then we can revert to Strobe or T-Strobe and provide a weaker level of consistency. (Strobe only guarantees convergence for global transactions, as it does for source-local transactions. As we stated above, T-Strobe is weakly consistent for global transactions.)

7. 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.)

Definition. Define $\Pi_{proj}(U_i)$ to be the tuple that agrees in all values with tuple of update U_i, but only has those attributes of U_i that appears in the list *proj*.

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 attributes (not just on key attributes). Finally, when we add tuples to $Delta$, we

Complete Strobe algorithm(C-Strobe)

1. At the warehouse:
2. ▷ Initially, $Delta = \emptyset$.
3. ▷ As updates arrive, they are placed in a holding queue.
4. ▷ We process each update U_i in order of arrival:
5. ∘ If U_i is a deletion
6. Apply $key\_delete(MV, U_i)$.
7. ∘ If U_i is an insertion
8. — Let $Q_{i,i,0} = V\langle U_i \rangle$;
9. — Let $A_{i,i,0} = source\_evaluate(Q_{i,i,0})$;
10. — Repeat for each $A_{i,j,k}$ until $UQS = \emptyset$:
11. Add $A_{i,j,k}$ to $Delta$ (without adding duplicate tuples).
12. For all deletions U_p received between U_j and $A_{i,j,k}$:
13. Let $Q_{i,p,new_p} = Q_{i,j,k}\langle U_p \rangle$;
14. Let $A_{i,p,new_p} = source\_evaluate(Q_{i,p,new_p})$;
15. When answer arrives, processing starts from line 10.
16. — 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 $\Pi_{proj}(U_j) = \Pi_{proj}(U_k)$, then apply $key\_delete^*(Delta, U_k)$.
17. — Let $MV = MV + Delta$ and $Delta = \emptyset$.
18. End algorithm.

assume that tuples with the same key values but different non-key values will 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.) We present the warehouse C-Strobe algorithm, the source behavior remains the same as for the Strobe algorithm.

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 [29].

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: C-Strobe applied to example of introduction

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 warehouse receives $U_1 = insert(r_2, [2,3])$ from source y. It generates query $Q_{1,1,0} = r_1 \bowtie [2,3] \bowtie r_3$. To evaluate $Q_{1,1,0}$, the warehouse first sends query $Q^1_{1,1,0} = r_1 \bowtie [2,3]$ to source x.
2. The warehouse 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 warehouse receives $U_2 = delete(r_1, [1,2])$ from source x. It saves this update in a queue.
4. The warehouse 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 warehouse 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 warehouse 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 warehouse updates the view to $MV = MV + Delta = ([1,2,3,4])$.
7. Next the warehouse 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 6. We now consider a different set of events at the warehouse.

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

The algorithm C-Strobe is complete. We give the proof of correctness in Appendix A.2.

As mentioned earlier, C-Strobe can be extended to update MV periodically, after processing $k > 1$ 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.

8. Implementation of the Strobe family of algorithms

The Strobe family of algorithms, include Strobe, T-Strobe and C-Strobe are implemented in the WHIPS prototype (WareHousing Information Project at Stanford) [27]. Figure 8.2 shows the architecture of the WHIPS system. Each box in the figure is a module that performs a specific function. Each module is implemented as a CORBA object. They communicate with each other using ILU, a COBRA compliant object library developed by Xerox PARC[9]. Different modules can reside on different machines and can be implemented in different language. Similar to a real-world data warehousing system, data sources are separated from the integration modules, and the warehouse is a separate database which may or may not be closely coupled with the integration components.

In WHIPS, views are defined through a *view specifier* using an SQL-like view definition language. When defining a view, the desired consistency level of this view is specified. For example, a clause "**with** strong **consistency**" specifies that the current view will be maintained in a strongly consistent fashion. When the *integrator* receives a view definition, it spawns a *view manager* that executes the required consistency algorithm. For example, the Strobe algorithm is implemented in a Strobe view manager that maintains a view requiring strong consistency. When the system is running, one view manager maintains one view. After a view is initialized and stored at the warehouse, *source monitors* detect changes on the source data and notify the integrator [19]. The integrator then forwards source updates to the relevant view managers.

All view managers work concurrently to maintain their own view. A view manager manages source updates and maintains data structures for view maintenance. For example, a Strobe view manager maintains UQS and $pending(Q)$ (the pending delete queue) required by the algorithm. A view manager sends proper queries to

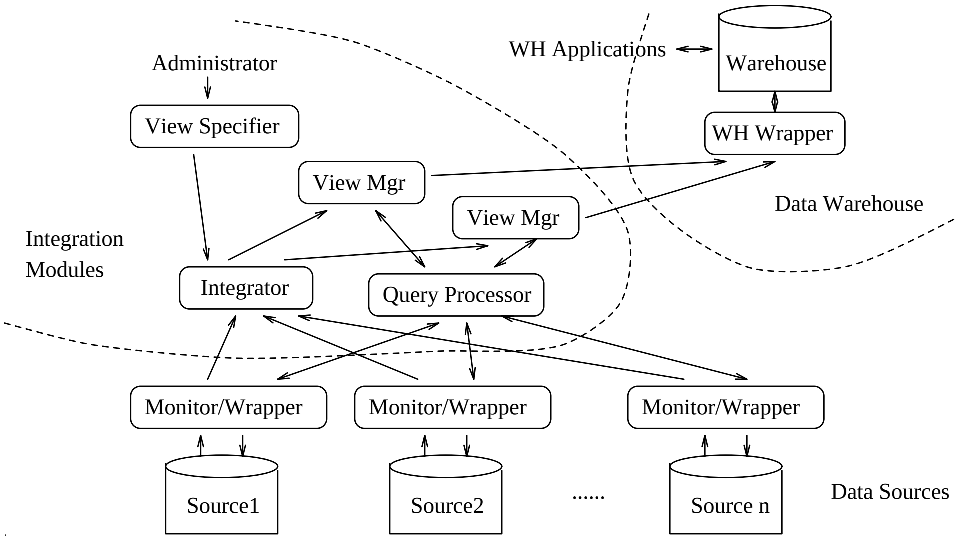

Figure 8.2. The WHIPS system architecture

the *query processor* and receives answers to those queries. Then it may perform some post processing on the answers and decide when to update the materialized view. It sends an *action list* to the warehouse wrapper to initialize or modify a materialized view.

The query processor basically implements the procedure *source_evaluate*() required by the Strobe algorithms. It receives global queries from the view managers and poses the appropriate single-source queries to the source wrappers to answer them. It then passes the composite global query answers back to the view managers. In our current implementation, the query processor uses standard techniques such as sideways information passing and filtering of selection conditions to prune the queries it poses to the wrappers. Because of the modularity of the WHIPS system, other query processing techniques can be added to the query processor without affecting other parts of the system. There could also be more than one query processor if necessary. The *warehouse wrapper* receives action lists from the view managers and executes the desired actions on views. All modifications received by the warehouse wrapper in a single message are applied to the warehouse in one transaction, as needed by the view consistency algorithms.

The implementation of the Strobe family algorithms is straightforward given the pseudo-code we have presented. We use *view tree*, an internal representation which is basically a parse tree structure for a relational view, for representing both view definitions and queries on a view. A view definition is translated into a view tree at initialization time. A query related to view V is the view tree of V with some conditions changed or some base relation substituted by know relations or tuples.

The answer to a query is simply a relation. Notice that although our internal view representation is relational, we do not require that the data warehouse or any of the data sources be relational. The wrappers can translate the relational queries and data used by the integrator into the native language and model used by the warehouse or sources.

The algorithms are relatively inexpensive to implement. In our implementation, the Strobe view manager is about 250 lines of C++ code, the T-Strobe is about 350 lines and C-Strobe is about 400 lines. As a comparison, we implemented a view manager that runs the conventional (centralized) view maintenance algorithm, and it is 150 lines. Remember that the centralized algorithm is inconsistent in this distributed warehousing environment. So the ability to guarantee correctness (Strobe), the ability to batch transactions (T-Strobe), and the ability to update the view consistently, whenever desired and without quiescing updates (C-Strobe) cost 100 to 250 extra lines of code in this implementation. The entire WHIPS system is about 12,000 lines of C++ code.

9. 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 9.3 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 [28].

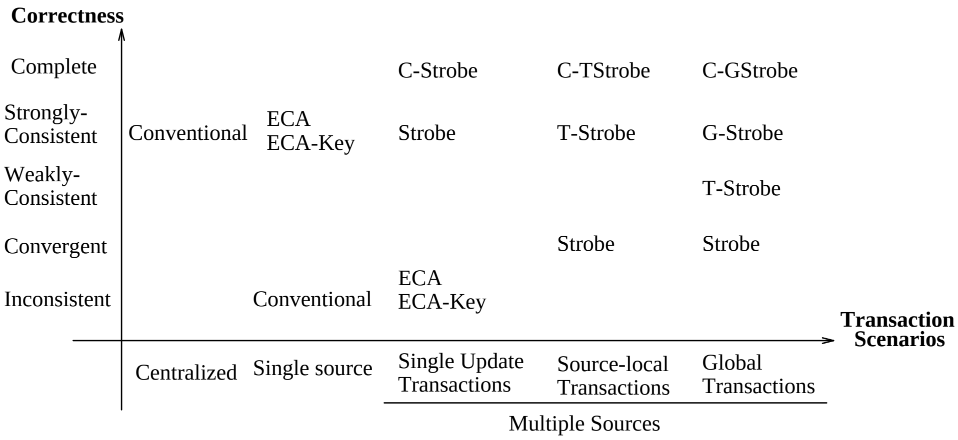

Figure 9.3. Consistency Spectrum

In Figure 9.3, 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 three 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 the 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.

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. We can use key information to avoid sending some queries to sources [15]. For example, suppose the view definition is $V = r_1 \bowtie r_2$, r_1 has attributes A, B, and r_2 has attributes B, C. Further suppose that the current MV contains tuple $[1, 2, 3]$ and we know that B is a key for r_2. If the warehouse receives an update $U_1 = insert(r_1, [4, 2])$, there is no need to send the query $[4, 2] \bowtie r_2$ to the source containing r_2. Because B is the key for r_2, and because the view contains $[1, 2, 3]$, we know that r_2 must contain $[2, 3]$. Therefore, we need to add (exactly) the tuple $[4, 2, 3]$ to the view.

4. 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 [4]. 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.

Recall that in Section 3 we made a critical "event ordering" assumption. This assumption states that sources report events in the correct order. As we stated, if legacy sources cannot guarantee this, then it is impossible to guarantee any type of correctness. However, if the sources only report updates in the correct order (but may report query answers in any order with respect to the updates), then we can still achieve eventual consistency (i.e., guarantee convergence). There are two cases to consider: (1) an answer A_i arrives at the warehouse *before* an update U_j that is already reflected in A_i, and (2) A_i is computed before U_j occurs at a source, but A_i arrives at the warehouse *after* U_j does. The first case may cause the warehouse to compute an incorrect materialized view, but eventually, when U_j arrives, the view will be corrected. Therefore, the warehouse can still guarantee convergence. The Strobe algorithms already handles the second case by using $pending(Q_i)$, and also guarantee convergence at the end of processing both Q_i and U_j. If we make the reasonable assumption that concurrent answers and updates will arrive soon after each other, if not in the correct order, then the warehouse will not diverge far from a consistent state, will always return to a consistent state, and will do so fairly quickly.

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 [30].

Acknowledgments

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

Appendix

A.1. Proof of correctness for Strobe Algorithm

The Strobe algorithm provides strong consistency for all single-update transaction environments. In this section we outline a proof for the Strobe algorithm. 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. We note that the ECA-key algorithm in [28] does not always process deletions correctly even in a single source environment. The problem occurs when the deleted tuple is the same as an inserted tuple participating in an ongoing query. The Strobe algorithm corrects this error and processes all updates correctly.

THEOREM 1 *The Strobe algorithm is strongly consistent.*

Proof: Assume that the warehouse receives and processes updates in the order $U_1, U_2, \ldots$. In this scenario, each update represents a source transaction, so let R be the serial schedule where transactions are executed in this order. Notice that R must be equivalent to the actual schedule executed at the sources, S. If transactions at a particular source x are executed in a given order, this must be the order in R because the Strobe algorithm processes updates in the order received. Transactions from different sources may appear in a different order in R, but this does not matter since they do not conflict (the transaction access data at different sites).

To illustrate, assume two updates U_{1x}, U_{2x} occur at source x in that order and update U_{1y} occurs at source y. Assume the source schedule is $S = U_{1x}, U_{2x}, U_{1y}$, but updates arrive at the warehouse in the order $R = U_{1x}, U_{1y}, U_{2x}$. Since U_{1y} and U_{2x} do not conflict they can be swapped, and the two schedules are equivalent.

Now that we have defined the schedule R required by the strong consistency definition, let us define the mapping m. Consider a warehouse state ws_x where $UQS = \emptyset$. Say that at this point the Strobe algorithm has processed updates U_1 through U_k inclusive. We map ws_x into ss_k, where $ss_k = result(U_1, \ldots, U_k)$.

Now we must show that $V[ss_k] = V[ws_x]$. We do this by contradiction, i.e., assume that $V[ss_k] \neq V[ws_x]$. Then there must be a tuple t that is either missing from $V[ws_x]$ or is an extra tuple in $V[ws_x]$. (In what follows, we do not have to worry about t appearing more than once either at the source or the warehouse due to our key condition.) We also know that $V[ss_0] = V[ws_0]$, where ss_0 is the initial source state and ws_0 is the initial warehouse state. There are two main cases to consider.

Case I: $t \in V[ss_k]$ and $t \notin V[ws_x]$.

Subcase I(a): There is (at least one) insert at the source that generates t. Let U_i be the last of these inserts that are processed at the warehouse. Insert U_i adds a tuple that contains one of the keys involved in t to some relation at a source. (Recall that t has a key value for each of the relations involved in the view definition.) There can be no deletes involving a t key after U_i and before U_k. If there were, they would remove t from $V[ss_k]$, a contradiction.

Under Strobe, U_i gets propagated to the warehouse, a query Q is generated, evaluated using source relations, and an answer returned to the warehouse. At the warehouse, both the processing of U_i and the receipt of the Q answer must occur before or at state ws_x. (U_i cannot be received after ws_x since we are assuming that U_i is a processed update. If the Q answer arrived after ws_x, then UQS would not have been empty at this state.) When Q is processed at the source, it sees all the key values involved in t, so the answer to Q contains t. Thus, t is inserted into the materialized view. Because there are no subsequent deletes affecting t at the source between the processing of U_i and U_k, t remains until ws_x, a contradiction.

Subcase I(b): There are no inserts at the source that generate t. This means that t must have been in $V[ss_0]$, and thus in $V[ws_0]$. Similarly to Case I(a), there are no deletes affecting t at the source before ss_k (otherwise t could not be in $V[ss_k]$) and hence no such deletes at the warehouse. Thus, t remains at the warehouse, a contradiction.

Case II: $t \notin V[ss_k]$ and $t \in V[ws_x]$.
Subcase II(a): There is at least one delete at some source that involves a t key value. The proof is analogous to Case I(a). We consider the last such source delete U_d processed at the warehouse. Clearly, the warehouse deletes t from the materialized view when processing U_d. Since $t \in V[ws_x]$, it must have been reinserted. Notice that there are no inserts that could generate t at the source after U_d, so there cannot be any corresponding inserts that warehouse. However, it could be the case that an insert U_j occurring at the source *before* U_d, could generate an answer that is processed at the warehouse *after* U_d. In Strobe, all such queries are processed by *key-delete*(Q_j, U_d). This deletes tuple t from the answer if it was there. Thus, there is no way the answer could contain t when it is added to MV, this is a contradiction.
Subcase II(b): There are no source deletes that involve a key value of t. Therefore t must not be in the initial $V[ss_0]$, and since the initial materialized view is correct, t is not in $V[ws_0]$. Since somehow t appears in $V[ws_x]$, it must have been inserted by some answer received at the warehouse. Say ss_l is the source state right after the answer is generated. At that point $V[ss_l]$ must contain t. Since there are no deletes affecting t at the source, then t remains, a contradiction.

We have now shown that $V[ss_k] = V[ws_x]$. Since the materialized view only changes at the warehouse when $UQS = \emptyset$, this completes the first part of the proof that Strobe is strongly consistent. For the second part we must show that if we have two warehouse states, ws_g and ws_x, where $g < f$, then the source states they map to occur in this same order. Let U_j be the last update processed at the warehouse at ws_g and U_k be the last update at ws_x. Clearly, $j < k$, so source state ss_j will occur before state ss_k as R is executed. This completes the proof.

■

A.2. Proof of correctness for Complete Strobe Algorithm

THEOREM 2 *The Complete Strobe (C-Strobe) algorithm is complete.*

Proof: C-Strobe processes updates in the order of their arrival. Assume that the list of updates at the warehouse in their processing order is $U_1, U_2, \ldots$. Let R be the serial schedule where transactions are executed in this order. As we argued in Section A.1, R is a serial source schedule that is equivalent to the actual execution, and R will serve as the serial schedule required by the proof of complete consistency. Let the warehouse state after processing U_x be ws_x and let the source state $ss_x = result(U_1, \ldots, U_x)$, ss_x is a consistent source state.

Let $MV[ws_x]$ be the materialized view at state ws_x. $MV[ws_x]$ is the same as $V[ws_x]$ in the consistency definition, we use $MV[ws_x]$ here because MV is used in the algorithm. We know that $MV[ws_0] = V[ss_0]$. Again, C-Strobe receives updates from sources, stores them in the order of their arrival and processes one update at a time (line 4 to line 17). We call the update U_i in line 4 the *current update*, and all the discussions below are within the context of processing U_i. Assume we have $MV[ws_{i-1}] = V[ss_{i-1}]$. That is, MV at state ws_{i-1} correctly reflects the source state ss_{i-1}. We prove by induction that $MV[ws_i] = V[ss_i]$, $MV[ws_i]$ is the materialized view after processing U_i using C-Strobe.

Case I: If U_i is a deletion, then the deletion is applied to MV directly. By inductive hypothesis we know that before applying U_i, MV is the same as $V[ss_{i-1}]$. After applying the deletion, $MV[ws_i]$ is $MV[ws_{i-1}]$ with all tuples whose keys agree with U_i deleted. At the same time, $V[ss_i]$ is $V[ss_{i-1}]$ with all tuples whose keys agree with U_i deleted. By the key assumption, one tuple in the view has only one derivation from the base relations. So $MV[ws_i] = V[ss_i]$.

Case II: When U_i is an insertion, we know from basic incremental view update theory that $V[ss_i] = V[ss_{i-1}] \cup V\langle U_i\rangle[ss_{i-1}]$. That is, the correct view increment corresponding to U_i is $V\langle U_i\rangle$ evaluated at source state ss_{i-1}. From the algorithm we have $MV[ws_i] = MV[ws_{i-1}] \cup Delta$. That is, the answer to $Q_{i,i,0}$ and the results of all following compensating queries and actions are stored in $Delta$ when MV is updated. The expression we want to prove is:

$$V[ss_{i-1}] \cup V\langle U_i\rangle[ss_{i-1}] = MV[ws_{i-1}] \cup Delta \qquad \text{(A.2.1)}$$

Following we first prove that $V\langle U_i\rangle[ss_{i-1}] \subset Delta$, then we prove that $Delta \subset MV[ws_{i-1}] \cup V\langle U_i\rangle[ss_{i-1}]$. The two cases will lead us to prove expression A.2.1.

Case II(a): All tuples in $V\langle U_i\rangle[ss_{i-1}]$ are in $Delta$.

A query Q sent by C-Strobe could be either the initial query corresponding to U_i (line 8), or one of compensating queries that is caused by a deletion U_d (line 13). In either case, one query Q is directly triggered by one update U. (U is U_i in the first case and U_d in the second case.) Q is intended to be evaluated at the moment when it is sent out, i.e., at the source state with no further updates occurring after U. If there are actually updates after U, then first (line 12-15) deletions among those updates are compensated by sending compensating queries, second (line 16) insertions are compensated by deleting those extra tuples introduced.

Assume t is a tuple in $V\langle U_i\rangle[ss_{i-1}]$. First, we prove that t should be in $Delta$ after all deletions are compensated (when the algorithm reaches line 16). Let $BT(t)$ be the set of *base relation tuples* that derive view tuple t. For example, say we have base relations $R(A,B)$, $S(B,C)$, with A and B as keys of R and S, respectively. If R contains a single tuple $[1,2]$ and S contains a single tuple $[2,3]$, and we have $V = \Pi_{A,B}(R \bowtie S)$, then $[1,2]$ is a view tuple and $BT([1,2]) = \{[1,2],[2,3]\}$. As we can see, different sets of base tuples may derive the same view tuple. However, because of our key constraints on view definitions, a view tuple has only one derivation at a given time. We consider the following three scenarios:

1. There are no further updates after the warehouse receives U_i and before it receives $A_{i,i,0}$. As a result of the event order assumption 3.3, no update *occurred* after U_i and before the evaluation of the answer. Since $A_{i,i,0}$ is the intended results for $Q_{i,i,0} = V\langle U_i\rangle$,

That means $Q_{i,i,0}$ was evaluated in state ss_{i-1}. So $A_{i,i,0} = V\langle U_i \rangle[ss_{i-1}]$ and t is in $A_{i,i,0}$. Since $A_{i,i,0}$ is added to *Delta* in line 11, t should be in *Delta* when the algorithm reaches line 16.

2. There are no deletions of those tuple in $BT(t)$ between the receipt of U_i and the receipt of $A_{i,i,0}$. In this scenario, t should be in $A_{i,i,0}$. Insertions and deletions of tuples not in $BT(t)$ do not affect the existence of t in $A_{i,i,0}$. As argued above, t is added to *Delta*.

3. At least one tuple in $BT(t)$ is deleted from a base relation after U_i and before the receipt of $A_{i,i,0}$. In this scenario, tuple t may be missing from $A_{i,i,0}$. Let $Q_{i,i,0}$ be query X_1. Let the first of those deletions from $BT(t)$ be U_k. According to C-Strobe, a query $Q_{i,k,x}$ is sent corresponding to U_k. Let query $Q_{i,k,x}$ be X_2. In query X_2, U_k is provided (as well as U_i). If there are no more deletions of tuple in $BT(t)$ after X_2 is sent, then because U_i and U_k are present in the query and other tuples in $BT(t)$ are still in the base relations, tuple t will be contained in the answer of X_2. Therefore, t is added to *Delta*.

   When the answer of query X_2 is received, either we have t in the answer and add it to *Delta*, or another query X_3 which contains U_i, U_k and another tuple in $BT(t)$ will be generated. We could make similar arguments in the situation where there are further deletions of tuples in $BT(t)$ before finish the entire processing of U_i. This process repeats until either

   (A) There are no deletions of tuples in $BT(t)$ between sending a query X_l and receiving its answer. In this case, tuple t is contained in the answer of X_l and added to *Delta*.

   (B) Query X_l contains all tuples in $BT(t)$. We know this will happen because each time C-Strobe generates a compensating query (line 13) it adds one known tuple into the original query. So X_{l+1} always contains one more tuple than X_l and that tuple is in $BT(t)$. We know that X_1 contains $n-1$ base relations (n is the number of base relations in the view definition). So query X_l will contain $n-l$ base relations, and $X_n (l = n)$ has only known tuples from $BT(t)$. Thus, X_n can be evaluated locally at the warehouse. Further deletions of tuples in $BT(t)$ from base relations do affect the answer of X_n. Since all tuples in $BT(t)$ are carried by X_n, tuple t is guaranteed to be in the answer of X_n, and thus, in *Delta*.

   Notice that to keep all potential tuples in *Delta*, we need to let *Delta* take tuples with the same key values but different non-key values. To see this, suppose that at a given time a tuple t is missing from *Delta* because a deletion U_d caused the answer that was to contain t to be incomplete. A compensation query Q_d was sent as a result of U_d. However, before the answer of Q_d arrives, an insertion into a base relation may derive a tuple t' which has the same key value with t. Tuple t' may be brought to the warehouse by another answer relation and be added into *Delta* before t. If t' does not agree with t on non-key values, then it may be removed from

Delta when compensating inserts later. However, t should remain in *Delta*. Therefore, at this point we should keep both t and t' in *Delta* and should not reject t because there is a tuple in *Delta* with the same key value.

Now we have shown that after processing all the deletions, all tuples in the view increment $V\langle U_i\rangle[ss_{i-1}]$ are in *Delta*. Next, we argue that no "useful" tuple t can be deleted accidently when processing further insertions (in line 16). Consider an insertion U_k into relation R that is received before we finish processing U_i. U_k is compensated for in line 16. Let t be a tuple in $V\langle U_i\rangle[ss_{i-1}]$,

1. If $U_k \notin BT(t)$, then $key\_delete^*(Delta, U_k)$ will not delete tuple t from *Delta*.
2. If $U_k \in BT(t)$ and U_k is an insert into relation R_1. Since t was in $V\langle U_i\rangle[ss_{i-1}]$, in state ss_{i-1} there must be an R_1 tuple t_1 that matches U_k in all the attributes in the *proj* list, that is, $\Pi_{proj}(t_1) = \Pi_{proj}(U_k)$. Tuple t_1 is not in R_1 when U_k occurs at the source (otherwise the insertion U_k will fail since U_k and t have the same projected attributes, including key attributes.) Therefore, t_1 must have been deleted before U_k. In this case, $key\_delete^*(Delta, U_k)$ is not applied and t remains in *Delta*.

Case II(b): All tuples in *Delta* must be either in $V\langle U_i\rangle[ss_{i-1}]$, or in $MV[ws_{i-1}]$. Tuples in *Delta* all came from answers of the queries (line 11). We consider the following three scenarios:

1. When there are no updates after U_i and before we finish processing U_i. In this scenario, $Delta = A_{i,i,0} = V\langle U_i\rangle[ss_{i-1}]$.
2. There are only deletions after U_i.

   We know from the above discussions that a query Q is always directly triggered by an update U (U_i or U_d). Let the *intended answer* of Q be the answer relation if Q is evaluated at the source state when there is no update after U. Let the *actual answer* of Q be the answer relation the warehouse receives when it executes C-Strobe. When there are only deletions after U_i, it is clear that the actual answer of any query can only have *less* tuples than the intended answer. In the following we show that any *intended answer* of queries sent by C-Strobe is a subset of $V\langle U_i\rangle[ss_{i-1}]$. Then the sum of all actual answers, which is a subset of all intended answers, is also a subset of $V\langle U_i\rangle[ss_{i-1}]$. *Delta* is the sum of all the actual answers.

   From item 1 above we know that for $Q_{i,i,0}$, the intended answer is the same as $V\langle U_i\rangle[ss_{i-1}]$.

   Any compensating query (sent at line 13) is obtained from substituting one base relation by a deleted tuple from a previous query. So a compensating query must have the following format:

   $$\begin{aligned} Q &= V\langle U_i, U_{d_1}, U_{d_2} \dots U_{d_k}\rangle \\ &= \Pi_{proj}(\sigma_{cond}(U_i \bowtie U_{d_1} \bowtie U_{d_2} \dots U_{d_k} \bowtie r_{b_1} \bowtie r_{b_2} \bowtie \dots \bowtie r_{b_n})) \end{aligned}$$

In this query expression, U_i is the current update, U_{d_j}s are deletions received after U_i, and r_{b_j}s are base relations. Notice that all the known tuples in the join expression have been placed before all base relations just for convenience. Assume U_{d_k} is the update that triggers query Q (and it is the last of all those deletions). The intended answer of Q is the answer of the expression evaluated right after U_{d_k} occurs.

$$A_{int} = \Pi_{proj}(\sigma_{cond}(U_i \bowtie U_{d_1} \bowtie U_{d_2} \ldots U_{d_k} \bowtie r_{b_1}[ss_{d_k}] \\ \bowtie r_{b_2}[ss_{d_k}] \bowtie \ldots \bowtie r_{b_n}[ss_{d_k}]))$$

At the same time, we have

$$V\langle U_i\rangle[ss_{i-1}] = \Pi_{proj}(\sigma_{cond}(U_i \bowtie r_{d_1}[ss_{i-1}] \bowtie r_{d_2}[ss_{i-1}] \ldots r_{d_k}[ss_{i-1}] \\ \bowtie r_{b_1}[ss_{i-1}] \bowtie r_{b_2}[ss_{i-1}] \bowtie \ldots \bowtie r_{b_n}[ss_{i-1}]))$$

In this expression, r_{d_j} is the corresponding base relation of U_{d_j}. Since we know $U_{d_j} \in r_{d_j}$ (only delete existing tuple) and $r_{b_j}[ss_{d_k}] \subset r_{b_j}[ss_{i-1}]$ (tuples are deleted from the relation), we can infer that $A_{int} \in V\langle U_i\rangle[ss_{i-1}]$

Since all intended answers contain a subset of tuples of $V\langle U_i\rangle[ss_{i-1}]$, $Delta$, which is the sum of all actual answers, is also a subset of $V\langle U_i\rangle[ss_{i-1}]$.

3. There are insertions and deletions after U_i.

   Let t be a tuple in $Delta$. We want to show that either t should be in the view at state ss_i, or t will be removed from $Delta$ by the algorithm. The view at state ss_i is $t \in MV[ws_{i-1}] \cup V\langle U_i\rangle[ss_{i-1}]$. Consider $BT(t)$, the set of tuples that actually derive t. There are three possible scenarios.

   (A) All tuples in $BT(t)$ (except U_i) exist in source base relations at the source state ss_{i-1}. In this scenario, we have $t \in MV[ws_{i-1}] \cup V\langle U_i\rangle[ss_{i-1}]$, because at state ss_i, those tuples in $BT(t)$ together with U_i will derive t.

   (B) There are one or more tuples in $BT(t)$ (except U_i) that do not exist in source base relations at source state ss_{i-1}. Those tuples must have been brought to $Delta$ because of insertions after U_i. Let the collection of those insertions be I. There are two subcases in this scenario:

      i. There exists at least one U_l in I such that U_l is not precede by a deletion of a tuple with the same projection attributes. In this case, when processing U_l in line 17, tuple t will be removed from $Delta$.

      ii. For all updates $U_l \in I$, there $\exists U'_l < U_l$ such that U'_l is a deletion and $\Pi_{proj}(U'_l) = \Pi_{proj}(U_l)$. In this case, t should also be in $MV[ws_{i-1}] \cup V\langle U_i\rangle[ss_{i-1}]$. We can see this from the following argument. Let the first such deletion be U''_l. We know U''_l derives t since it agrees on all projection attributes (as well as key attributes) with U_l. We also know that U''_l exists in base relation at state ss_{i-1}. All those deleted tuples (such as U''_l together with tuples in $BT(t)$ and U_i will derive t at source state ss_i.

This completes the proof for Case II(b).

Now that we have proved in Case II(a) that

$$V\langle U_i\rangle[ss_{i-1}] \subset Delta \Longrightarrow V[ss_{i-1}] \cup V\langle U_i\rangle[ss_{i-1}] \subset V[ss_{i-1}] \cup Delta \quad \text{(A.2.2)}$$

Also we have proved in Case II(b) that

$$\begin{aligned} &Delta \subset MV[ws_{i-1}] \cup V\langle U_i\rangle[ss_{i-1}] \\ &\quad \Longrightarrow MV[ws_{i-1}] \cup Delta \subset MV[ws_{i-1}] \cup V\langle U_i\rangle[ss_{i-1}] \end{aligned} \quad \text{(A.2.3)}$$

From inductive hypothesis $V[ss_{i-1}] = MV[ws_{i-1}]$, we can infer that $V[ss_{i-1}] \cup V\langle U_i\rangle[ss_{i-1}] = MV[ws_{i-1}] \cup Delta$. That is, we proved expression A.2.1.

From Case I and II, we proved $MV[ws_i] = V[ss_i]$ for any update U_i. We established a one to one relationship between the sequence of warehouse states and a sequence of consistent source states, therefore, the C-Strobe algorithm is complete. ■

Notes

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 warehouse. As stated earlier, we do not describe this straightforward extension here.
2. After T_2 commits, its updates are sent separately from sources x and y. Therefore, one update necessarily arrives before the other; in this case, U_2 arrives from y before U_3 arrives from x.

References

1. R. Alonso, D. Barbara, and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM Transaction on Database Systems*, 15(3):359–384, Sept. 1990.
2. E. Baralis, S. Ceri, and S. Paraboschi. Conservative timestamp revised for materialized view maintenance in a data warehouse. In *Workshop on Materialized Views*, pages 1–9, Montreal, Canada, June 1996.
3. P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, Massachusetts, 1987.
4. J. Blakeley, P.-A. Larson, and F. Tompa. Efficiently updating materialized views. In *SIGMOD*, pages 61–71, Washington, D.C., June 1986.
5. Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of multidatabase transaction management. *VLDB Journal*, 1(2):181–239, Oct. 1992.
6. S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *VLDB*, pages 577–589, Barcelona, Spain, Sept. 1991.
7. M. Cochinwala and J. Bradley. A multidatabase system for tracking and retrieval of financial data. In *VLDB*, pages 714–721, 1994.
8. L. Colby, T. Griffin, L. Libkin, I. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *SIGMOD*, pages 469–480, Montreal, Quebec, Canada, June 1996.
9. A. Courtney, W. Janssen, D. Severson, M. Spreitzer, and F. Wymore. Inter-language unification, release 1.5. Technical Report ISTL-CSA-94-01-01 (Xerox accession number P94-00058, Xerox PARC, May 1994.

10. R. Gallersdorfer and M. Nicola. Improving performance in replicated databases through relaxed coherency. In *VLDB*, pages 445–456, Zurich, Switzerland, Sept. 1995.
11. H. Garcia-Molina and G. Wiederhold. Read-only transactions in a distributed database. *ACM Transaction on Database Systems*, 7(2):209–234, June 1982.
12. R. Goldring and B. Hamel, Jan. 1996. Personal correspondence about IBM's data warehouse customer needs.
13. A. Gupta and I. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Engineering Bulletin, Special Issue on Materialized Views and Data Warehousing*, 18(2):3–18, June 1995.
14. A. Gupta, I. Mumick, and V. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, pages 157–166, Washington, D.C., May 1993.
15. A. Gupta and J. Widom. Local verification of global integrity constraints in distributed databases. In *SIGMOD*, pages 49–58, Washington, D.C., May 1993.
16. J. Harrison and S. Dietrich. Maintenance of materialized views in a deductive database: An update propagation approach. In *Proceedings of the 1992 JICLSP Workshop on Deductive Databases*, pages 56–65, 1992.
17. R. Hull and G. Zhou. A framework for supporting data integration using the materialized and virtual approaches. In *SIGMOD*, pages 481–492, Montreal, Quebec, Canada, June 1996.
18. W. Inmon and C. Kelley. *Rdb/VMS: Developing the Data Warehouse.* QED Publishing Group, Boston, Massachusetts, 1993.
19. W. Labio and H. Garcia-Molina. Efficient snapshot differential algorithms in data warehousing. In *VLDB*, pages 63–74, Bombay, India, Sept. 1996.
20. B. Lindsay, L. Haas, C. Mohan, H. Pirahesh, and P. Wilms. A snapshot differential refresh algorithm. In *SIGMOD*, Washington, D.C., May 1986.
21. X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *IEEE Transactions on Knowledge and Data Engineering*, 3(3):337–341, Sept. 1991.
22. A. Segev and W. Fang. Currency-based updates to distributed materialized views. In *ICDE*, pages 512–520, Los Alamitos, Feb. 1990.
23. A. Segev and J. Park. Updating distributed materialized views. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):173–184, June 1989.
24. O. Shmueli and A. Itai. Maintenance of views. In *SIGMOD*, pages 240–255, Boston, Massachusetts, May 1984.
25. Sybase, Inc. *Command Reference Manual*, release 4.9 edition, 1992.
26. G. Wiederhold and X. Qian. Consistency control of replicated data in federated databases. In *Proceedings of the IEEE Workshop on Management of Replicated Data*, pages 130–132, Houston, Texas, Nov. 1990.
27. J. Wiener, H. Gupta, W. Labio, Y. Zhuge, H. Garcia-Molina, and J. Widom. A system prototype for warehouse view maintenance. In *Workshop on Materialized Views*, pages 26–33, Montreal, Canada, June 1996.
28. Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *SIGMOD*, pages 316–327, San Jose, California, May 1995.
29. Y. Zhuge, H. Garcia-Molina, and J. Wiener. The Strobe algorithms for multi-source warehouse consistency. Technical report, Stanford University, October 1995. Available via anonymous ftp from host `db.stanford.edu` as `pub/zhuge/1995/consistency-full.ps`.
30. Y. Zhuge, J. L. Wiener, and H. Garcia-Molina. Multiple view consistency for data warehousing. In *ICDE*, Birmingham, UK, Apr. 1997.

Received Date
Accepted Date
Final Manuscript Date