INCREMENTAL MAINTENANCE OF CONSISTENT DATA WAREHOUSES

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Yue Zhuge
June 1999

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Héctor García-Molina
(Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Jennifer Widom

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Jeffrey D. Ullman

Approved for the University Committee on Graduate Studies:

Abstract

A data warehouse is a data repository containing integrated information for efficient querying and analysis. Warehousing is the technique for retrieval and integration of data from distributed and possibly heterogeneous information sources. In effect, the warehouse stores materialized views over the source data. This dissertation studies the maintenance of warehouse views as the data sources are updated.

The first part of this dissertation presents a family of algorithms that incrementally and consistently maintain relational materialized views in a data warehouse. This view maintenance problem differs from the traditional one in that the view definition and the base data are decoupled, and data sources are autonomous. We show that this decoupling can result in anomalies if traditional algorithms are applied. We formalize notions of consistency for warehouse views and present new algorithms that maintain consistency as the warehouse is updated. In addition, we develop simple, scalable, algorithms for ensuring mutual consistency among multiple views at a warehouse. We also present the implementation of the algorithms in the WHIPS (WareHousing Information Project at Stanford) prototype and related performance results.

The second part of this dissertation studies how to maintain graph-structured materialized views. A graph-structured database consists of records containing identifiers of other records. The data could represent semi-structured information such as Web pages, documents, XML data, or data integrated from heterogeneous data sources. We define views and materialized views for such graph-structured data, analyzing options for representing record identity and references in the view. We then develop incremental maintenance algorithms for these views, discuss how to realize these algorithms in a data warehouse, and study how to maintain the warehouse views without accessing base data.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Today's businesses require the ability to consolidate information efficiently and automatically from a variety of sources. In this dissertation we study how to consistently integrate data into a summarized view and how to reflect changes of source data incrementally. We set our framework in a data warehousing environment. A *data warehouse* is a repository of integrated information available for queries and analysis (e.g., browsing, searching, mining, decision support). *Warehousing* is the technique for retrieval and integration of data from distributed, autonomous, possibly heterogeneous, information sources.

In this chapter we give examples of the problems that one may encounter when managing data in a warehousing environment, compare different data integration techniques, and highlight the approaches we propose.

1.1 Data Warehousing

We start by giving two examples of how data warehouses can be used to solve business problems. The scenario in Example 1.1.1 was first presented in [Mic98].

EXAMPLE 1.1.1 (Personal Investment Management)

Investors need access to personalized information regarding their portfolios. Currently, financial institutions hand-craft portfolio reports only for large companies or very important individuals. Demand for such reports is increasing from a broader range of individuals participating in group investment plans. To meet these demands, institutions must develop an automated service that combines data from various sources, such as Moody's Investors Service, Standard and Poor's, EDGARS, financial analysts, and personal profiles. In addition, customers are looking for more up-to-the minute information about their investments.

To provide better service to the investors, investment managers can build a data warehouse that caches and summarizes the data relevant to the investors. A data warehouse can provide a customized set of data (called *views*) for each investor. The set of views are specified in a simple, declarative language. They may refer to data cached at the warehouse (e.g., stock quotes or the investor's portfolio) or data at other sources (e.g., company profiles). Once specified, the warehouse will make sure that the views reflect any changes in the data source automatically. For example, a view may combine the information from an investor's portfolio with current stock prices to compute his or her total assets. This value will change with variations of stock prices or of the portfolio, but the change coordination will be managed by the warehouse. The investors (i.e., warehouse users) will always be supplied with current information and notified about events concerning their financial interests, without the need to manipulate data manually. □

EXAMPLE 1.1.2 (Targeted Advertisement)

Consider an organization (e.g., American Collegiate Marketing) that handles magazine subscriptions. The subscription information, like the subscribers' names and addresses, may be stored in a database and managed by one department of the organization. This data can also be bought from or exchanged with other subscription agencies. A separate department may manage information about the magazines themselves, such as their publishers, prices, and availabilities. The company also needs to manage historical information such as previous subscriptions, subscriber demographics, and other statistics. Sometimes free gifts are sent to subscribers to promote certain products: free toys can be sent to readers of Parent magazine, AOL CDs for Home PC readers, and educational software for those who subscribe to both. Promotions may be decided among the subscription agency, the magazine publishers, and the company (or companies) that produce the products.

In order to handle the shipping and bookkeeping of promotional products, this organization may set up a data warehouse. The data sources may be distributed (e.g., from different departments) and heterogeneous (e.g., from different companies). The warehouse provides a set of views on top of the source data. For example, a view may contain the addresses of people who should receive free toys, or the total numbers of different free gifts sent out so far. Once the data warehouse is set up, it provides a complete and clear picture of all the current promotions. If desired, the warehouse will also continuously or periodically provide the list of new subscribers and what gifts they should receive. Furthermore, the warehouse may combine historical data (e.g., those who bought products from the catalog sent) with current information (e.g., a company's new promotions) to enable the organization to make more profitable decisions. □

In general, a data warehouse can be used for storing cleaned, summarized, analytical data; for storing historical data or backup data; or for caching query results for fast response time. The characteristics of warehouse data is that it is read-only to the warehouse users, and that it reflects the content of some source data. We believe these two features capture the most important aspects of a data warehouse, regardless of how it is being used. One can think of a data warehouse as defining and storing integrated *materialized views* over the data from multiple, autonomous information sources [LW95].

1.2 View Maintenance Anomaly

One of the major challenges is to correctly and efficiently update the warehouse as the source data changes. That is, as relevant information becomes available from a source, or when relevant information is modified, the information must be extracted from the source, translated into a common model (e.g., the relational model), and integrated with existing data at the warehouse. Then, queries can be answered and analysis can be performed quickly and efficiently, since the integrated information is directly available at the warehouse, with any differences already resolved.

Numerous methods have been developed for materialized view maintenance in conventional database systems; these methods are discussed in Chapter 2. Unfortunately, existing materialized view maintenance algorithms fail in a warehousing environment. Existing approaches assume that view maintenance is done in the same transaction as the source update, so that the transaction holds locks on all source data necessary to maintain all views until maintenance is completed.

However, in a warehousing environment such as in Examples 1.1.1 and 1.1.2, the sources are often autonomous. They can also be legacy or unsophisticated systems that do not understand views. Sources can inform the warehouse when an update occurs, e.g., when a new employee has been hired or a patient has paid her bill. However, they cannot lock their data for the purpose of view maintenance. When the update information arrives at the warehouse, the warehouse may discover that additional source data is necessary to update the views. Thus, the warehouse may have to issue queries to some of the sources, as illustrated in Figure 1.1. Because the source data may be updated by another source transaction, a warehouse query may be evaluated at a source state later than the one at which the query is intended to be evaluated. This decoupling between the base data on the one hand (at the sources) and the view definition and view maintenance machinery on the other (at the warehouse) can lead the warehouse to compute incorrect views.

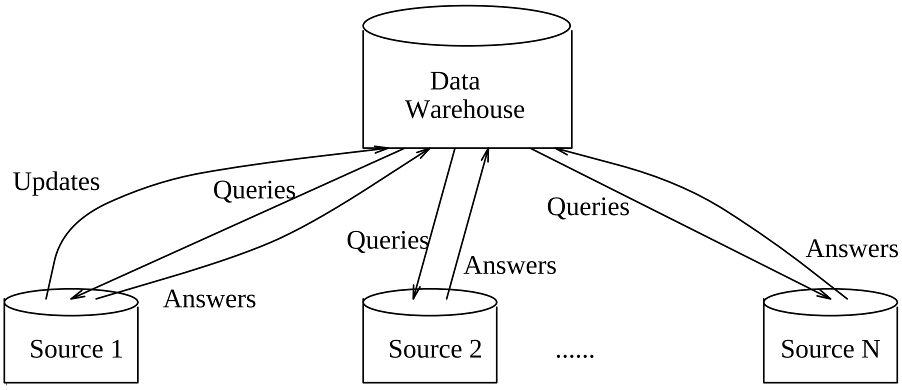


Figure 1.1: Update processing in a warehouse

We illustrate the problems with three examples. For these examples we use the relational model for data and relational algebra select-project-join queries for views; each update therefore consists of a single tuple action, such as inserting or deleting a tuple. Although we are using relational algebra, we assume that duplicates are retained in the materialized views. Duplicate retention (or at least a replication count) is essential if it is required by the semantics of the view, if deletions are to be handled incrementally [BLT86, GMS93], and if aggregate views are to be defined using the existing materialized views. Note that the type of solution we propose here can be extended to other data models and view specification languages.

EXAMPLE 1.2.1 (Correct Incremental View Maintenance)
Consider the subscription agency in Example 1.1.2. One department (S_1) maintains subscription information in a relation $S = \mathit{Subscription}(Name, Magazine)$, and a second department (S_2) maintains promotion information $P = \mathit{Promotion}(Magazine, Gift)$. A third department, in charge of shipping and handling, needs to find out what free gifts to send to each subscriber. This information can be obtained by examining view $V = \Pi_{Name,Gift}(S \bowtie P)$. View V can be materialized in a data warehouse with the two department databases as data sources.

To maintain V incrementally, one computes increments of the view, so-called *view deltas*, when source updates occur. The view delta corresponding to an update U_s on relation S is $\triangle V = \Pi_{Name,Gift}(U_s \bowtie P)$, and that corresponding to an update U_p on relation P is $\triangle V = \Pi_{Name,Gift}(S \bowtie U_p)$. Whether the view delta should be inserted into or deleted from the view is determined by whether the source update is an insertion or a deletion.

Suppose the source relations are initialized as follows:

At S_1, S :

Name	Magazine
Joe	Time

At S_2, P :

Magazine	Gift
Time	CD

Initially the materialized view at the warehouse, MV, contains the single tuple $[Joe, CD]$, meaning that Joe should receive the promotional gift CD because of his subscription to Time magazine. Now suppose that tuple $[\underline{Time}, book]$ is inserted into P at source S_2. For notation, we use $insert(r,t)$ to denote the insertion of tuple t into relation r, (similarly for $delete(r,t)$), and we use $([t_1],[t_2],\ldots,[t_n])$ to denote a relation with tuples $t_1, t_2, \ldots, t_n$. The following events occur:

1. Update $U_1 = insert(P, [\underline{Time}, book])$ occurs at source S_2. Since the source does not know the details or contents of the view managed by the warehouse, it simply sends a notification to the warehouse that update U_1 occurred.

2. The warehouse receives U_1. Applying an incremental view maintenance algorithm, it sends query $Q_1 = \Pi_{Name,Gift}(S \bowtie [\underline{Time}, book])$ to source S_1. (That is, the warehouse asks the source S_1 which S tuples match with the new $[\underline{Time}, book]$ tuple in P.)

3. Source S_1 receives Q_1 and evaluates it using the current base relations. It returns the answer relation $A_1 = ([Joe, \mathit{book}])$ to the warehouse.

4. The warehouse receives answer A_1 and adds $([Joe, \mathit{book}])$ to the materialized view, obtaining $([Joe, CD], [Joe, \mathit{book}])$.

The final view at the warehouse is correct, or, it is equivalent to what one would obtain using a conventional view maintenance algorithm directly at the source. The next two examples show how the final view can be incorrect. □

EXAMPLE 1.2.2 (A View Maintenance Anomaly)
Assume we have the same relations as in Example 1.2.1, but initially P is empty:

S :

Name	Magazine
Joe	<u>Time</u>

P :

Magazine	Gift
–	–

Consider the same view definition as in Example 1.2.1: $V = \Pi_{Name,Gift}(S \bowtie P)$, and now suppose there are two consecutive updates: $U_1 = \mathit{insert}(P, [\underline{Time}, \mathit{book}])$ and $U_2 = \mathit{insert}(S, [Amy, \underline{Time}])$. The following events occur. Note that initially $MV = \phi$.

1. Source S_2 executes $U_1 = \mathit{insert}(P, [\underline{Time}, \mathit{book}])$ and sends U_1 to the warehouse.

2. The warehouse receives U_1 and sends $Q_1 = \Pi_{Name,Gift}(S \bowtie [\underline{Time}, \mathit{book}])$ to S_1.

3. Source S_1 executes $U_2 = \mathit{insert}(S, [Amy, \underline{Time}])$ and sends U_2 to the warehouse.

4. The warehouse receives U_2 and sends $Q_2 = \Pi_{Name,Gift}([Amy, \underline{Time}] \bowtie P)$ to source S_2.

5. Source S_1 receives Q_1 and evaluates it on the current base relations: $S = ([Joe, \underline{Time}], [Amy, \underline{Time}])$. The resulting answer relation is $A_1 = ([Joe, \mathit{book}], [Amy, \mathit{book}])$, which is sent to the warehouse.

6. The warehouse receives A_1 and updates the view to $MV \cup A_1 = ([Joe, \mathit{book}], [Amy, \mathit{book}])$.

7. Source S_2 receives Q_2 and evaluates it on the current base relation P. The resulting answer relation is $A_2 = ([Amy, \mathit{book}])$, which is sent to the warehouse.

8. The warehouse receives answer A_2 and updates the view to $MV \cup A_2 = ([Joe, \mathit{book}], [Amy, \mathit{book}], [Amy, \mathit{book}])$[1].

If the view had been maintained using a conventional algorithm directly at the source, then it would be $([Joe, \mathit{book}])$ after U_1 and $([Joe, \mathit{book}], [Amy, \mathit{book}])$ after U_2. However, the warehouse first changes the view to $([Joe, \mathit{book}], [Amy, \mathit{book}])$ (in Step 6) and to $([Joe, \mathit{book}], [Amy, \mathit{book}], [Amy, \mathit{book}])$ (in Step 8), which is an incorrect final state. The problem is that the query Q_1 issued in Step 2 is evaluated using a state of the base relations that differs from the state where Q_1 is intended to be evaluated (before update U_2 occurs). □

We call the behavior exhibited by this example a *distributed incremental view maintenance anomaly*, or *anomaly* for short. Anomalies occur when the warehouse attempts to update a view while base data at the source is changing. Anomalies arise in a warehousing environment because of the decoupling between the information sources, which are updating the base data, and the warehouse, which is performing the view update.

EXAMPLE 1.2.3 (Deletion Anomaly)

Our third example shows that deletions can also cause anomalies. Consider source relations:

S :

Name	Magazine
Joe	<u>Time</u>

P :

Magazine	Gift
<u>Time</u>	CD

Suppose that the view definition is the same as in Example 1.2.1. The following events occur. Note that initially $MV = ([Joe, CD])$.

1. Source S_1 executes $U_1 = \mathit{delete}(S, [Joe, \underline{Time}])$ and notifies the warehouse.

2. The warehouse receives U_1 and sends $Q_1 = \Pi_{Name,Gift}([Joe, \underline{Time}] \bowtie P)$ to S_2.

3. Source S_2 executes $U_2 = \mathit{delete}(P, [\underline{Time}, CD])$ and notifies the warehouse.

4. The warehouse receives U_2 and sends $Q_2 = \Pi_{Name,Gift}(S \bowtie [\underline{Time}, CD])$ to S_1.

5. Source S_2 receives Q_1. The answer it returns is $A_1 = \phi$, since relation P is now empty.

[1]As stated earlier, we need to keep both $[Amy, \mathit{book}]$ tuples in the view, because it may be true that Amy should receive two books as the result of subscribing to two different magazines.

6. The warehouse receives A_1 and replaces the view with $MV - A_1 = ([Joe, CD])$. (Difference is used here since the update to the base relation was a deletion [BLT86].)

7. Similarly, source S_1 evaluates Q_2, returns $A_2 = \phi$, and the warehouse replaces the view by $MV - A_2 = ([Joe, CD])$.

This final view is incorrect: Since S and P are empty, the view should be empty too. □

We want to point out here that the anomaly problem does not arise only in the data warehousing environment. It may occur in any system that tries to maintain derived data (e.g., indices, caches, or replicated data) that is decoupled from data sources. However, it is convenient for us to discuss the problem by referring to the warehouse and the sources. In the next section we briefly describe how we can avoid view maintenance anomalies.

1.3 Solutions

There are a number of mechanisms for avoiding view maintenance anomalies. As argued above, we are interested only in mechanisms where the source does not perform any "view management." We also are not interested in, for example, solutions where the source must lock data while the warehouse updates its views, or in solutions where the source must maintain timestamps or versions in order to answer warehouse queries correctly. The source will only notify the warehouse of relevant updates and answer queries asked by the warehouse. We propose the following two approaches, in both of which view maintenance is autonomous from source updating:

- *Use compensating queries.* The basic idea is to add to queries sent to the source *compensating queries* to offset the effect of concurrent updates. For instance, in Example 1.2.2, consider the receipt of $U_2 = insert(S, [Amy, \underline{Time}])$ in Step 4. If we assume that messages are delivered in order and that the sources handle requests atomically, then when the warehouse receives U_2 it can deduce that its previous query Q_1 will be evaluated in an "incorrect" state—Q_1 will see the $[Amy, \underline{Time}]$ tuple of the second insert. (Otherwise, the warehouse would have received the answer to Q_1 before it received the notification of U_2.) To compensate, the warehouse sends query:

$$Q_2 = \Pi_{Name,Gift}([Amy, \underline{Time}] \bowtie P) - \Pi_{Name,Gift}([Amy, \underline{Time}] \bowtie [\underline{Time}, book])$$

The first part of Q_2 is as before; the second part compensates for the extra tuple that Q_1 will see. We see that indeed the answer received in Step 6, $A_1 = ([Joe, book], [Amy, book])$, contains the extra tuple $[Amy, book]$. But, because of the compensation, the A_2 answer received in Step 8 is empty, and the final view is correct. In Chapters 3 and 4 we provide algorithms ECA and LCA (for single-source warehouses), MECA (for multiple-source warehouses), and other algorithms that use compensating queries to provide correct view maintenance. We show how the compensating queries are determined for an arbitrary view, and how query answers are integrated into the view.

- *Use key information.* If a view includes a key from every base relation involved in the view, then we can streamline the algorithms in two ways: (1) Deletions can be handled at the warehouse without issuing a query to the source, and (2) insertions require queries to the source, but compensating queries are unnecessary. To illustrate point (1), consider Example 1.2.3, and suppose $Name$ and $Gift$ are keys for S and P. When the warehouse receives $U_1 = delete(S, [Joe, \underline{Time}])$ (Step 1), it can immediately determine that all tuples of the form $[Joe, x]$ have been deleted from the view (where x denotes any value)—no query needs to be sent to the source. Similarly, $U_2 = delete(P, [\underline{Time}, book])$ causes all $[x, book]$ tuples to be deleted from the view, without querying the source. The final empty view is correct. Section 3.3.4, in which we describe our view maintenance algorithm using keys, illustrates point (2). In Chapter 3 and 4 we design algorithms ECA-Key (for single-source warehouses), Strobe (for multiple source warehouses), and other algorithms that use key information to provide correct view maintenance. Note that these algorithms do have the disadvantage that they can only be used for a subset of all possible views—those that contain keys for all base relations.

The view maintenance algorithms we develop in Chapters 3 and 4 handle all possible source update interleaving scenarios correctly. Variations of the algorithms, such as handling batches of source updates together or evaluating compensating queries locally at the warehouse, are also discussed in those chapters.

1.4 Comparison of View Maintenance Techniques

In this section we first compare incremental view maintenance with recomputation, then discuss an orthogonal approach that copies source data at the warehouse.

In a warehousing environment, incremental maintenance means that as source data changes, new information is gathered and integrated into the warehouse without throwing away all the existing data. Example 1.2.1 is an example of incremental maintenance. This method can be performed for each single update or for a batch of source updates (*batch-incremental*). Recomputation (*complete refresh* or *refresh*) means that the entire warehouse is recomputed, either on demand or periodically.

Most current warehousing systems use refresh because of its simplicity. However, incremental maintenance has important advantages that may make it more attractive than recomputation. For example:

- First, some applications may wish to observe the actual change of views, not just periodic snapshots. For example, in order to make shipments of promoted products, the subscription agency in Example 1.2.1 needs the list of new subscribers together with their free gifts and other information. Incremental view maintenance provides these view deltas that can be used by the warehouse applications while recomputation does not. The view deltas can be made available in different granularities. For example, algorithms ECA (Chapter 3), Strobe, and T-Strobe (Chapter 4) compute batches of view deltas at a time, while algorithms CCA (Chapter 3) and CStrobe (Chapter 4) enable the warehouse view to reflect each single change of source data.

- Second, a complete refresh may be infeasible because of its duration. In particular, a warehouse may be unavailable to users during recomputation. Therefore, recomputation is often done at night or on a weekend. However, for large warehouses, the refresh time may exceed the available "update window" of a few hours, because a large amount of data might need to be transferred from the sources to the warehouse. Furthermore, international corporations in multiple time zones may need 24×7 operation, without any downtime. In contrast, incremental maintenance performs a relatively small number of operations on each update or each batch of updates, so it may be done concurrently with warehouse user transactions.

- Third, some applications, such as one that tracks stock market changes to perform trading or computes current assets for investors as in Example 1.1.1, may need updates to be reflected quickly. Only incremental maintenance may provide this data freshness.

Other advantages offered by incremental maintenance include reduced network traffic and less storage space for intermediate results. Incremental maintenance has its disadvantages, however. In particular, some operations, like cleaning, must be done on each update or batch, and hence may result in larger maintenance overhead. (On the other hand, cleaning needs to be done for both old and new data for refresh.) Also, incremental maintenance may lead to anomaly problems as we showed in Section 1.2. We designed algorithms to solve the anomaly problems, but they are more complicated to understand and implement than simple recomputation.

Batch-incremental, the incremental integration method for batches of source updates, may be the most practical approach for view maintenance. This is because on one hand, updates are often available in batches in reality (e.g., all updates produced by the same transaction or within an hour), while on the other hand, the batch-incremental method only incorporates new data into the warehouse, and hence has the advantages of the incremental approach discussed above. Batch-incremental encounters the same anomaly problem as the single-update incremental approach, and the problem can also be handled by the algorithms we develop in Chapters 3 and 4.

In Section 3.6 we compare the predicted performance of recomputation with one of the incremental algorithms we develop. In Chapter 6 we further study performance tradeoffs between different incremental approaches.

A third approach that is orthogonal to both incremental integration and recomputation is to store at the warehouse copies of all relations involved in views. The user-defined materialized views can then be stored as derived data on top of those copies. In the case of Example 1.2.2, suppose that the warehouse keeps up-to-date copies of S and P. When U_1 arrives, Q_1 can be evaluated "locally," and no anomaly arises. The stored copies need to be maintained, either incrementally or by recomputation. The derived (summarized) data also needs to be maintained, either incrementally or by recomputation. Depending on how the warehouse data is maintained, this approach inherits some of the advantages and disadvantages of the first two methods.

While this "copying source" approach may be adequate for some applications, it also has several disadvantages. First, although storage is not very costly, the storage requirement at the warehouse may be very high. For instance, suppose that a relation C contains data on companies, e.g., their name, stock price, and profit history. If we copy all of C 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 we could be unable to update the warehouse automatically.) 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 [CB94], much of it for 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. In this dissertation, we focus on incremental view maintenance without storing copies of all base data at the warehouse.

1.5 Dissertation Outline

The next chapter surveys research work related to this dissertation. In Chapter 3 and Chapter 4, we design incremental view maintenance algorithms for data warehouses with single and multiple data sources, respectively. In Chapter 5 we study how to maintain multiple views in a consistent fashion. Chapter 6 presents a performance analysis of the integration algorithms. The results are obtained in the context of WHIPS (WareHousing Information Project at Stanford) prototype. Chapter 7 studies the incremental view maintenance problem for graph-structured data, and Chapter 8 discuss how to maintain graph-structured views without accessing source data (*self-maintenance*). Finally, Chapter 9 summarizes the contributions of this dissertation and discusses future work.

Chapter 2

Related Work

In this chapter we briefly survey the related previous research. We classify the related work into four categories: data warehousing, materialized views, concurrency control and consistency for replicated data, and graph-structured databases. We discuss each of these fields in turn, then discuss a few related prototype systems, and finally mention some research that was partially inspired or based on work presented in this dissertation. Detailed comparisons between approaches taken in the dissertation and those of previous work are given in the relevant chapters.

2.1 Data Warehousing

Data warehouses are large repositories for analytical data and have recently generated tremendous interest in industry (e.g., [DDJ$^+$98, CJ98]). An early description of the data warehousing idea may be found in [IK93]. Because the market is new and extremely promising, almost all database vendors are targeting their existing products to data warehousing applications, and others have built specialized tools to extract, load, clean, and transform data into the warehouse. [SC97] provides a comprehensive survey of recent work in data warehousing and OLAP (on-line analytical processing) technology, as well as a list of references in this area. One can also find more detailed discussions on several projects in the area of warehousing and materialized views in [LW95].

2.2 Materialized Views

With the increased popularity of data warehousing, materialized views have attracted much attention as an important technique for managing warehouse data. Discussions on warehousing-related materialized view problems can be found in [LW95, GM96].

Many incremental view maintenance algorithms have been developed for centralized database systems [BLT86, GMS93, CW91, HD92, QW91, SI84, CGL+96, GL95, SJ96]. Overviews of materialized views and their maintenance can be found in [GM95, GM98, LW95]. Most of the algorithms are designed for a traditional, centralized database environment, where it is assumed that view maintenance is performed by a system that has full control and knowledge of the view definition, the base relations, the updated tuples, and the view. These algorithms differ somewhat in the view definitions they handle. For example, [BLT86] considers select-project-join (SPJ) views only, algorithms in [GMS93] handle views defined by any SQL or Datalog expression, and [GBLP96, Qua96] addresses aggregate view maintenance. Some algorithms depend on key information to deal with duplicate tuples [CW91], while others use a counting approach [GMS93]. As we showed in Section 1.2, when a centralized algorithm is applied to the warehouse, the warehouse user may see inconsistent views of the source data.

A series of papers by Segev et al. studies materialized views in distributed systems [SF90, SF91, SP89a, SP89b]. All algorithms in these papers are based on timestamping the updated tuples, and the algorithms assume there is only one base table. Other incremental algorithms, such as the "snapshot" procedure in [LHM+86], also assume timestamping and a single base table. In contrast, our algorithms have no restrictions on the number of base tables, and they require no additional information. Note that although we describe our algorithms for SPJ views, our approach can be applied to adapt any existing centralized view maintenance algorithm to the warehousing environment. (See Chapter 9 for more discussion.)

In both centralized and distributed systems, there are three general approaches to the timing of view maintenance: *immediate* update [BLT86], which updates the view after each base relation is updated, *deferred* update [RK86, CGL+96], which updates the view only when a query is issued against the view, and *periodic* update [LHM+86], which updates the view at periodic intervals. Performance studies on these strategies have determined that efficiency depends heavily on the structure of the base relations and on update patterns [Han87]. We assume immediate update, but we observe that with little or no modification our algorithms can be applied to deferred and periodic update as well. View maintenance anomalies arise regardless of whether the centralized algorithm computes changes using the old base relations or using the new base relations. 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.

2.3 Concurrency Control and Consistency

A warehouse holds a copy of "summarized" source data, so essentially we have a distributed database system with replicated data. A good overview of mechanisms for managing replicated data may be found in [BHG87]. For example, in a multi-database system [BGMS92, LMR90], updating the global "view" is within the same global transaction as reading source base data and therefore multiple views can be updated consistently in a single global transaction. Other systems use mechanisms such as *primary copy*, *group vote* or *eager propagation* to ensure atomic commit of updates [GHOS96, BK97]. We consider a more loosely coupled system where data sources are autonomous, and traditional concurrency mechanisms are often not applicable. A variety of concurrency control schemes have been suggested over the years for such environments (for example, [ABGM90, WQ90, GN95]). They either provide weaker notions of consistency, or exploit the semantics of applications. The algorithms we present in this dissertation exploit the semantics of materialized view maintenance to obtain consistency without traditional distributed concurrency control (e.g., locking).

Regarding consistency of derived data (mostly replicated data), most of the previous work tries to ensure serializability (e.g., *1SR* [BHG87], *Mat-Serializable* [KLM+97]) or some weaker notion of serializability (e.g., lower isolation degrees [GR93]). This aspect of data consistency is captured by our definition of *snapshot consistency* (Section 4.3). But we go one step further and define a variety of consistency levels that reflect the correspondence between the derived and base data (Sections 3.1 and 4.3). This second aspect of data consistency is specific to derived data and has not been identified by previous work, to our knowledge.

2.4 Graph-structured Data

In Chapter 7 and Chapter 8 we study views defined using a data model called *graph-structured database* (GSDB). Our work is related to previous work on semi-structured data models [Suc97, QRUW95] and querying object oriented databases [CD92, KKS92, KS95, Cat94]. We will compare that work in detail in the related chapters, after we discuss our data model and views.

2.5 Prototypes

Most of the work presented in this dissertation is implemented in a warehouse prototype called WHIPS (WareHousing Information Project at Stanford). The prototype uses incremental algorithms to automatically and consistently maintain its warehouse. In [HGMW+95, WGL+96] we presented short preliminary descriptions of the WHIPS architecture, applications, and implementation. Some work in this dissertation has been published in [ZGMHW95, ZGMW96, ZWGM97, ZGMW98, ZGM98a].

Another incremental maintenance prototype is the Squirrel Project [HZ96a]. It provides a framework for supporting distributed data integration using materialized views. However, their approach first materializes the relevant portions of each base relation, 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. A related preliminary performance study comparing the performance of virtual and materialized approaches is given in [HZ96b].

In another project ADMS (Adaptive Database Management System) [RK86], Roussopoulos et al. introduce the idea of a *ViewCache*, which caches and reuses intermediate and final query results to evaluate later queries. Each ViewCache maintains a pointer array consisting of pointers to those tuples of the base relations or other views needed to construct the view. Then, ViewCaches are maintained incrementally using update logs.

2.6 Other Related Research

There is research that address different aspects of warehouse maintenance. For example, [HRU96, GHRU97] discuss how to select views or indices to materialize, and [QW97] discusses how to use versioning techniques to perform maintenance without blocking warehouse user read transactions.

Published work presented in this dissertation has been extended by other researchers. For example, [AASY97] studies how to use the idea of compensation to maintain materialized views, but compensates without going back to data sources. Similarly, [CM97] studies how to solve the anomaly problem when messages between two sites could be out of order. Our work has also served as an incentive for some later research; for example, to avoid getting into complicated anomaly problems, it is desirable to make views *self-maintainable* by using auxiliary information or view content [QGMW96, Huy96], or by making partial copies of base data at the warehouse [HZ96a].

Chapter 3

View Maintenance in a Single-source Warehouse

As we showed in Chapter 1, existing materialized view maintenance algorithms generally fail in a warehousing environment. In this chapter we develop algorithms that correctly maintain materialized views for a warehouse with a single data source. Working in this simple environment, we are able to introduce the key ideas of the algorithms. In the next chapter, we extend the algorithms to handle a more realistic multi-source environment with more complex source transaction scenarios.

The warehouse architecture we study in this chapter is shown in Figure 3.1. The view maintenance anomaly may occur in this environment, because a query may be evaluated at the source later than the corresponding update. For instance, anomalies shown in Examples 1.2.2 and Examples 1.2.3 remain had relations S and P resided at the same data source.

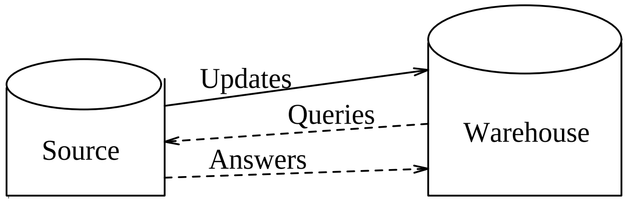


Figure 3.1: Update processing in a single-source warehouse

In the first section of this chapter we provide a formal definition of correctness for view maintenance in a warehousing environment. As we will see, there are a variety of "levels" of correctness that may be of interest to different applications. In Section 3.2 we present basic concepts used by the algorithms. In Sections 3.3, 3.4 and 3.5 we present three view maintenance algorithms ECA (Eager Compensating Algorithm), LCA (Lazy Compensating Algorithm), and CCA (Complete Compensating Algorithm), along with a number of examples. All three algorithms use *compensating queries* as we previewed in Chapter 1 to correct view maintenance anomalies, but they compensate differently and provide different levels of consistency. In Section 3.3 we also provide an algorithm ECA^K to show how key information can be used instead of compensating queries. In Section 3.6 we compare the performance of our ECA algorithm to the view recomputation approach. Finally, in Section 3.7 we discuss general properties of compensating algorithms developed in this chapter.

3.1 Correctness

In this chapter we assume each source update consists of a single tuple action such as inserting or deleting a tuple. Our first task is to define what correctness means in an environment where activity at the source is decoupled from the view at the warehouse. 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. Thus, we believe it is important to offer customers a variety of consistency options and ways to enforce them.

3.1.1 Events

We start by defining the notion of *events*. In our context, an event corresponds to a sequence of operations performed at the same site. There are two types of events at the source:

1. S_up: the source executes an update U, then sends an update notification to the warehouse.

2. S_qu: the source evaluates a query Q using its current base relations, then sends the answer relation A back to the warehouse.

There are two types of events at the warehouse:

1. W_up: the warehouse receives an update U, generates a query Q, and sends Q to the source for evaluation.

2. W_ans: the warehouse receives the answer relation A for a query Q and updates the view based on A.

We will assume that events are atomic. That is, we assume there is a local concurrency control mechanism (or only a single user) at each site to ensure that conflicting operations do not overlap. With some extensions to our algorithms, this assumption could be relaxed. We also assume that, within an event, actions always follow the order described above. For example, within an event S_up, the source always executes the update operation first, then sends the update notification to the warehouse.

We use e to denote an arbitrary event, se to denote a source event, and we to denote a warehouse event. Event types are subscripted to indicate a specific event, e.g., S_up_i, or W_up_j. For each event, relevant information about the event is denoted using a functional notation: For an event e, $query(e)$, $update(e)$, and $answer(e)$ denote respectively the query, update, and answer associated with event e (when relevant). If event e is caused ("triggered") by another event, then $trigger(e)$ denotes the event that triggered e. For example, $trigger(W\_up_j)$ is an event of type S_up. The state of the data after an event e is denoted by $state(e)$.

It is useful to immediately rule out algorithms that are trivially incorrect; for example, where the source does not propagate updates to the warehouse, or refuses to execute queries. These two examples are captured formally by the following rules:

- $\forall U = update(S\_up_i)$: $\exists\ W\_up_j$ such that $update(W\_up_j) = U$.

- $\forall Q = query(W\_up_i)$: $\exists\ S\_qu_j$ such that $query(S\_qu_j) = Q$.

There are a number of other obvious rules such as these that we omit for brevity.

Finally, we define the binary event operator "$<$" to mean "occurs before". We assume that messages are delivered in order and are processed in order. In particular, let e_1, e_2, e_3, e_4 be four events. If $trigger(e_2) = e_1$, $trigger(e_4) = e_3$, and e_1 and e_3 occurred at the same site, then $e_1 < e_3$ iff $e_2 < e_4$. We also use the $<$ relation to order states. That is, we say that $s_i < s_j$ if $state(e_i) = s_i$, $state(e_j) = s_j$, and $e_i < e_j$. We will discuss more on message and event ordering in the next chapter.

3.1.2 Levels of Correctness for View Maintenance

Materialized views are derived data. The content of a materialized view should reflect the content of the base data it is derived from. Intuitively, correctness means that the warehouse view "makes sense", that is, it reflects (e.g., is a copy or summary of) an actual source state at some "recent time". When base data changes, the view should reflect the new values of base data as well as the intermediate values, if any. As we will see, there are a variety of "levels" of correctness that may be of interest to different applications.

During the execution of a view maintenance algorithm, the system will process a sequence of updates $U_1, U_2, \ldots, U_n$. In doing so, the source executes events $se_1, se_2, \ldots se_p$ with corresponding resulting states $ss_1, ss_2, \ldots, ss_p$. At the warehouse the triggered events are $we_1, we_2, \ldots we_q$ with corresponding resulting states $ws_1, ws_2, \ldots, ws_q$. When we define the notion of *convergence* (below), we consider executions that are finite, i.e., that have a last update U_n and last events se_p and we_k. However, in general executions may be finite or infinite.

At the warehouse, the state of materialized view V after event we_i is given by $V[ws_i]$, where V is the view definition. Similarly, at the source, $Q[ss_i]$ is the result of evaluating query expression Q on the relations existing after event se_i. If we apply the view definition V to a source state, $V[ss_i]$, we get the state of the view had it been evaluated at the source after event se_i.

An algorithm for warehouse view maintenance may exhibit the following properties:

- *Convergence*: For all finite executions, $V[ws_q] = V[ss_p]$. That is, after the last update and after all activity has ceased, the view is consistent with the relations at the source.

- *Weak Consistency:* Convergent, and for all executions and for all ws_i, there exists an ss_j such that $V[ws_i] = V[ss_j]$. That is, every state of the view corresponds to some valid state of the relations at the source.

- *Strong Consistency:* Convergent, and for all executions and for every pair $ws_i < ws_j$, there exist $ss_k \leq ss_l$ such that $V[ws_i] = V[ss_k]$ and $V[ws_j] = V[ss_l]$. That is, every state of the view corresponds to a valid source state, and in a corresponding order.

- *Completeness:* Strong consistency, and for each ss_i there exists a ws_j such that $V[ws_j] = V[ss_i]$. That is, there is a complete order-preserving mapping between the states of the view and the states of the source.

Although completeness is a nice property since it states that the view "tracks" the base data exactly, sometimes it is 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 other cases, consistency (weak or strong) may be required, i.e., knowing that every warehouse state is valid with respect to some source state. Because a strongly consistent algorithm can batch intertwined updates, it is often more desirable in practice. Examples 1.2.2 and 1.2.3 showed that a conventional incremental view maintenance algorithm is not even weakly consistent in the environments we consider. In the next few sections, we will develop strongly consistent and complete view maintenance algorithms.

Our consistency definitions are compatible with standard serializability theory. In fact, all four definitions can be rephrased in terms of serializability theory, by treating the warehouse view evaluation as a read only transaction [GMW82] at the sources.

The definition of consistency for replicated data in [HZ96a] is similar to one of the levels we defined: strong consistency. Other definitions of materialized view consistency (e.g., [BCP96]) are based on the database state, a global clock, timestamps, or transaction isolation properties. However, most of these definitions can also be adapted to obtain one or more levels of consistency we just defined.

3.2 Views and Queries

Before presenting our algorithms, we must define the warehouse views we handle and the types of queries generated. In this chapter, we consider views defined as:

$$V = \Pi_{proj}(\sigma_{cond}(r_1 \times r_2 \times \ldots \times r_n))$$

where *proj* is a set of attribute names, *cond* is a boolean expression, and $r_1, \ldots, r_n$ are base relations. Note that any relational algebra expression constructed with select, project, and join operations can be transformed into an equivalent expression of this form. For simplicity, we assume that $r_1, \ldots, r_n$ are distinct relations. Our algorithms can be extended to allow multiple occurrences of the same relation (e.g., by handling updates to such relations once for each appearance of the relation). We believe that SPJ views are the most common and therefore are good subproblems on which to focus initially. Once the problems and solutions are well understood for SPJ views, one may use similar approach to handle views defined by other operators. We will come back to this discussion in Chapter 9.

3.2.1 Signs

Our warehouse algorithms will handle two types of updates: insertions and deletions. (Modifications must be treated as deletions followed by insertions[GM95], although extensions to our approach could permit modifications to be treated directly.) For convenience, we adopt an approach similar to [BLT86] and use *signs* on tuples: $+$ to denote an inserted or an existing tuple, and $-$ to denote a deleted tuple. Tuple signs are propagated through relational operations, as we will illustrate.

For view $V = \Pi_W(r_1 \bowtie r_2)$, consider an update $U_1 = delete(r_1, [1,2])$, which causes the warehouse to issue a query $Q_1 = \Pi_W([1,2] \bowtie r_2)$. Using signs, we instead issue the query $Q_1 = \Pi_W(-[1,2] \bowtie r_2)$, where "$-$" attached to tuple [1,2] represents that this is a deleted tuple. Suppose that at the source there is an r_2 tuple [2,3] (which by default has a $+$ sign). The result of Q_1, which we call A_1, will contain the tuple $-[1]$, i.e., the minus sign carries through. Relation A_1 is then returned to the warehouse, where it is combined with the existing view by an operation (explained below): $MV \leftarrow MV + A_1$. Because of the minus sign, the [1] tuple in A_1 is removed from the view. Note that if the original update U_1 had been an insert, the tuple [1,2] would have a plus sign, and tuple [1] would instead have been added to the view. Using tuple signs allows us to handle inserts and deletes uniformly and compactly in our algorithms.

More formally, existing tuples and inserted tuples always have a plus sign, while deleted tuples always have a minus sign. When a relational algebra expression operates on signed tuples, the signs of the result tuples are given by the following tables, where t, t_1 and t_2 are signed tuples:

t	$\sigma_{cond}(t)$	$\Pi_{proj}(t)$
$+$	$+$	$+$
$-$	$-$	$-$

t_1	t_2	$t_1 \times t_2$
$+$	$+$	$+$
$+$	$-$	$-$
$-$	$-$	$+$
$-$	$+$	$-$

Table 3.1: Signs of resulting tuples after relational algebra operations

In addition, we define two binary operators, also called $+$ and $-$, which operate on relations with signed tuples. For a relation r, let $pos(r)$ denote the tuples in r with a plus sign and let $neg(r)$ denote the tuples with a minus sign. Then:

$$r_1 + r_2 = (pos(r_1) \cup pos(r_2)) - (neg(r_1) \cup neg(r_2))$$
$$r_1 - r_2 = r_1 + (-r_2)$$

Operators $+$ and $-$ are commutative and associative, and they generalize to relational expressions and to single tuples in the obvious way. The cross product $\times$ is distributive over $+$ and $-$.

Note that the use of signed tuples is a notational convenience only—it is not necessary for sources to handle signed tuples in order to participate in our algorithms.

3.2.2 Query Expressions

In maintaining a view over relations $r_1, \ldots, r_n$, our algorithms will generate queries that contain a collection of terms. Each term is of the form:

$$T = \Pi_{proj}(\sigma_{cond}(\tilde{r}_1 \times \tilde{r}_2 \times \ldots \times \tilde{r}_n)) \tag{3.1}$$

where each $\tilde{r}_i$ is either a relation r_i or an updated tuple t_i of r_i. A query is formed by a sum of terms:

$$Q = \sum_i T_i. \tag{3.2}$$

As an example, the following relational algebra expression is a query we might generate:

$$\begin{aligned} Q \;&=\; \Pi_{proj}(\sigma_{cond}(r_1 \times [2,3] \times r_3)) \\ &\quad + \Pi_{proj}(\sigma_{cond}(-[1,2] \times [2,3] \times r_3)) \end{aligned}$$

In our algorithms we often derive queries from earlier queries or from view definitions. For example, say we are given a view definition $V = \Pi_{proj}(\sigma_{cond}(r_1 \times r_2))$, and we receive a deletion $U = delete(r_2, [3,4])$. Then the query we want to send to the source is V with r_2 substituted by $-[3,4]$, i.e., $Q = \Pi_{proj}(\sigma_{cond}(r_1 \times -[3,4]))$. We use $V\langle U\rangle$ to denote view expression V with the updated tuple of U substituted for U's relation.

More formally, consider any query (or view definition) of the form $Q = \sum_i T_i$ (recall Equation 3.2). Let U be an update involving relation r_k, and let $tuple(U)$ be the updated tuple. Then $Q\langle U\rangle = \sum_i T_i\langle U\rangle$, where for each T_i (recall Equation 3.1):

$$T_i\langle U\rangle = \begin{cases} \phi & \textit{if } \tilde{r}_k \textit{ is an updated tuple} \\ \Pi_{proj}(\sigma_{cond}(\tilde{r}_1 \times \ldots \times \tilde{r}_{k-1} \times tuple(U) & \\ \quad \times \tilde{r}_{k+1} \times \ldots \times \tilde{r}_n)) & \textit{if } \tilde{r}_k \textit{ is relation } r_k \end{cases}$$

We also recursively define $Q\langle U_1, U_2 \ldots U_k\rangle$ to be $(Q\langle U_1, U_2, \ldots U_{k-1}\rangle)\langle U_k\rangle$. That is, $Q\langle U_1, U_2 \ldots, U_k\rangle$ is the query in which all updated relations have been replaced by the corresponding updated tuples. Note that, by definition, if any two or more of the updates $U_1, U_2, \ldots U_k$ occur on the same relation, then $Q\langle U_1, U_2, \ldots U_k\rangle = \phi$.

3.3 The Eager Compensating Algorithm

In this section we present the details of our *Eager Compensating Algorithm* (ECA), introduced in Section 1.3 as a solution to the anomaly problem. ECA is an incremental view maintenance algorithm based on the centralized view maintenance algorithm described in [BLT86]. ECA anticipates the anomalies that arise due to the decoupling between base relation updates and view modification, and ECA compensates for the anomalies as needed to ensure correct view maintenance. Before we present ECA and its extensions, we first review the original incremental view maintenance algorithm.

Algorithm 3.3.1 (Conventional Algorithm)

At the source:

- S_up_i: execute U_i;
  send U_i to the warehouse;
  trigger event W_up_i at the warehouse
- S_qu_i: receive query Q_i;
  let $A_i = Q_i[ss_i]$; (ss_i is the current source state)
  send A_i to the warehouse;
  trigger event W_ans_i at the warehouse

At the warehouse:

- W_up_i: receive update U_i;
  let $Q_i = V\langle U_i\rangle$;
  send Q_i to the source;
  trigger event S_qu_i at the source
- W_ans_i: receive A_i;
  update view: $MV = MV + A_i$

End Algorithm

3.3.1 The Conventional Algorithm

The view maintenance algorithm described in [BLT86] applies incremental changes to a view each time changes are made to relevant base relations. We adapt this algorithm to the warehousing environment and use our event-based notation, and obtain Algorithm 3.3.1.

Notice that each update at the source triggers an *S_up* event, which then triggers a *W_up* event at the warehouse, triggering an *S_qu* event back at the source, and finally a *W_ans* event at the warehouse (recall Figure 3.1). As shown in Examples 1.2.2 and 1.2.3, this algorithm may lead to anomalies. Consequently, using our definitions from Section 3.1, this conventional algorithm is neither convergent nor weakly consistent in a warehousing environment.

3.3.2 The Eager Compensating Algorithm (ECA)

We start by defining the set of "pending" queries at the warehouse:

Definition: Consider the processing of an event we at the warehouse. Let the *unanswered query set* for we, $UQS(we)$, be the set of queries that were sent by the warehouse before we occurred, but whose answers have not been received before we. We shorten $UQS(we)$ to UQS when we refers to the event being processed. □

When the warehouse receives an update U_i and UQS is not empty, then U_i may cause queries Q_j in UQS to be evaluated incorrectly. The incorrectness arises because the queries Q_j are assumed to be computed before U_i, but are actually computed after U_i. Thus, all queries in UQS will "see" a source state that already reflects update U_i. (Recall our assumption that messages are processed and delivered in order, so if a query's answer has not yet been received, the query will be evaluated after U_i.) ECA takes this behavior into account: When ECA issues its query in response to update U_i, ECA incorporates one "compensating query" for each query in UQS. The compensating queries offset the effects of U_i on the results of queries in UQS.

An important and subtle consequence of using compensating queries is that the results of queries should be applied to the view only after the answer to this query and all related compensating queries have been received. If instead we updated the view after the receipt of each answer, then the view might temporarily assume an invalid state. (That is, in the terminology of Section 3.1, the algorithm would be convergent but not consistent.) To avoid invalid view states, ECA collects all intermediate answers in a temporary relation called `COLLECT`, and only updates the view when all answers to pending queries have been received (i.e. when $UQS = \phi$).

Algorithm 3.3.2 (Eager Compensating Algorithm (ECA))

The source events behave exactly as in Algorithm 3.3.1.
At the warehouse, `COLLECT` $= \phi$ initially.

- W_up_i: receive U_i;
  let $Q_i = V\langle U_i\rangle - \sum_{Q_j \in UQS} Q_j\langle U_i\rangle$
  send Q_i to the source;
  trigger event S_qu_i at the source
- W_ans_i: receive A_i;
  let `COLLECT` = `COLLECT` + A_i;
  if $UQS = \phi$
  then let $MV \leftarrow MV +$ `COLLECT`;
  let `COLLECT` $\leftarrow \phi$
  else do nothing

End Algorithm

We call this an "eager" algorithm because the warehouse is compensating even before the answer for a query has arrived. A proof showing that ECA is strongly consistent is given in Appendix A. Recalling Section 3.1, completeness requires that every source state be reflected in some view state. Clearly some source states may be "missed" by the ECA algorithm while it collects query answers. In the next section we discuss a "lazy" version of the algorithm, and in the section after that we develop a complete algorithm.

3.3.3 ECA Examples

The following examples illustrate ECA handling different update interfering scenarios in a strongly consistent fashion.

EXAMPLE 3.3.1 (ECA)

Consider source relations:

$$r_1 : \begin{array}{cc} W & X \\ \hline 1 & 2 \end{array} \qquad r_2 : \begin{array}{cc} X & Y \\ \hline - & - \end{array} \qquad r_3 : \begin{array}{cc} Y & Z \\ \hline - & - \end{array}$$

We assume that three insertions occur at the source before any queries are answered. The base relations after the insertions are:

$$r_1 : \begin{array}{cc} W & X \\ \hline 1 & 2 \\ 4 & 2 \end{array} \qquad r_2 : \begin{array}{cc} X & Y \\ \hline 2 & 5 \end{array} \qquad r_3 : \begin{array}{cc} Y & Z \\ \hline 5 & 3 \end{array}$$

Let the view definition be $V = \Pi_W(r_1 \bowtie r_2 \bowtie r_3)$, and suppose the following events occur. Initially, the materialized view at the warehouse is empty, and `COLLECT` is initialized to empty. For brevity, we omit the source events, only listing events occurring at the warehouse.

1. Warehouse receives $U_1 = insert(r_1, [4,2])$. It sends $Q_1 = V\langle U_1 \rangle = \Pi_W([4,2] \bowtie r_2 \bowtie r_3)$ to the source.

2. Warehouse receives $U_2 = insert(r_3, [5,3])$. Currently, $UQS = \{Q_1\}$. Warehouse sends $Q_2 = V\langle U_2 \rangle - Q_1\langle U_2 \rangle = \Pi_W(r_1 \bowtie r_2 \bowtie [5,3]) - \Pi_W([4,2] \bowtie r_2 \bowtie [5,3])$.

3. Warehouse receives $U_3 = insert(r_2, [2,5])$. Since $UQS = \{Q_1, Q_2\}$, the warehouse sends

$$\begin{aligned} Q_3 &= V\langle U_3 \rangle - Q_1\langle U_3 \rangle - Q_2\langle U_3 \rangle \\ &= \Pi_W(r_1 \bowtie [2,5] \bowtie r_3) - \Pi_W([4,2] \bowtie [2,5] \bowtie r_3) \\ &\quad -(\Pi_W(r_1 \bowtie [2,5] \bowtie [5,3]) - \Pi_W[4,2] \bowtie [2,5] \bowtie [5,3])). \end{aligned}$$

4. Warehouse receives $A_1 = [4]$ and updates `COLLECT` $= \phi + ([4]) = ([4])$. Now $UQS = \{Q_2, Q_3\}$.

5. Warehouse receives $A_2 = [1]$ and updates `COLLECT` $= ([4]) + ([1]) = ([1],[4])$. Now $UQS = \{Q_3\}$.

6. Warehouse receives $A_3 = \phi$ and updates `COLLECT` $= ([1],[4]) + \phi = ([1],[4])$. Since $UQS = \phi$, the warehouse updates $MV = \phi +$ `COLLECT` $= ([1],[4])$, which is correct.

Observe that if we had installed COLLECT into MV in step 4, without waiting for $UQS = \phi$, then the view would have temporarily assumed an invalid state (resulting in convergence but not consistency). □

EXAMPLE 3.3.2 (ECA applies to a different update overlapping scenario) In this example we consider the same base relations, the same view and the same source updates as in Example 3.3.1. However, in Example 3.3.1 update U_3 occurs before query Q_1 is evaluated at the source; in this example, U_3 occurs after query Q_1 is evaluated. We show that ECA also correctly maintains the view under this update scenario. The following are the events occurring at the warehousing while processing the three updates.

1. Warehouse receives $U_1 = insert(r_1, [4,2])$. It sends $Q_1 = V\langle U_1\rangle = \Pi_W([4,2] \bowtie r_2 \bowtie r_3)$ to the source.
2. Warehouse receives $U_2 = insert(r_3, [5,3])$. Now $UQS = \{Q_1\}$, and the warehouse sends $Q_2 = V\langle U_2\rangle - Q_1\langle U_2\rangle = \Pi_W(r_1 \bowtie r_2 \bowtie [5,3]) - \Pi_W([4,2] \bowtie r_2 \bowtie [5,3])$.
3. Warehouse receives $A_1 = \phi$ and lets COLLECT $= \phi$. Now $UQS = \{Q_2\}$.
4. Warehouse receives $U_3 = insert(r_2, [2,5])$. Since $UQS = \{Q_2\}$, the warehouse sends

$$\begin{aligned} Q_3 &= V\langle U_3\rangle - Q_2\langle U_3\rangle \\ &= \Pi_W(r_1 \bowtie [2,5] \bowtie r_3) - \Pi_W(r_1 \bowtie [2,5] \bowtie [5,3]) + \Pi_W([4,2] \bowtie [2,5] \bowtie [5,3]). \end{aligned}$$

5. Warehouse receives $A_2 = [1]$, COLLECT $= ([1])$, and now $UQS = \{Q_3\}$.
6. Warehouse receives $A_3 = [4]$, COLLECT $= ([1],[4])$. Since $UQS = \phi$, the view is updated to be: $MV = \phi +$ COLLECT $= ([1],[4])$, which is correct. □

EXAMPLE 3.3.3 (ECA processing both insertions and deletions) We again consider the same base relations and the the same view as in Example 3.3.1, but the source updates are one deletion and two insertions.

The source relations before the updates are:

r_1 :

W	X
1	2

r_2 :

X	Y
-	-

r_3 :

Y	Z
-	-

The source relations after the updates are:

r_1 :

W	X
-	-

r_2 :

X	Y
2	3

r_3 :

Y	Z
3	4

The view maintenance will lead to an anomaly when using the conventional algorithm. Now we show the warehouse events occurring using ECA:

1. Warehouse receives $U_1 = delete(r_1, [1,2])$. It sends $Q_1 = V\langle U_1\rangle = -\Pi_W([1,2] \bowtie r_2 \bowtie r_3)$ to the source.

2. Warehouse receives $U_2 = insert(r_2, [2,3])$. Currently, $UQS = \{Q_1\}$. The warehouse sends $Q_2 = V\langle U_2\rangle - Q_1\langle U_2\rangle = \Pi_W(r_1 \bowtie [2,3] \bowtie r_3) + \Pi_W([1,2] \bowtie [2,3] \bowtie r_3)$.

3. Warehouse receives $U_3 = insert(r_3, [3,4])$. Since $UQS = \{Q_1, Q_2\}$, the warehouse sends

$$\begin{aligned} Q_3 &= V\langle U_3\rangle - Q_1\langle U_3\rangle - Q_2\langle U_3\rangle \\ &= \Pi_W(r_1 \bowtie r_2 \bowtie [3,4]) + \Pi_W([1,2] \bowtie r_2 \bowtie [3,4]) \\ &\quad -\Pi_W(r_1 \bowtie [2,3] \bowtie [3,4]) - \Pi_W([1,2] \bowtie [2,3] \bowtie [3,4]). \end{aligned}$$

4. Warehouse receives $A_1 = -[1]$ and updates COLLECT $= \phi$ + (-[1]) = (-[1]). Currently, $UQS = \{Q_2, Q_3\}$.

5. Warehouse receives $A_2 = [1]$ (notice that $r_1 = \phi$ at the source). It updates COLLECT = (-[1]) + ([1]) = ϕ. Currently $UQS = \{Q_3\}$.

6. Warehouse receives $A_3 = \phi$, it updates COLLECT $= \phi$. Since $UQS = \phi$, the warehouse updates $MV = \phi +$ COLLECT $= \phi$, which is correct. □

We want to mention here that in all three examples we have somewhat "heavy interleaving" for source updates: the processing of an update is often interfered by subsequent updates, interfering updates affect different base relations used in the same view, and the updated tuples in the interfering updates often join with each other. We use these scenarios to illustrate how the algorithm performs. In realty, we expect that heavy interleaving occurs rarely. When source updates are spaced out and do not interleave, or when the base relations of interleaving updates are not used in the same view, ECA behaves just like the conventional algorithm. When the tuples of interfering updates do not join with each other, the corresponding compensating queries will have empty results and need not to be evaluated at the source. In Chapter 6 we study performance under different source update interleaving scenarios.

3.3.4 The ECA-Key Algorithm

We can "streamline" ECA in the case where the attributes in the projection list of the view definition (recall Section 3.2) contain key attributes for each of the base tables. (In fact, it could be advisable to design warehouse view definitions with this property in mind, for more efficient maintenance.) The *ECA-Key Algorithm* (ECA^K) proceeds as follows:

1. `COLLECT` is initialized with the current materialized view (not the empty set). Instead of storing modifications to MV, `COLLECT` is a "working copy" of MV.

2. When a delete is received at the warehouse, no query is sent to the source. Instead, the delete is applied to `COLLECT` immediately, using a special *key-delete* operation defined below. Also, if a pending query(in UQS) is triggered by the insertion of the same tuple as the deleted tuple, then mark the answer of this query "ignore".

3. When an insert is received at the warehouse, a query is sent to the source. However, no compensating queries are added. Thus, when an insert U is received, the query sent to the source is simply $V\langle U \rangle$.

4. As answers are received, they are accumulated in the `COLLECT` set, as in the original ECA. However, those answers that are marked "ignore" are not added to `COLLECT`. Also, duplicate tuples are not added to the `COLLECT` set. When the view contains keys for all base relations, there can be no duplicates in the view. Thus, if a duplicate occurs, it is due to an anomaly and can be ignored.

5. When UQS is empty, the materialized view is updated, replacing it with the tuples in `COLLECT`.

In the next chapter we discuss more on algorithms taking advantage of key information. Here we use an example to illustrate how ECA^K solves the anomaly problem.

EXAMPLE 3.3.4 (ECA^K)

Consider the following source relations, where W and Y are key attributes.

r_1 :

W	X
1	2

r_2 :

X	Y
2	3

Suppose that the warehouse view definition is $V = \Pi_{W,Y}(r_1 \bowtie r_2)$, and the following events occur. Initially, the materialized view at the warehouse is MV = ([1,3]), and `COLLECT` = MV.

1. The warehouse received $U_1 = insert(r_2, [2,4])$ and sends $Q_1 = V\langle U_1 \rangle = \Pi_{W,Y}(r_1 \bowtie [2,4])$.

2. The warehouse receives $U_2 = insert(r_1, [3,2])$ and sends $Q_2 = V\langle U_2 \rangle = \Pi_{W,Y}([3,2] \bowtie r_2)$.

3. The warehouse received $U_3 = delete(r_1, [1,2])$. Operation *key-delete*(V, r_1, [1,2]) (see below) deletes tuples of the form [1,x], obtaining `COLLECT` = `COLLECT` − ([1,3]) = ϕ, $UQS = \{Q_1, Q_2\}$.

4. The warehouse receives $A_1 = ([3,4])$. It updates `COLLECT` = `COLLECT` + ([3,4]) = ([3,4]). Currently $UQS = \{Q_2\}$.

5. The warehouse receives $A_2 = ([3,3],[3,4])$. First, A_2 is added to `COLLECT`: duplicate tuple [3,4] is not added, so `COLLECT` = ([3,3],[3,4]). Next, since $UQS = \phi$, MV is set to `COLLECT`, so MV = ([3,3],[3,4]). Note that `COLLECT` is not reset to empty. □

The special operation *key-delete*(MV, r_1, [1,2]) deletes from MV all tuples whose attribute corresponding to r_1's key (i.e., attribute W) is equal to the key value in tuple [1,2] (i.e., 1).

It is the presence of keys in the view definition that makes it possible to perform deletes at the warehouse without issuing queries to the source, and that eliminates the need for compensating queries in the case of inserts. Consider deletions first. Since each view tuple contains key values for all base relations, when a base relation tuple t is deleted, we can use the key values in t to identify which tuples in the view were derived using t. These are the view tuples that must be deleted.

Now consider insertions. Since insertions cause queries to be sent to the source, anomalies can still arise. However, all such anomalies result in either duplicate view tuples (which we can detect and ignore), or missing tuples that would have been deleted anyway. As illustration, suppose Q_1 of Example 3.3.4 had been executed when U_1 occurred; it would have evaluated to ([1,4]). Instead, because of the delay, we have A_1 = ([3,4]). A_1 is incorrect in two ways: (1) It contains an extra tuple [3,4] produced because Q_1 was evaluated after insert U_2. (2) It is missing tuple [1,4] because Q_1 was evaluated after delete U_3. However, both of these problems are resolved at the warehouse: (1) The extra tuple [3,4] is identified as a duplicate when A_2 is received. (2) The missing tuple [1,4] would have been deleted by U_3. Therefore, ECA^K is strongly consistent. A correctness proof for ECA^K appears in Appendix B.

3.3.5 The ECA-Local Algorithm

The original ECA uses compensating queries to avoid the anomalies that may occur when queries are sent to the source. ECA^K relies on key properties to avoid compensating queries; furthermore, ECA^K introduces the concept of *local updates* (deletions, in the case of ECA^K), which can be handled at the warehouse without sending queries to the source. Now we discuss the idea of another algorithm, the *ECA-Local Algorithm* (ECA^L), that combines the compensating queries of ECA with the local updates of ECA^K to produce a streamlined algorithm that applies to general views.

In ECA^L, each update is handled either locally or non-locally. A number of papers, e.g., [BLT86, GB95, TB88], describe conditions when, for a particular view definition and a particular base relation update, the view can be updated without further access to base relations (i.e., the view is self-maintainable with respect to one or all source updates). These results can be used to identify which updates ECA^L will process locally. For updates that cannot be processed locally, ECA is used (assuming the key condition for ECA^K does not hold), with compensating queries whenever necessary.

Unfortunately, maintaining the correct order of execution among local and non-local processing in ECA^L is not straightforward. Intuitively, we can process a local update as soon as all queries for previous updates have been answered and applied to the view. However, consider three updates, U_1, U_2, and U_3, where U_2 is the only local update. Suppose we process U_2 as soon as the query Q_1 for U_1 is answered. Since ECA^L uses compensating queries, the "true" view update corresponding to U_1 may include not only the answer for Q_1, but also compensations appearing in queries for later updates (such as U_3). To correctly handle this scenario, ECA^L must buffer updates and, in some cases, "split" query results (separating compensation from original), in order to process local updates on a correct version of the view. Same as original queries, some compensating queries may also be evaluated locally. We will not present the details of ECA^L in this dissertation.

3.4 The Lazy Compensating Algorithm

In this section we present an alternative to ECA, the *Lazy Compensating Algorithm* (LCA). LCA also sends compensating queries to the source, to correct any potential view maintenance anomalies. However, while ECA compensates before the answer for a query is received, LCA sends compensating queries after it receives the answer to a query and after it finds out that the answer may be incorrect due to concurrent source updates.

LCA is not only a different approach to maintaining warehouse views, but, more importantly, it serves as the basis for the Complete Compensating Algorithm (CCA), which we discuss in the next section. (Remember that ECA is strongly consistent, and as we will see, so is LCA.)

3.4.1 The Lazy Compensating Algorithm (LCA)

The basic idea of LCA is the following: When the warehouse sends a query Q to the source, it expects that Q be evaluated at source state ss_i, i.e., $Q[ss_i]$ is the intended answer for Q. When the actual answer A is received, the warehouse may detect that the query was evaluated at a later source state ss_j, i.e., $A = Q[ss_j], j > i$. Then to obtain $Q[ss_i]$ which is the correct answer, the warehouse sends a *compensating query*

CQ, such that the answer of CQ together with A will be the correct answer to Q, i.e., $Q[ss_i] = A + CQ[ss_j]$. Of course, when evaluating CQ the source state may be advanced again, so that CQ itself may be compensated recursively.

We define a notation $Q\langle S\rangle$, which will be used to construct the compensating query CQ in LCA.

Definition: For a set of updates $S_n = \{U_1, U_2, \ldots U_n\}$, define $Q\langle S\rangle = Q\langle S_{n-1}\rangle + Q\langle U_n\rangle - Q\langle S_{n-1}\rangle\langle U_n\rangle$. □

EXAMPLE 3.4.1 ($Q\langle S\rangle$)

$$\begin{aligned}
Q\langle S_1\rangle &= Q\langle U_1\rangle; \\
Q\langle S_2\rangle &= Q\langle U_1\rangle + Q\langle U_2\rangle - Q\langle U_1, U_2\rangle; \\
Q\langle S_3\rangle &= Q\langle U_1\rangle + Q\langle U_2\rangle - Q\langle U_1, U_2\rangle + Q\langle U_3\rangle - (Q\langle U_1\rangle + Q\langle U_2\rangle - Q\langle U_1, U_2\rangle)\langle U_3\rangle \\
&= Q\langle U_1\rangle + Q\langle U_2\rangle + Q\langle U_3\rangle - Q\langle U_1, U_2\rangle - Q\langle U_1, U_3\rangle - Q\langle U_2, U_3\rangle \\
&\quad + Q\langle U_1, U_2, U_3\rangle
\end{aligned}$$

□

We also introduce a set IUS, which contains the set of source updates that may interfere with the evaluation of a query:

Definition: Consider a query answer $A_{i,j}$ the warehouse received. Let the *interfering update set* for $A_{i,j}$, $IUS(A_{i,j})$, be the set of updates such that each update U in this set satisfies the conditions that (1) U was received by the warehouse after query $Q_{i,j}$ is sent, but before the answer $A_{i,j}$ is received; (2) $rel(U)$ is used in $Q_{i,j}$. □

Updates in $IUS(A_{i,j})$ are those that actually affect the evaluation of $Q_{i,j}$, and thus those updates need to be compensated. (If we think in the context of ECA, an update in $IUS(A_{i,j})$ is one that is received when $Q_{i,j}$ is in UQS.) If $IUS(A_{i,j})$ is empty, then the answer to $Q_{i,j}$ is correct.

Say a query Q was intended to be evaluated at ss_i but was actually evaluated at ss_j. Let $A = Q[ss_j]$ and $S = IUS(A)$. An important conclusion given in Lemma C.0.7 in Appendix C is that $Q[ss_i] = A - Q\langle S\rangle[ss_j]$. Thus, $-Q\langle S\rangle$ is the compensating query LCA should send.

Now we present algorithm LCA, then give examples to show how LCA correctly maintains warehouse views. In the algorithm, `COLLECT` is a temporary relation that collects intermediate query answers.

Algorithm 3.4.1 (Lazy Compensating Algorithm (LCA))

The source events behave exactly as in Algorithm 3.3.1.
At the warehouse, $\texttt{COLLECT} = \phi$ initially.

- W_up_i: receive U_i;
  let $Q_{i,0} = V\langle U_i \rangle$;
  send $Q_{i,0}$ to the source
- $W\_ans_{i,p}$: receive $A_{i,p}$;
  let $\texttt{COLLECT} = \texttt{COLLECT} + A_{i,p}$;
  if $UQS = \phi$
  then let $MV \leftarrow MV + \texttt{COLLECT}$;
  let $\texttt{COLLECT} \leftarrow \phi$
  else let $S = IUS(A_{i,p})$;
  if $S \neq \phi$, let $Q_{i,p+1} = -Q_{i,p}\langle S \rangle$;
  send $Q_{i,p+1}$ to the source

End Algorithm

LCA is also strongly consistent. The correctness proof of LCA is given in Appendix C. LCA may be less efficient than ECA in that it sends more queries to the source, but, as we will see in the next section, LCA can be made into a complete algorithm easily. LCA is also useful when the warehouse cannot detect whether an update affects a previous query until the query answer is received, for instance, when the messages between the source and the warehouse can be out-of-order and other mechanisms (e.g., sequence numbers) are used to detect whether a query answer sees a relation in the correct state.

3.4.2 LCA Examples

The following examples show the same update scenarios as those ECA examples in Section 3.3.3, except that the anomalies are corrected by LCA.

EXAMPLE 3.4.2 (LCA)

We now consider the same base relations, view and source updates as in Example 3.3.1. Before updates, the source relations are:

$$r_1: \begin{array}{cc} W & X \\ \hline 1 & 2 \end{array} \qquad r_2: \begin{array}{cc} X & Y \\ \hline - & - \end{array} \qquad r_3: \begin{array}{cc} Y & Z \\ \hline - & - \end{array}$$

The view definition is $V = \Pi_W(r_1 \bowtie r_2 \bowtie r_3)$, three insertions occur at the source before any queries are answered. Initially, the materialized view at the warehouse is empty, and `COLLECT` is initialized to empty. We list the events occurring at the warehouse.

1. Warehouse receives $U_1 = insert(r_1, [4,2])$. It sends to the source $Q_{1,0} = V\langle U_1 \rangle = \Pi_W([4,2] \bowtie r_2 \bowtie r_3)$.

2. Warehouse receives $U_2 = insert(r_3, [5,3])$. It sends $Q_{2,0} = V\langle U_2 \rangle = \Pi_W(r_1 \bowtie r_2 \bowtie [5,3])$.

3. Warehouse receives $U_3 = insert(r_2, [2,5])$. It sends $Q_{3,0} = V\langle U_3 \rangle = \Pi_W(r_1 \bowtie [2,5] \bowtie r_3)$.

4. Warehouse receives $A_{1,0} = [4]$ and let `COLLECT` $= \phi + ([4]) = ([4])$. Since $S = \{U_2, U_3\}$ for this query, the warehouse sends compensating query $Q_{1,1} = -Q_{1,0}\langle S \rangle = -(\Pi_W([4,2] \bowtie r_2 \bowtie [5,3]) + \Pi_W([4,2] \bowtie [2,5] \bowtie r_3) - \Pi_W([4,2] \bowtie [5,3] \bowtie [2,5]))$.

5. Warehouse receives $A_{2,0} = ([1],[4])$ and let `COLLECT` $= ([4]) + ([1],[4]) = ([1],[4],[4],[4])$. Since $S = \{U_3\}$, the warehouse sends compensating query $Q_{2,1} = -Q_{2,0}\langle S \rangle = -\Pi_W(r_1 \bowtie [2,5] \bowtie [5,3])$.

6. Warehouse receives $A_{3,0} = ([1],[4])$ and let `COLLECT` $= ([1],[4],[4]) + ([1],[4]) = ([1],[4],[4],[1],[4])$. Since $S = \phi$, no compensating query is sent.

7. Warehouse receives $A_{1,1} = -([4])$ and let `COLLECT` $= ([1],[4],[4],[1],[4])-([4]) = ([1],[4],[4],[1])$.

8. Warehouse receives $A_{2,1} = -([1],[4])$ and let `COLLECT` $= ([1],[4],[4],[1])-([1],[4]) = ([1],[4])$. Since there is no pending query, the warehouse updates $MV = \phi +$ `COLLECT` $= ([1],[4])$, which is correct. □

EXAMPLE 3.4.3 (LCA applies to a different update overlapping scenario) In this example we consider the same base relations, the same view and the same source updates as in Example 3.3.2. Now update U_3 occurs after query Q_1 is evaluated. We show that LCA also correctly maintains the view under this update scenario. The following are the events occurring at the warehousing while processing the three updates.

1. Warehouse receives $U_1 = insert(r_1, [4,2])$. It sends to the source $Q_{1,0} = V\langle U_1 \rangle = \Pi_W([4,2] \bowtie r_2 \bowtie r_3)$.

2. Warehouse receives $U_2 = insert(r_3, [5,3])$. It sends $Q_{2,0} = V\langle U_2 \rangle = \Pi_W(r_1 \bowtie r_2 \bowtie [5,3])$.

3. Warehouse receives $A_{1,0} = \phi$ and COLLECT $= \phi$. Since $S = \{U_2\}$, the warehouse sends compensating query $Q_{1,1} = -Q_{1,0}\langle S \rangle = -(\Pi_W([4,2] \bowtie r_2 \bowtie [5,3])$.

4. Warehouse receives $U_3 = insert(r_2, [2,5])$. It sends $Q_{3,0} = V\langle U_3 \rangle = \Pi_W(r_1 \bowtie [2,5] \bowtie r_3)$.

5. Warehouse receives $A_{2,0} = ([1],[4])$ and let COLLECT $= \phi + ([1],[4]) = ([1],[4])$. Since $S = \{U_3\}$ for this query, the warehouse sends compensating query $Q_{2,1} = -Q_{2,0}\langle S \rangle = -\Pi_W(r_1 \bowtie [2,5] \bowtie [5,3])$.

6. Warehouse receives $A_{3,0} = ([1],[4])$ and let COLLECT = ([1],[4])+ ([1],[4]) = ([1],[4],[1],[4]). Since $S = \phi$, no compensating query is sent.

7. Warehouse receives $A_{1,1} = -([4])$ and let COLLECT = ([1],[4],[1],[4])-([4]) = ([1],[4],[1]). Since $S = \{U_3\}$ for this query, the warehouse sends compensating query $Q_{1,2} = -Q_{1,1}\langle S \rangle = \Pi_W([4,2] \bowtie [2,5] \bowtie [5,3])$.

8. Warehouse receives $A_{2,1} = -([1],[4])$ and let COLLECT = ([1],[4],[1])-([1],[4]) = ([1]).

9. Warehouse receives $A_{1,2} = ([4])$ and let COLLECT = ([1])+([4]) = ([1],[4]). Since there is no pending query, the warehouse updates $MV = \phi +$ COLLECT $= ([1],[4])$, which is correct.

Notice that in this example, query $Q_{1,0}$ was compensated twice, because the answer to the first compensating query, $A_{1,1}$, was again incorrect due to the presence of U_3. □

3.5 The Complete Compensating Algorithm

Recalling Section 3.1, completeness requires that every source update be reflected in the view in the order of occurrence. In this section we modify LCA to obtain a complete algorithm, which we call *Complete Compensating Algorithm* (CCA).

There are two separate phases to maintain a view corresponding to an update: the *computation phase*, which computes the corresponding view deltas, and the *application phase*, which applies the view deltas to the materialized view. Consequently, there are two approaches to maintain a view in a complete fashion:

- First approach: The warehouse computes the view deltas corresponding to different source updates concurrently (as we did in ECA and LCA). View deltas corresponding to different updates are collected into different temporary relations and applied to the materialized view in the order of the updates.

- Second approach: The warehouse computes and applies view deltas for one update at a time. That is, all the updates the warehouse receives are placed in a queue. The warehouse processes the first update in the queue only. After the processing is completed and the corresponding view deltas applied, the warehouse starts processing the next update.

The first approach is often more efficient because the computation of view deltas corresponding to different updates can be performed concurrently. CCA takes this approach. When key information is used to correct anomalies, the order of update processing becomes more important. In Section 4.6, we will develop a complete algorithm using the second approach and key information.

Algorithm CCA sends the same queries (including compensating queries) to the source as Algorithm LCA. Remember that one update triggers a series of queries. CCA waits until it has received answers to all those queries, then applies them together. In particular, CCA collects answers to all queries related to an update U_i into a separate temporary relation `COLLECT[i]`, then decides when `COLLECT[i]` can be applied to the view. Two conditions must be satisfied before `COLLECT[i]` can be applied: First, $IUS(A_{i,j})$ must be empty so no more compensating queries need to be sent, and second, all view deltas corresponding to updates prior to U_i must already have been applied. Because it is possible that, for two updates U_i and U_j, $i < j$, all query answers related to U_j are received but not those related to U_i, CCA maintains a Boolean array $AllAnswersReceived[i]$ to remember which view deltas are ready to be applied.

Algorithm 3.5.1 (Complete Compensating Algorithm (CCA))

The warehouse update event W_up_i is the same as in Algorithm 3.4.1 (LCA).
At the warehouse, initially let `COLLECT[i]` $= \phi$.

```
• W_ans_{i,p}: receive A_{i,p};
               let COLLECT[i] = COLLECT[i] + A_{i,p};
               if IUS[Q_{i,p}] = φ
                  then call MaintainView(i);
                  else  let S = IUS[A_{i,p}];
                        let Q_{i,p+1} = -Q_{i,p}⟨S⟩;
                        send Q_{i,p+1} to the source

Procedure    MaintainView(i)
     if (i ≠ first)
               let AllAnswersReceived[i] = true; return;
     if (i == first)
               let MV ← MV + COLLECT[i];
               let first = first+1;
               while (first < last)
                  if (AllAnswersReceived[first])
                     let MV ← MV + COLLECT[i];
                     let first = first +1;
                  else return
End Procedure
End Algorithm
```

At any given time, CCA maintains two indices, *first* and *last*, indicating the first and last update received but whose view deltas have not been applied. Initially, when the first update is received by CCA, it is labeled as U_1, and CCA sets both *first* and *last* to 1. Whenever a new update is received by CCA, *last* is advanced by 1 and the new update is labeled as U_{last}. Boolean flag $AllAnswersReceived[i]$ is initially set to false for all i's.

CCA can update the view incrementally corresponding to each base database state. The correctness proof of CCA is given in Appendix C.

The following is an example of how CCA can maintain the view correctly in the scenario given in Example 3.3.1.

EXAMPLE 3.5.1 (CCA)

We now consider the same base relations, view and source updates as in Example 3.3.1. Initially, the materialized view at the warehouse is empty, and `COLLECT[i]`s are initialized to empty. We list the events occurring at the warehouse using CCA.

1. Warehouse receives $U_1 = insert(r_1, [4,2])$. It sends to the source $Q_{1,0} = V\langle U_1\rangle = \Pi_W([4,2] \bowtie r_2 \bowtie r_3)$.

2. Warehouse receives $U_2 = insert(r_3, [5,3])$. It sends $Q_{2,0} = V\langle U_2\rangle = \Pi_W(r_1 \bowtie r_2 \bowtie [5,3])$.

3. Warehouse receives $U_3 = insert(r_2, [2,5])$. It sends $Q_{3,0} = V\langle U_3\rangle = \Pi_W(r_1 \bowtie [2,5] \bowtie r_3)$.

4. Warehouse receives $A_{1,0} = [4]$ and let `COLLECT[1]` $= \phi + ([4]) = ([4])$. Since $S = \{U_2, U_3\}$ for this query, the warehouse sends compensating query $Q_{1,1} = -Q_{1,0}\langle S\rangle = -(\Pi_W([4,2] \bowtie r_2 \bowtie [5,3]) + \Pi_W([4,2] \bowtie [2,5] \bowtie r_3) - \Pi_W([4,2] \bowtie [5,3] \bowtie [2,5]))$.

5. Warehouse receives $A_{2,0} = ([1],[4])$ and let `COLLECT[2]` $=$ ([1],[4]). Since $S = \{U_3\}$, it sends compensating query $Q_{2,1} = -Q_{2,0}\langle S\rangle = -\Pi_W(r_1 \bowtie [2,5] \bowtie [5,3])$.

6. Warehouse receives $A_{3,0} = ([1],[4])$ and lets `COLLECT[3]` $=$ ([1],[4]). Since $IUS[A_{3,0}] = \phi$, no compensating query is sent and `COLLECT[3]` is ready to be applied. However, since $first = 1$, $AllAnswersReceived[3]$ is set to true and the view delta is not applied.

7. Warehouse receives $A_{1,1} = -([4])$ and let `COLLECT[1]` $=$ ([4])-([4]) $= \phi$. Since $IUS[A_{1,1}] = \phi$ and $first = 1$, `COLLECT[1]` is ready to be applied. The warehouse updates $MV = \phi +$ `COLLECT[1]` $= \phi$.

8. Warehouse receives $A_{2,1} = -([1],[4])$ and let COLLECT[2] = ([1],[4])-([1],[4]) = ϕ. Since $IUS[A_{1,1}] = \phi$ and $first = 2$, COLLECT[2] is ready to be applied. The warehouse updates $MV = \phi +$ COLLECT[2] $= \phi$.

   Since $AllAnswersReceived[3]$ is true, the warehouse let $MV = \phi +$ COLLECT[3] = ([1].[4]).

For this example, the warehouse updated MV three times, to ϕ, ϕ, and $([1],[4])$. These are the correct views corresponding to U_1, U_2 and U_3 respectively. □

3.6 Performance Evaluation

In Chapter 1 we outlined several strategies for view maintenance in a warehousing environment: recomputing the view (RV), storing copies of all base relations (SC), our compensating algorithms ECA, LCA, and ECA^K. All of these approaches provide strong consistency (recall Section 3.1). Thus, a natural issue to explore is the relative performance of these strategies. (CCA is complete. We will study performance comparison between complete and strongly consistent algorithms in Chapter 6.) In this section we analyze the performance of RV and ECA. From the performance perspective, ECA^K is an enhancement to ECA that eliminates querying the source in certain cases. Since there is very little additional overhead in ECA^K, ECA^K should certainly be used when it is possible to arrange for warehouse views to include all base relation keys. Storing copies of base relations (SC) can be seen as an enhancement to any of our algorithms, requiring an "orthogonal" performance comparison (based on warehouse storage costs, etc.) that is beyond the scope of this analysis. As mentioned in Section 3.4, LCA is in general less efficient than ECA, thus it should also be used only when ECA does not apply.

When we intuitively compare RV and ECA, it seems that ECA should certainly outperform RV, since ECA is an incremental update algorithm while RV recomputes the view from scratch. However, ECA may need to send many more queries to the source than RV. In addition, ECA's queries grow in complexity as compensations upon compensations are added. Hence, we seek to determine when it is more effective to recompute the entire view, rather than maintaining it incrementally with the associated extra activity. We note that this is not a comprehensive analysis, as there are a number of parameters we have not fully studied (ranging from the number of relations, to the exact sequence and timing of the updates, to the way queries are optimized at the source). Rather, we have selected a "representative" scenario that serves to illustrate the performance tradeoffs. In Chapter 6, we provide more detailed performance results comparing different incremental view maintenance algorithms developed in this and the next chapter.

For the analysis in this section, we focus on three separate cost factors: M, the number of messages sent between the source and warehouse, B, the number of bytes transferred from the source to the warehouse, and IO, the number of I/O's performed at the source. In RV and ECA, identical update notification messages are sent to the warehouse, so these costs are not included in our calculations. Throughout this section, we use the variables listed in Table 3.2, shown with their default values. Note that our results are for relatively small base relations. The reason that we choose these default values is because in this situation RV and ECA are comparable and there are tradeoffs to investigate. When the sizes of base relations are large, as expected in real data warehouses, ECA almost always outperforms RV, as we will show in the following. The RV algorithm is described informally in Section 1.3; a formal specification is provided in [ZGMHW94].

	Variable Description	**Default**
M	Number of messages sent	N/A
B	Total number of bytes transferred	N/A
IO	Number of I/O's	N/A
C	Cardinality of a relation	100
S	Size of projected attributes	4 bytes
σ	Selection factor	1/2
J	Join factor	4
k	# of updates at the source	N/A

Table 3.2: List of variables used by performance analysis of RV and ECA

3.6.1 Performance Based on Number of Messages

Assume there is a sequence of k updates. For RV, assume the warehouse sends a query message to the source asking it to recompute the view after every s updates, $s \leq k$. Counting both the query and answer messages, the total number of messages is $M_{RV} = \lceil \frac{k}{s} \rceil \times 2$. Thus, RV generates at least 2 messages (if the view is only recomputed once; $s = k$) and at most $2k$ messages (if the view is recomputed after every update; $s = 1$). For ECA, if there are k updates, we always have k queries and k answers, so there are $2k$ messages.[1]

Thus, in the situation least favorable for ECA, ECA sends $2k$ messages while RV sends 2 messages. Of course, the price of the RV approach in this case is that the state of the warehouse view lags well behind the state of the base relations. In the most favorable situation for ECA, ECA and RV both send $2k$ messages.

3.6.2 Performance Based on Data Transferred

To analyze the number of bytes transferred (and later on the number of I/O's), we introduce a sample scenario consisting of a particular view and a particular sequence of update operations. As mentioned earlier, we have chosen to focus on a sample scenario to illustrate the performance tradeoffs while keeping the number of parameters manageable.

EXAMPLE 3.6.1 (Example warehouse scenario)

Base relation schema: $r_1(W, X)$, $r_2(X, Y)$, $r_3(Y, Z)$

View definition: $V = \Pi_{W,Z}(\sigma_{cond}(r_1 \bowtie r_2 \bowtie r_3))$

Updates: $U_1 = insert[r_1, t_1]$, $U_2 = insert[r_2, t_2]$, $U_3 = insert[r_3, t_3]$ □

The condition *cond* involves a comparison between attributes W and Z (e.g., $W > Z$). (This condition has an impact on the derivation of the the number of I/O's performed.) Later we extend this example to a sequence of k updates.

[1]Because ECA uses signed queries (recall Section 3.2.1), and some sources—such as an SQL server—may need to handle the positive and negative parts of such queries separately, we may need to send a pair of queries for some updates instead of a single query. We assume the pair of queries is "packaged" as one message, and the pair of answers also is returned in a single message.

We make the following assumptions in our analysis:

1. The cardinality (number of tuples) of each relation is some constant C.
2. The size of the combined W, Z attributes is S bytes.
3. The join factor $J(r_i, a)$ is the expected number of tuples in r_i that have a particular value for attribute a. We assume that the join factor is a constant J in all cases. For example, if we join a 20-tuple relation with a second relation, we expect to get $20J$ tuples.
4. The selectivity for the condition *cond* is given by σ, $0 \leq \sigma \leq 1$.
5. We assume that C, J and our other parameters do not change as updates occur. This closely approximates their behavior in practice when the updates are single-tuple inserts and deletes (so the size and selectivity do not change significantly), or when C and J are so large that the effect of updates is insignificant.

Not surprisingly, for RV the fewest bytes are transferred (B_{RVBest}) when the view is recomputed only once, after U_3 has occurred. The worst case ($B_{RVWorst}$) is when the view is recomputed after each update. For ECA, the best case ($B_{ECABest}$) is when no compensating queries are needed, i.e., the updates are sufficiently spaced so that each query is processed before the next update occurs at the source. Note that in this case, ECA performs as efficiently as the conventional incremental algorithm (Algorithm 3.3.1). The worst case for ECA ($B_{ECAWorst}$) is when all updates occur before the first query arrives at the source. Intuitively, the difference between the best and worst cases of ECA represents the "compensation cost."

The calculations for analyzing our algorithms can be found in [ZGMHW94]. In particular, the expressions derived in [ZGMHW94] for the number of bytes transferred are:

$$
\begin{aligned}
B_{RVBest} &= S\sigma CJ^2 \\
B_{RVWorst} &= 3S\sigma CJ^2 \\
B_{ECABest} &= 3S\sigma J^2 \\
B_{ECAWorst} &= 3S\sigma J(J+1)
\end{aligned}
$$

Figure 3.2 shows the number B of bytes transferred as a function of the cardinality C. (In all of our graphs, parameters have the default values of Table 3.2 unless otherwise indicated.) Best and worst cases are shown for both algorithms. Thus, for each algorithm, actual performance will be somewhere in between the best and worst case curves, depending on the timing of update arrivals (for ECA) and the frequency of view recomputation (for RV).

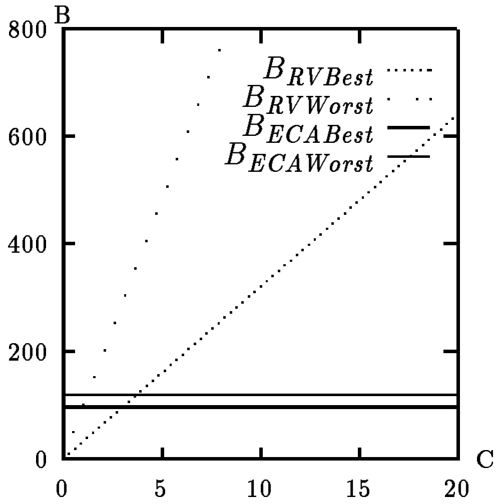


Figure 3.2: Number B of bytes transferred as a function of the cardinality C

From Figure 3.2 we see that in spite of the compensating queries, ECA is much more efficient than RV (in terms of data transferred), unless the relations involved are extremely small (less than approximately 5 tuples each). This result continues to hold over wide ranges of the join selectivity J, except if J is very small (see equations above).

One of the reasons ECA appears to perform so well is that we are considering only three updates, so the amount of "compensation work" is limited. In [ZGMHW94] we extend our analysis to an arbitrary number k of updates and obtain the following equations:

$$\begin{aligned}
B_{RVBest} &= S\sigma C J^2 \\
B_{RVWorst} &= kS\sigma C J^2 \\
B_{ECABest} &= kS\sigma J^2 \\
B_{ECAWorst} &= kS\sigma J^2 + k(k-1)S\sigma J/3
\end{aligned}$$

Figure 3.3 shows the number of bytes transferred as a function of k when $C = 100$. As expected, there is a crossover point beyond which recomputing the view once (RV's best case) is superior to even the best case for ECA. For our example, this crossover is at 100 updates. In the ECA worst case, when all updates occur at the source before any of the warehouse queries arrive, each warehouse query must compensate every preceding update. This behavior results in ECA transmitting additional data that is quadratic on the number of updates. Hence, in the situation least favorable for ECA, RV outperforms ECA when 30 or more updates are involved. Bear in mind that this situation occurs only if all updates precede all queries. If updates and queries are interleaved at the source, then performance will be somewhere between the ECA best and worst cases, and the crossover point will be somewhere between 30 and 100 updates.

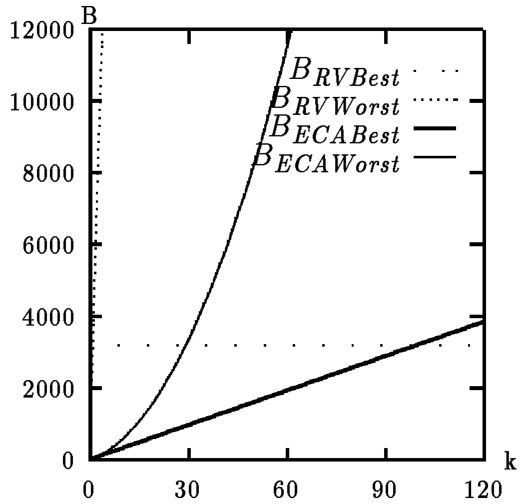


Figure 3.3: Number of bytes transferred as a function of the total interfering updates

Also notice that Figure 3.3 is for relatively small relations ($C = 100$); for larger cardinalities the crossover points will be at larger number of updates. Finally, note that the RV best case we have been comparing against assumes the view is recomputed once, no matter how many updates occur. If RV recomputes the view more frequently (such as once per some number of updates), then its cost will be substantially higher. In particular, $B_{RVWorst}$ is very expensive and always substantially worst than $B_{ECAWorst}$.

3.6.3 Performance Based on I/O

Estimating the number of I/O's performed at the source is similar to estimating the number of bytes transferred. Details of the estimation are discussed in [ZGMHW94]. We consider two extreme scenarios there: when indexing is used and ample memory is available, and when memory is very limited and there are no indexes. Studying these extremes lets us discover the conditions that are most favorable for the algorithms we consider. Here we present one graph to illustrate the type of results obtained. Figure 3.4 gives the number of I/O's as a function of the number of updates for the second scenario studied (limited memory and no indexes).

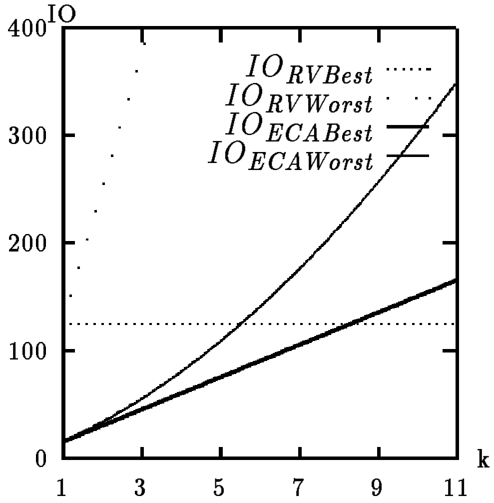


Figure 3.4: Source IO versus k, with limited memory and no indexes

The shape of the curves in Figure 3.4 is similar to those in Figure 3.3, and thus our conclusions for I/O costs are similar to those for data transmission. The main difference is that the crossover points occur with smaller update sequences: $5 < k < 8$ in this case, as opposed to a crossover between $k = 30$ and $k = 100$ when data transfer is the metric. Intuitively, this means that ECA is not as effective at reducing I/O costs as it is at reducing data transfer. However, ECA can still reduce I/O costs over RV significantly, especially if relations are larger than the 100 tuples considered for these figures. Also, we expect that the I/O performance of ECA would improve if we incorporated multiple term optimization or caching into the analysis.

3.7 Summary

In this chapter we presented algorithms for correctly maintaining materialized views in a warehousing environment. Our ECA, LCA, and two streamlined versions of ECA, ECA^K and ECA^L, are all strongly consistent, meaning that the warehouse data always corresponds to a meaningful state of the source data. CCA is complete, meaning that all source updates are reflected in the view in the correct order. An initial performance study analyzing three different cost factors (messages, data traffic, and I/O) suggests that, except for very small relations, ECA is consistently more efficient than periodically recomputing the warehouse view from scratch.

The compensating algorithms we presented in this chapter all have the following desirable properties:

1. They are incremental, meaning that they update the warehouse based on updates to the source, rather than recomputing the complete warehouse view from scratch.

2. They do not place any additional burden on the sources (e.g., timestamps, locks).

3. When the update frequency is low, i.e., when the answer to a warehouse query comes back before the next update occurs at the source, then the compensating algorithms behave exactly like the conventional incremental view maintenance algorithm 3.3.1. (Compensating queries are used only when necessary, i.e., when the answer to a query has not been received before the next update occurs at the source.)

In the next chapter we will develop view maintenance algorithms to handle views over multiple sources and handle batched updates.

Chapter 4

View Maintenance in a Multi-source Warehouse

In this chapter we show how to provide consistency for views defined over multiple sources. Many aspects of the anomaly problem remain the same as for single-source warehouses. However, additional issues are raised because warehouse queries (both regular queries and compensating queries) must be fragmented for execution at multiple sources. While fragmenting itself does not pose a novel problem (at least in the straightforward relational case), coordinating the query results and the necessary compensations for anomaly-causing updates requires some intricate algorithms.

In this chapter we also consider how the view maintenance algorithms can be extended to handle a set of updates at once, rather than one update at a time. Since we expect that in practice many source updates will be "batched," this extension should result in a very useful performance enhancement.

The organization of this chapter is as follows. In Section 4.1 we illustrate the additional view maintenance problems in a multi-source warehouse by an example. In Section 4.2, we define three transaction scenarios and specify our assumptions about the order of messages and events in a multi-source warehouse environment. In Section 4.3 we redefine the four levels of correctness for warehouse views over multiple sources and with arbitrary source transaction scenarios. Then, in Sections 4.4 and 4.5, we develop two new algorithms based on key information, and demonstrate the levels of consistency that each algorithm achieves for the different transaction scenarios. In Section 4.6, we present a complete algorithm that handles views over data across multiple sources, and show that the algorithms will terminate. In Section 4.7 we show how the compensating algorithms developed in the previous chapter can be adapt to multi-source data warehouses. We summarize in Section 4.8 and outline optimizations to our algorithms.

4.1 Multi-source View Maintenance Anomaly

The following example illustrates some of the inconsistencies that may arise in a multi-source environment. For simplicity, we assume 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 4.1.1 (View maintenance anomaly over multiple sources) Let view V be defined as $V = r_1 \bowtie r_2 \bowtie r_3$, where r_1, r_2, r_3 are three relations residing on sources x, y and z, respectively. Initially, the relations are

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

The materialized view at the warehouse is $MV = \phi$. We consider two source updates: $U_1 = insert(r_2, [2,3])$ and $U_2 = delete(r_1, [1,2])$. Figure 4.1 shows how the update U_1 is handled in the warehouse with three data sources.

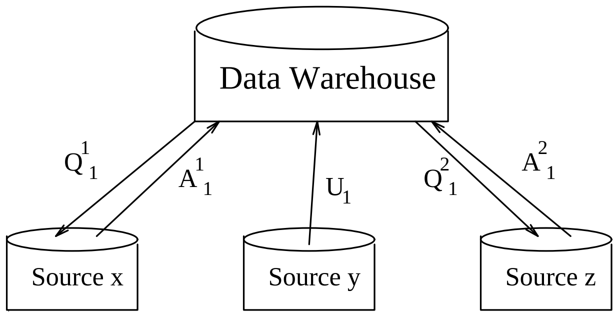


Figure 4.1: Handling source update U_1 in a three-source warehouse

Using a conventional incremental view maintenance algorithm, or using ECA-Key, 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. □

Note that the algorithms developed in the last chapter do not directly apply to handle views defined using base relations from multiple data sources. Because an update from one source may need to be integrated with data from several other sources, gathering the data corresponding to one source 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.

In Example 4.1.1, the interleaving of query Q_1 with updates arriving from the sources causes the incorrect view. 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.

The intelligent processing of updates at the warehouse also depends on how and if sources run transactions. 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. Since we do not wish to assume a particular transaction scenario, in this chapter we cover the three main possibilities: sources run no transactions, some sources run local (but not global) transactions, and some sources run global transactions.

In most of this chapter we focus on discussing algorithms for views which include the keys of all of the relations involved. We do so because we have discussed the compensating approach extensively in the last chapter. Single-source compensating algorithms can be extended to handle multi-source views straightforwardly, as we will discuss in Section 4.7, provided it is possible to detect what updates interfere with which query exactly. Key algorithms that we will develop in this chapter are more restrictive, but more efficient and easier to implement. (See Chapter 6 for performance comparison between key algorithms and compensating algorithms.) Keys also make it easier for the applications to interpret and handle the warehouse data.

The algorithms we present in this chapter provide a wide number of options for the consistency and distribution space. We map out the space of warehouse maintenance algorithms in Figure 4.3. We have implemented the multi-source algorithms in our warehouse prototype [WGL+96], demonstrating that the algorithms are practical and efficient.

4.2 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. We then address assumptions about messages and events in the multi-source environment.

4.2.1 Update Transaction Scenarios

The three transaction scenarios we consider in this chapter are:

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

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

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

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

4.2.2 Messages

As in a single-source system, 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 [Syb92]. However, batching multiple transactions into the same message does not affect the algorithms developed in this chapter. 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 4.5.2 we discuss the implications of the different schemes.

4.2.3 Event Ordering

Each source action (including a source local transaction), plus the resulting message sent to the warehouse, is considered one event. For example, evaluating a query at a source and sending the answer back to the warehouse is considered one event. We assume that events are atomic, and are ordered by the sequence of the corresponding actions. This atomic event assumption 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 reports 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 4.8.

We still 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 point-to-point message ordering assumption may also 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.

4.2.4 Discussion on Update Transaction Scenarios

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 source 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 source of an unsophisticated legacy source to report changes, or a source might choose to report updates lazily when the warehouse does not need to be kept strictly up to date.

In general, smarter sources (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 4.5.2. 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.

4.3 Correctness and Consistency

Now we generalize the correctness definition for materialized view maintenance presented in Section 3.1 for a multi-source warehousing environment, and for sources that may run transactions.

As mentioned in Chapter 2, there are two aspects of the materialized view maintenance consistency: whether at any given time the view content reflects the contents of the base data, and *how* the view reflects the changes of the base data. Correctness is defined based on source and warehouse states. The source state is the state of base data, it changes when the base data is updated; the warehouse state is the state of the warehouse view, it changes when the view is updated.

4.3.1 Source States

A *source state* is the state of all source data at a particular moment of time. Source states represent the contents of source base relations. When there are 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.

We assume that source transactions are executed is a serializable fashion across all sources, i.e., there is some serial schedule $S = T_1, T_2, \ldots T_f$ that represents execution of the transactions. (However, what constitutes a transaction varies according to the scenario.) When we define the notion of convergence in the next section, we consider executions that are finite, i.e., that have a last transaction T_f. In general executions may be finite or infinite.

Each source transaction is guaranteed to bring the sources from one consistent state to another. Given any serial schedule $R = T_{k_1}, T_{k_2}, \ldots T_{k_m}$, we define its *state sequence*, $R_{seq} = ss_{k_0}, ss_{k_1}, \ldots ss_{k_m}$ to be the base data states after each transaction commits. Corresponding to schedule R, state ss_{k_0} comprises (the value of) all base data before any transaction commits, and ss_{k_i} is the source state after transaction T_{k_i} commits.

We say that R_{seq} is a *consistent source state sequence* if R is equivalent to S.

(Recall that S is a serial schedule that represents the actual source execution.) Apparently, $S_{seq} = ss_0, ss_1, \ldots ss_f$ is itself a consistent source state sequence, with ss_f as the final source state after S completes. Corresponding to one execution, there may be more than one consistent source state sequence.

For a specific view V, $V[ss_i]$ is the result of computing the view V over the source state ss_i. That is, for each relation r that contributes to the view, $V[ss_i]$ is evaluated over r at the state ss_i. When there are multiple sources, for each relation r at source x that contributes to the view, $V[ss_i]$ is evaluated over r at the state $ss_i[x]$.

EXAMPLE 4.3.1 (Consistent source state sequence)
Consider a single data source and two relations $R(W, X)$ and $S(X, Y)$ at this data source, both relations are initially empty. Let there be two source transactions, each containing one update operation: $T_1 = \{insert(R, [1, 2])\}, T_2 = \{insert(S, [2, 3])\}$. Suppose $S = T_1, T_2$ is the source execution schedule. Schedule $R = T_2, T_1$ is another serial schedule that is equivalent to S. Let the view definition be $V = R \bowtie S$.

Let the source state sequence corresponding to S be $S_{seq} = ss_0, ss_1, ss_2$. Following figure shows the contents of base relations in each state of S_{seq} as well as the corresponding view $V[ss_i]$.

	R: W	X	S: X	Y	V:	W	X	Y
ss_0	-	-	-	-	$V[ss_0]$	-	-	-
ss_1	1	2	-	-	$V[ss_1]$	-	-	-
ss_2	1	2	2	3	$V[ss_2]$	1	2	3

Corresponding to execution R, there is a different but also consistent source state sequence $R_{seq} = ss'_0, ss'_1, ss_2$'. Following figure shows this state sequence.

	R: W	X	S: X	Y	V:	W	X	Y
ss'_0	-	-	-	-	$V[ss'_0]$	-	-	-
ss'_1	-	-	2	3	$V[ss'_1]$	-	-	-
ss'_2	1	2	2	3	$V[ss'_2]$	1	2	3

Notice that $V[ss_i]$ is not the content of a materialized view, but the "correct" value of view V at a given time according to the view definition. The actual content of the view is decided by the view maintenance algorithm. It may or may not agree with $V[ss_i]$. □

4.3.2 Warehouse States

Consider one view V at the warehouse, which is defined over a set of base relations at one or more sources. As the view is maintained, the warehouse goes through a sequence of *warehouse states*: $W_{seq} = ws_0, ws_1, \ldots, ws_q$. (We assume there is a final warehouse state after all activity ceases.) Each warehouse state ws represents the contents of the warehouse, and the warehouse state changes whenever the view is updated. The view at state ws_j is $V[ws_j]$.

4.3.3 Snapshot Consistency

Consider a materialized view V and a warehouse state ws_j.

Definition: We say that ws_j is *snapshot consistent* with source state ss_i, written $ws_j \doteq ss_i$, if V at ws_j has the same content as $V[ss_i]$, which is the result of evaluating the expression of V at source state ss_i. □

When $ws_j \doteq ss_i$, a warehouse application that reads view V at ws_j sees the content of the view had it been evaluated at the source state ss_i. We say that ws_j is a *consistent warehouse state* when it is snapshot consistent with a consistent source state. If a warehouse operation reads a view in a consistent warehouse state, then this operation can be conceptually mapped into a source read-only transaction that is serializable with other source transactions. Therefore, providing snapshot consistency is similar to providing serializability in data replications.

4.3.4 Consistency Over Time

We now redefine the four levels of consistency for warehouse views, each level subsumes all prior levels. The intuitive meanings of the consistency levels are the same as those in Section 3.1, the latter definitions can be seen as special cases that work for single-source warehouse with single-update transactions. Assume that the view at the warehouse is initially synchronized with the source data, i.e., $V[ss_0] = V[ws_0]$. Let the warehouse state sequence corresponding to the maintenance of view V be $W_{seq} = ws_0, ws_1, \ldots, ws_q$.

Definition: The warehouse state sequence W_{seq} is *convergent* if there exists a consistent source state sequence $S_{seq} = ss_0, ss_1, \ldots, ss_f$ and $ws_q \doteq ss_f$. That is, after the last update and after all activity has ceased, the view is consistent with the base data at the sources. □

Definition: The warehouse state sequence W_{seq} is *weakly consistent* if there exists a consistent source state sequence $S_{seq} = ss_0, ss_1, \ldots, ss_f$ such that $ws_q \doteq ss_f$, and for all ws_j, there exist $0 \leq i \leq f$ such that $ws_j \doteq ss_i$. That is, each warehouse state reflects some consistent source state. □

Definition: The warehouse state sequence W_{seq} is *strongly consistent* if there exists a consistent source state sequence $S_{seq} = ss_0, ss_1, \ldots, ss_f$ and a mapping m from warehouse to source states ($m(ws_j) = ss_i$ for some i), such that:

- Each warehouse state reflects some consistent source state. That is, for all ws_j, $ws_j \doteq m(ws_j)$.
- The order of warehouse states matches the order of corresponding source states. That is, if $ws_j < ws_k$, then $m(ws_j) < m(ws_k)$.
- Eventually, the view reflects the final state of the source base data. That is, $ws_q \doteq ss_f$. □

Definition: A warehouse state sequence W_{seq} is *complete* if it is strongly consistent, and for every source state ss_i, there exists a ws_j such that $m(ws_j) = ss_i$. That is, there is a complete order-preserving mapping between the states of the view and the states of the source. □

A view maintenance algorithm is responsible for bringing a view into a new consistent state after the source changes, it generates a sequence of warehouse states when maintaining the view. We define the consistency level of a maintenance algorithm to be the weakest of all consistency levels of the warehouse state sequences the algorithm may generate. For example, the algorithm CStrobe developed in Chapter 4 is complete for single update transactions. That means CStrobe will always generate a complete warehouse state sequence. Specifically, it processes one update U_j at a time and generates the warehouse view that is consistent with the source state after U_j executed. Algorithm Strobe, on the other hand, generates strongly consistent warehouse state sequences. It can batch multiple updates, U_i through U_{i+k}, bringing the warehouse from a state consistent with the sources before U_i to a state consistent with the sources after U_{i+k}. An algorithm may achieve different levels of consistency depending on the update transaction scenario to which it is applied.

Figure 4.2 shows the possible warehouse and source states for a view maintained by algorithms with different consistency levels. This view is defined on two base relations at two data sources. In the figure, a circle represents the state at a particular site, and the arrows can be interpreted as "computed from". By definition, it is possible for an algorithm with lower level of consistency (e.g., the strongly consistent Strobe algorithm) to generate warehouse state sequence with higher level of consistency (e.g., a complete state sequence), but not vice-versa.

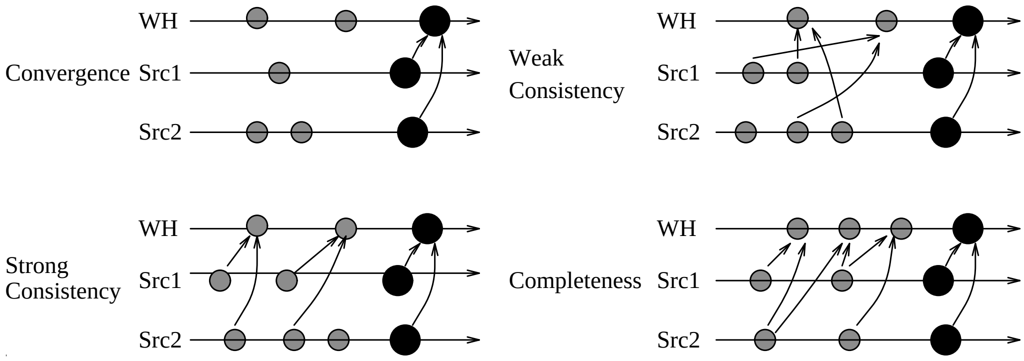


Figure 4.2: Consistency levels of a materialized view

4.4 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 4.6 we present modifications to these algorithms that attain completeness for their respective transaction scenarios.

4.4.1 Terminology

We use the same view as defined in last chapter: $V = \Pi_{proj}(\sigma_{cond}(r_1 \bowtie r_2 \bowtie \ldots \bowtie r_n))$. Any two relations may reside at the same or at different sources. For now, we assume that the projection list contains the key attributes for each relation.

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 SPJ 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). Since queries generated by an algorithm can also be unions of SPJ expressions, *next_source* can simply selects one of the expressions for evaluation. An improvement would be to find common subexpressions.

EXAMPLE 4.4.1 (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. The following procedure returns the answer for query Q.

Procedure $source\_evaluate(Q)$
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 Procedure

The procedure $source\_evaluate(Q)$ may return an incorrect answer when there are concurrent transactions at the sources that interfere with the query evaluation.

For example, in Example 4.1.1, 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.

4.4.2 The Strobe Algorithm

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

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

As defined in last chapter, 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. $V\langle U\rangle$ denotes the view expression V with the tuple U substituted for U's relation. The operation $key\_delete(R, U_i)$ deletes from relation R the tuples whose key attributes have the same values as U_i.

Algorithm 4.4.1 (Strobe Algorithm)

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

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

End Algorithm

The following example applies the Strobe algorithm to the warehouse scenario in Example 4.1.1. 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 4.4.2 (Strobe avoids deletion anomaly)

As in Example 4.1.1, 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 $MV = \phi$. We again consider two 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 is $\langle key\_delete(MV, U_2)\rangle$.

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 $A_2 = \phi$. 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 $MV = \phi$. 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 D.

4.5 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.

Before getting to the algorithm, we want to mention that the algorithms developed for processing single updates (e.g., Strobe, ECA) can handle batches of updates on the same base relation directly by changing the update U in an algorithm to a batch of updates B. T-Strobe and G-Strobe process sets of updates from different base relations. Like Strobe, they use key information to avoid anomaly. Compensating algorithms like ECA can also be extended to handle multiple updates on different base relations.

4.5.1 The 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. □

Algorithm 4.5.1 (Transaction-Strobe Algorithm (T-Strobe))

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

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 4.5.1 (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:

$$T_1 = \langle \mathit{delete}(r_1, [1,2]), \mathit{insert}(r_1, [3,4]) \rangle.$$

When the Strobe algorithm is applied to this scenario, the warehouse first adds the deletion to AL. Since there are no pending updates, AL is applied to MV and MV is updated to $MV = \phi$, 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

[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.

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.

4.5.2 The Global-Strobe Algorithm

While the T-Strobe algorithm is strongly consistent for source-local transactions, we show in the next example that it is only convergent 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 4.5.2 (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 $MV = \phi$. $AL = \langle\rangle$. 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 $UQS = \phi$, the warehouse applies AL to MV and $MV = ([1,3,5])$, which is a globally inconsistent state.

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 $UQS = \phi$, $AL = \langle key\_delete(MV, U_2), insert(MV, A_2)\rangle$ 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).

Example 4.5.2 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. $UQS = \phi$;

[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.

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 4.5.2, 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 [BHG87]. 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.

4.6 Completeness and Termination of the Algorithms

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

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

Before presenting the algorithm, we need a few definitions.

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

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

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

Definition: *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*. □

Algorithm 4.6.1 (Complete Strobe Algorithm(C-Strobe))

1. At the warehouse:
2. ▷ Initially, $Delta = \phi$.
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 $UQS = \phi$:
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 $Delta = \phi$.

End Algorithm

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 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 Appendix E.

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. (See Lemma C.0.8 in Appendix C.) 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 4.1.1, and show how C-Strobe processes this scenario differently from the Strobe algorithm (shown in Example 4.4.2).

EXAMPLE 4.6.1 (C-Strobe applied to example of introduction)

As in examples 4.1.1 and 4.4.2, 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 $MV = \phi$. We 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. $Delta = \phi$. 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. $UQS = \phi$, 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 $MV = \phi$. □

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 4.6.2 (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 4.6.1. We now consider a different set of events at the warehouse.

1. $Delta = \phi$. 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 $A^1_{1,1,0} = \phi$ from source x. This implies that $A_{1,1,0} = \phi$. 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 $UQS = \phi$, 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 $MV = \phi$. □

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.

4.7 Multi-source Compensating Algorithms

Compensating algorithms ECA, LCA, and CCA can be modified to handle views whose base relations span multiple data sources. In this environment, it is impossible to detect and correct view maintenance anomalies purely dependent on the event order. For instance, in Example 4.1.1, the answer to query Q_1, $A_{1,2}$, is received after update U_2. However, this answer actually did not "see" U_2. Thus, compensating algorithms must detect exactly which query is affected by which update. Therefore, we define the following concept in order to extend ECA to multi-source warehouses:

Definition: Define the *relevant unanswered query set* $RUQS$ of an update U_i as the set of queries Q_j that satisfy the following three conditions: (1) $Q_j \in UQS$ when U_i is received; (2) source_evaluate(Q_j) needs to visit $rel(U_i)$; (3) source_evaluate(Q_j) has not visited $rel(U_i)$. □

Now we extend algorithm ECA to obtain a strongly consistent algorithm MECA to maintain views in a multi-source data warehouse. The only difference between MECA and ECA is that the former compensates for queries in $RUQS$ instead of UQS.

Algorithm 4.7.1 (Multi-source ECA (MECA))

The source events and warehouse event W_ans_i behave exactly as in Algorithm 3.3.2 (ECA). At the warehouse, `COLLECT` $= \phi$ initially.

- W_up_i: receive U_i;
  let $Q_i = V\langle U_i\rangle - \sum_{Q_j \in RUQS} Q_j\langle U_i\rangle$
  send Q_i to the source;
  trigger event S_qu_i at the source

End Algorithm

MECA is strongly consistent for single update transactions over multiple data sources. We do not prove the correctness of MECA here. The basic idea is that the list of events generated at the warehouse using MECA could be generated by ECA, if all base relations reside at a single data source.

Analogous to ECA, algorithms LCA and CCA can also be extended to handle multi-source views. While UQS can be obtained by keeping track of queries sent by the warehouse, $RUQS$ need to be obtained by keeping track of "query fragments" sent to each data source (assuming the event order holds between the warehouse and each of the data source.) If it is impossible in practice to detect whether a query actually saw a source update; key algorithms have to be used.

4.8 Summary and Optimizations

In this chapter, we identified three fundamental transaction processing scenarios for data warehousing and developed the Strobe family of algorithms to consistently maintain the warehouse data. Figure 4.3 summarizes the algorithms of this and the previous chapters. In the figure, "Conventional" refers to a conventional centralized view maintenance algorithm.

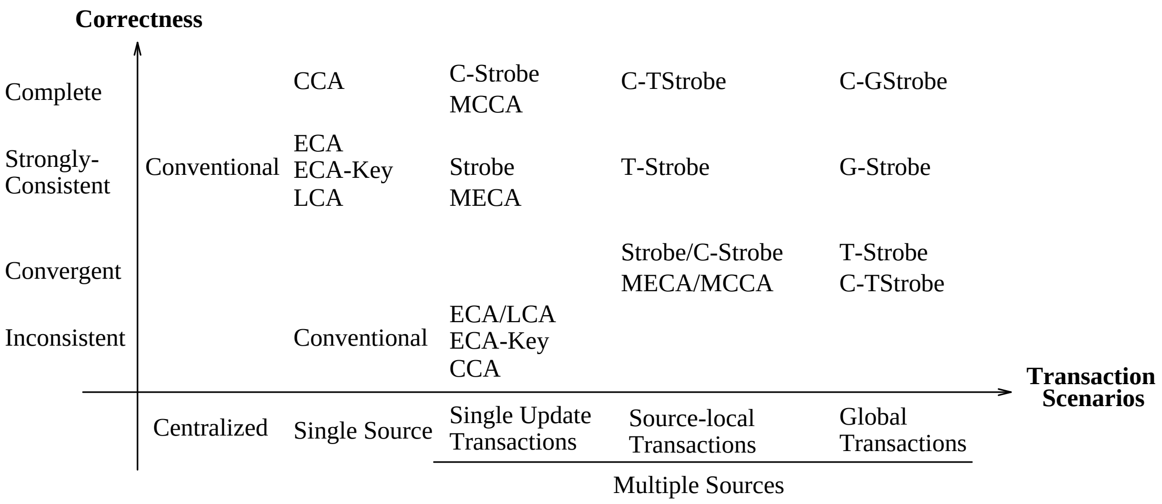


Figure 4.3: Consistency spectrum

In Figure 4.3, 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 also 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. We will come back to this issue in Section 6.6.

3. We can use key information to avoid sending some queries to sources [GW93]. 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 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 those in [BLT86]. 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.

We conclude the chapter with some discussion about the "event ordering" assumption we made in Section 4.2. 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 algorithm already handles the second case by using *pending*(Q_i), and also guarantees 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.

Chapter 5

Multiple View Consistency

There are two phases to making the warehouse consistent: making each individual view consistent, and making the set of warehouse views mutually consistent. In this chapter we study the latter, which we call the *multiple view consistency (MVC)* problem.

We provide a scalable architecture for consistently handling multiple views in a data warehouse. In this architecture, each view is managed by a separate concurrent process, increasing concurrency and easily allowing different algorithms for materializing each view. This architecture is implemented in our WHIPS warehouse prototype [WGL+96]. We then develop simple, scalable algorithms for achieving MVC at a warehouse. The algorithms coordinate concurrent view managers so that their warehouse updates do not violate consistency. Each algorithm can be thought of as a specialized concurrency control mechanism that exploits the application semantics (incremental view maintenance) to achieve greater concurrency.

In Section 5.1 we illustrate the multiple view consistency problem by an example, then address the importance of MVC. In Section 5.2, we formally define the consistency levels a view maintenance algorithm can ensure for multiple views. Section 5.3 provides more details of our system architecture and framework. We develop two algorithms, the Simple Painting Algorithm and the Painting Algorithm, used to achieve MVC in Sections 5.4 and 5.5, respectively. Then we conclude in Section 5.7, by discussing extensions of the algorithms.

5.1 Introduction

In this section we first give an example of the multiple view consistency problem in a data warehouse, then we argue that multiple view consistency is important for some applications, and sketch the basic idea behind our architecture and MVC algorithms.

EXAMPLE 5.1.1 (The multiple view consistency problem)
Suppose there are two views at the warehouse defined as $V_1 = R \bowtie S$ and $V_2 = S \bowtie T$. The contents of the base relations and the views are shown in the following Table 5.1.1.

	R:		S:		T:		V_1 :			V_2 :		
Time	A	B	B	C	C	D	A	B	C	B	C	D
t_0	1	2	-	-	3	4	-	-	-	-	-	-
t_1	1	2	2	3	3	4	-	-	-	-	-	-
t_2	1	2	2	3	3	4	1	2	3	-	-	-
t_3	1	2	2	3	3	4	1	2	3	2	3	4

Table 5.1: Changes of base relations reflect in multiple views

At time t_0, relation S is empty, so both V_1 and V_2 are empty. At t_1, a tuple $[2, 3]$ is inserted into S. At t_2, changes to V_1 are computed (by joining tuple $[2, 3]$ with relation R) and the result is inserted into view V_1. At t_3, changes to V_2 are computed and the result is inserted into view V_2.

For both V_1 and V_2, the above view maintenance steps are correct. That is, their contents are correctly updated to reflect the change in the base relation S. However, after time t_2, V_1 reflects the new state of relation S but V_2 does not. The two views are not consistent with each other.

Notice that computing consistent values for each view, e.g., obtaining tuple $[1, 2, 3]$ of V_1, is not as simple as it may first appear. As we showed in the last few chapters, while we are computing this tuple, base relation R or S could be updated. Then, results may need to be compensated, post-processed, or combined with others before it can be applied to the view. The algorithms we developed in Chapters 3 and 4 can be used to ensure that each view reflects consistent data. The algorithms we develop in this chapter work with single view consistency algorithms to ensure that updates to *multiple* views are applied in a mutually consistent fashion. □

5.1.1 Importance of Multiple View Consistency

As with traditional database systems, data consistency may or may not be important, depending on the application. Many current warehouses are used only for statistical or trend analysis, and inconsistencies may not have an impact on the final results. However, warehouses are being used more and more for other activities where consistency may be very important. For example, a warehouse may be used to handle customer inquiries, removing load from the operational systems. When the customer calls with a question, we would like to be able to read her data consistently: her checking account record, for instance, should match with her linked savings account record. If the warehouse is trying to identify *particular* customers for a special promotion (and not just general trends) based on data from multiple views, it would be useful to get the correct customers.

It is also important to note that MVC is required by some view maintenance algorithms. For example, in the multiple view maintenance problem described in [RSS96, LQA97], auxiliary views are stored in order to maintain primary views efficiently. For example, in order to maintain $V = R \bowtie S \bowtie T$, the algorithm might choose (according to some heuristic) to materialize relations $R \bowtie S$ and $S \bowtie T$ and compute V from them. The two sub-views must be consistent with each other whenever V is computed. Auxiliary views may also be stored to guarantee view self-maintainability [GJM96, QGMW96].

The key problems in keeping multiple views consistent at the warehouse are as follows.

1. One source update transaction may invoke a set of actions on multiple warehouse views, as we saw in Example 5.1.1. These actions must be applied at the warehouse as an "atomic unit." These atomic units need to be applied at the warehouse in the same (or a consistent) order as the original updates.

2. Computing those actions, or the *delta computation* of views, often takes time. It may involve queries back to the sources if base data is not cached at the warehouse. This makes it desirable to process both different updates and different views concurrently.

3. A delta computation not only takes time but it may be "intertwined" with subsequent updates (e.g., when strongly consistent view managers are used). For instance, in Example 5.1.1, in between times t_1 and t_2 we computed the join of the new S tuple $[2, 3]$ with R. If R is updated before we read it, we may get fewer or more tuples than what we wanted. One way out of this dilemma is to combine both updates (the original one to S and the new one to R) into a single "atomic unit;" however, we now need the machinery to combine the updates.

The simplest solution to the MVC problem is to create a single *integrator process* that handles updates sequentially. For each update it receives, it sequentially computes the changes to all views that result from the update. (If later updates are intertwined, they must be pulled into the computation. This in itself must be done very carefully to avoid violating consistency.) After all the view changes are computed, the process submits a transaction to the warehouse, waits for it to commit, and moves on to the next update. Clearly, this does not allow for any concurrency, and given Problem 2 above, is not acceptable in a high update rate environment. However, note that this sequential approach to the MVC problem is taken (by default) by many current proposed view maintenance algorithms, e.g., [HZ96a].

5.1.2 Merge Process to Enforce MVC

Instead, we propose an architecture that allows as much parallelism as possible while still guaranteeing a correct solution to the MVC problem. Our architecture is shown in Figure 5.1. Updates on base data are reported from data sources to the integrator, and then forwarded to relevant view managers. Each view manager is a separate process that handles the delta computation for one view. View managers may reside on different machines. After computing the changes to its view, each view manager sends a list of actions to a *merge process*. The merge process collects changes to the views, holds them until all affected views can be modified together, and then forwards all of the views' changes to the warehouse in a single warehouse transaction. The merge process also keeps track of dependencies between warehouse transactions in order to effectively control their commit order.

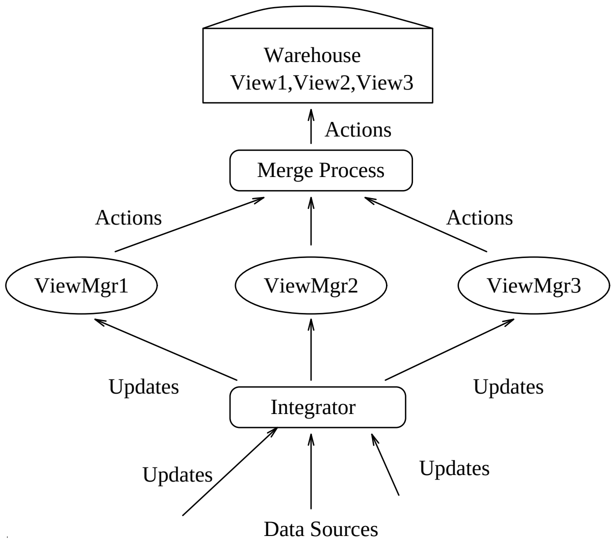


Figure 5.1: Merge process to enforce MVC

Returning to Example 5.1.1, the merge process will receive the changes to V_1 at time t_2 (from the V_1 manager), but it will hold them until it receives the corresponding changes to V_2 at time t_3. The merge process will then notify the warehouse to update both views in a single transaction. In this chapter we provide algorithms for the merge process to decide when to hold and when to forward actions.

Although we split the view maintenance computation by view, there are other options to parallelize the system. For example, one could choose to use one process per update, or to use multiple threads within a process. The MVC problem arises with each of these options, and the algorithms we present in this chapter can be adapted to those scenarios. However, for correctness, we only consider the architecture of Figure 5.1. Incidentally, this view manager architecture does have one important advantage over other parallel architectures: Since each view is under the control of a separate process, it is very easy to use different maintenance algorithms for each view. This selection of an algorithm on a per view basis is very important since some views, e.g., aggregate views, need to use different maintenance algorithms than other views, e.g., a copy of a base relation. It is also possible to design specific view managers to handle new, non-traditional types of views without affecting the existing system. For example, a specialized view manager might predict the new results of data mining tests on a view using the changes to be integrated into the view. Our architecture also makes it easy to add and delete views on the fly.

5.2 Multiple View Consistency

We divide the consistency problem into three layers: source consistency, view consistency and multiple view consistency (MVC). Source consistency refers to the consistency among base data, and is enforced by source transaction protocols (if any). View consistency refers to the consistency between a view at the warehouse and its base data at the sources and is enforced by the view managers. MVC refers to the mutual consistency among views, which will be enforced by the MVC algorithms we develop in Sections 5.4 and 5.5. Figure 5.2 shows where these three layers of consistency lie in the system.

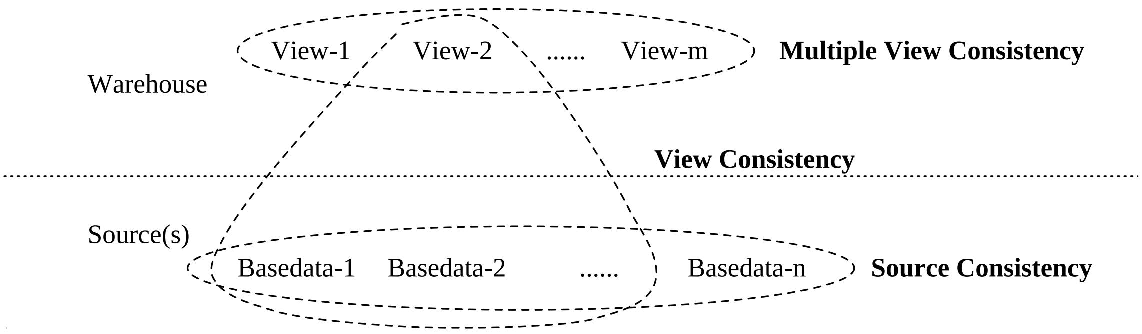


Figure 5.2: Three layers of consistency in a data warehouse

A view manager is responsible for bringing a view into a new consistent state after the source changes. Different view managers can be designed to maintain a view with different consistency levels. A *complete view manager* generates a complete warehouse state sequence. It processes one update U_j at a time and generates the warehouse view that is consistent with the source state after U_j executed. A *strongly consistent view manager* generates strongly consistent warehouse state sequences. It can batch multiple updates, U_i through U_{i+k}, bringing the warehouse from a state consistent with the sources before U_i to a state consistent with the sources after U_{i+k}.

When there are multiple views at the warehouse, a *warehouse state ws* is a vector with one element for the state of each view. Each warehouse view maintenance transaction updates one or more views. The warehouse state advances after each warehouse transaction and the sequence of warehouse maintenance transactions yields a sequence of warehouse states $W_{seq} = ws_0, ws_1, \ldots, ws_q$.

Definition: We say that ws_j is *multiple view consistent* with source state ss_i, written $ws_j \equiv ss_i$, if for each view V at the warehouse state ws_j, its content is the same as $V[ss_i]$. □

The definitions for multiple view consistency are very similar to that for single view consistency (Section 4.3). All we need to do is replace "$\doteq$" by "$\equiv$" in our previous definitions. For example, a warehouse system is strongly consistent if its warehouse state sequence is strongly consistent, as stated by our earlier definition with "$\doteq$" replaced by "$\equiv$." With such a system, each materialized view will be strongly consistent, and for each warehouse state, all of the views will be mutually consistent. The two algorithms we present in Sections 5.4 and 5.5 guarantee MVC completeness and strong consistency, respectively.

5.3 System Framework

In this section we present the basic framework for our view maintenance algorithms. In particular, we describe how the processes of Figure 5.1 interact.

5.3.1 Data Model

In all of the discussions and algorithms in this chapter, the data at the sources and at the warehouse need not be relational. (Note that as in the previous chapters, the consistency definitions of Section 5.2 are also independent of the data model.) However, for simplicity, in our examples we use a relational model with simple SPJ views. Also, for our examples we assume that each update is a single tuple insert or delete. In Section 5.7.1 we show that the algorithms developed in this chapter can be easily extended to handle source transactions involving multiple sources and updates.

5.3.2 Integrator Process

The integrator receives updates from each source. We assume that updates from the same source arrive at the integrator in the order they committed.

When the integrator receives an update, it performs the following steps.

1. The integrator numbers the updates by arrival order. For example, U_5 is the fifth update received. Let U_i be the update that just arrived.

2. The integrator determines the *relevant* views for U_i, REL_i. A view is relevant to U_i if it needs to be modified because of U_i. For example, for a relational model, the integrator can determine the source relation R that was modified by U_i. Then it can include in REL_i all views that use R in their definition. We could be more discerning by using selection conditions in the view definitions or other optimizations to rule out irrelevant updates[JBL86].

3. The integrator sends REL_i to the merge process. The subscript i is implicitly sent, so that the merge process knows to which update REL_i refers. The set REL_i will be used by the merge process to know from which view managers to expect actions.

4. The integrator sends a copy of U_i to each view manager responsible for a view in REL_i. Again, the subscript i is implicitly included.

Notice that it is not necessary for the integrator to send a relevant view set REL_i directly to the merge process. An alternative could be for the integrator to send REL_i to one or more of the view managers in that set, together with U_i. The view managers could then forward REL_i to the merge process when they deliver their action lists (see below). This reduces the number of messages and may be more efficient than the scheme we adopt here. However, it is easier to describe the merge process when REL_i arrives directly, so this is what we use. The changes to the merge process to handle the alternate scheme are straightforward.

5.3.3 View Managers

A view manager VM_x receives (in the order sent by the integrator) a sub-sequence of updates $U_{i_1}, U_{i_2}, \ldots U_{i_f}$. If VM_x is complete, then for each update U_j it will generate an *action list* AL_j^x. This list contains the operations necessary to make view V_x (handled by VM_x) consistent with the source state existing after U_j was performed at its source. The view manager then forwards AL_j^x to the merge process, in order by j subscript. The view manager identifier, x, and the update identifier, j, are implicitly included. If an action list AL_j^x happens to be empty, it is still sent to the merge process. (This is not essential; it just simplifies our merge algorithm.)

A strongly consistent view manager VM_x is similar, except that several updates $U_{i_k}, \ldots U_{i_{k+n}}$ can generate a single action list $AL_{i_{k+n}}^x$. Notice that the subscript of the action list identifies the last update that is included in the batch. Thus, again, AL_j^x contains the actions needed to make the view consistent with the source state existing after U_j executed.

In our solution, all actions pass through the merge process where they are coordinated and later forwarded to the warehouse. An alternative is to let the integrator in Figure 5.1 run a traditional distributed transaction processing protocol involving all relevant view managers and the warehouse. When update information is received, the integrator then starts a transaction within which each relevant view manager executes actions on its view directly at the warehouse. The transaction ensures that the set of actions executed is serialized with respect to actions belonging to other transactions. However, the merge process (running the algorithms we will present here) is still necessary in order to coordinate transaction commits. This alternative approach also binds one warehouse transaction to each update, while our framework allows each warehouse transaction to correspond to multiple intertwined source updates. Further, the alternative approach is only possible if different processes may contribute actions to the same transaction and this is difficult to do in some commercial database systems.

5.4 Simple Painting Algorithm

The Simple Painting Algorithm (SPA) is used by the merge process to maintain MVC at the warehouse when all view managers are complete. As we will see in Section 5.4.4, SPA guarantees complete warehouse states. SPA receives action lists from view managers and REL_i sets from the integrator, and generates transactions for the warehouse database system. We make no restrictions on message arrival order, except that messages from the same process must arrive in the order sent. This means that the merge process may receive a list AL_j^x without having received REL_j. This situation is taken care of by the algorithm.

5.4.1 Data Structures

Through an example, we introduce a data structure called the (*ViewUpdateTable*) (*VUT*) that we will use in the Simple Painting Algorithm.

EXAMPLE 5.4.1 (ViewUpdateTable (VUT))

Suppose we have three views: $V_1 = R \bowtie S$, $V_2 = S \bowtie T \bowtie Q$ and $V_3 = Q$. There are two source updates: U_1 on S and U_2 on Q. When the merge process receives REL_1 and REL_2, it builds the following (*ViewUpdateTable*) (*VUT*). The *VUT* is a two dimensional table. $VUT[i, x]$ corresponds to update U_i and view V_x.

VUT:

	$V_1(R,S)$	$V_2(S,T,Q)$	$V_3(Q)$
$U_1(S)$	(white)	(white)	(black)
$U_2(Q)$	(black)	(white)	(white)

In SPA, each entry $VUT[i, x]$ contains a single *color* field. $VUT[i, x].color$ indicates whether an update is related to a view, and whether the corresponding action list has been received. SPA initializes the color for each update U_i and view V_x when it first receives REL_i. There are four possible colors:

- *white* (w for short): waiting for the corresponding action list for this entry;
- *red* (r for short): the corresponding action list has been received, however, the merge process is waiting for other actions before applying it;

- *gray* (g for short) : the corresponding action list has just been applied;
- *black* (b for short): the entry need not be examined.

We know that a complete view manager sends one AL per relevant update. The merge process waits for one AL for each entry in the VUT whose color is *white*. As the action lists are received, they are saved in an array WT until they can be applied. To illustrate, suppose that the merge process receives AL_1^2 from VM_2 (which handles V_2). The merge process saves this action in WT_1 and sets the color of the corresponding entry:

	V_1	V_2	V_3	WT_i
U_1	w	w	b	ϕ
U_2	b	w	w	ϕ

$\Longrightarrow$

	V_1	V_2	V_3	WT_i
U_1	w	r	b	$\{AL_1^2\}$
U_2	b	w	w	ϕ

The merge process at this time cannot apply actions in AL_1^2 to the warehouse because it knows from the VUT that U_1 also affects V_1 and it needs to wait for the corresponding actions from VM_1. Only after the merge process receives AL_1^1 can it apply both action lists together and update both views. When the warehouse applies an action list, the corresponding entry in the VUT is set to *gray*. A row in the VUT can be purged when all actions for the row have been applied to the views. □

Notice that collecting the action lists and applying them together may not always be as trivial as in the above example. For example, the receipt of an action list in row i may enable action lists in later rows to be applied. Also, some actions corresponding to later updates may be applied before actions for earlier ones, provided that those updates do not affect the same views. SPA handles all of these cases, some of which we will illustrate in Example 5.4.2.

5.4.2 Simple Painting Algorithm (SPA)

The family of MVC algorithms are called Painting Algorithms because applying the action lists is like painting a grid wall from top to bottom, following "guidelines." For example, one guideline is that a grid row can only be painted *gray* when all of its entries are either *red* or *black*. This ensures that a view is only updated to a consistent state, when all relevant action lists have arrived.

SPA is driven by two kinds of events: the receipt of REL_i from the integrator and the receipt of AL_i^x from view manager VM_x. As stated earlier, it is possible that the merge process receives AL_i^x before it receives REL_i. In this scenario, the merge process needs to delay the processing of AL_i^x until after REL_i arrives.

Algorithm 5.4.1 (Simple Painting Algorithm)

Initially, $WT_i = \phi$ for all i.

$\triangleright$ When the merge process receives REL_i
- — Allocate a new row i in the VUT. $VUT[i, x]$ refers to U_i and $V_x \in VM$.
- — For all $V_x \in REL_i$ set $VUT[i, x].color = white$; otherwise set $VUT[i, x].color = black$.
- — For all $AL_i^x \in WT_i$, call $ProcessAction(AL_i^x)$.

$\triangleright$ When the merge process receives action list AL_i^x:
- — Let $WT_i = WT_i \cup AL_i^x$.
- — If REL_i has arrived, call $ProcessAction(AL_i^x)$.

Procedure $ProcessAction(AL_i^x)$
Let $VUT[i, x].color = red$ and call $ProcessRow(i)$.
End Procedure

Procedure $ProcessRow(i)$
Line 1: If $\exists x$, $VUT[i, x] = white$, return.
Line 2: If $\exists x, \exists i' < i$, $VUT[i, x] = red$ and $VUT[i', x] = red$, return.
Line 3: For any $x \in VM$, if $VUT[i, x].color = red$, then let $VUT[i, x].color = gray$.
Line 4: Apply all actions in WT_i as a single warehouse transaction. [See Section 5.4.3.]
Line 5: For all $VUT[i, x] = gray$, if $nextRed(i, x) \neq 0$, then call $ProcessRow(nextRed(i, x))$.
Line 6: Purge row i from the VUT. Return.
End Procedure
End Algorithm

Procedure $ProcessRow(i)$ checks whether all action lists in row i and earlier have arrived and can be applied to yield a new consistent warehouse state. If so, it applies all actions in row i to the views in a single warehouse transaction. Then it recursively calls itself to check whether subsequent rows of actions can now be applied. We further explain some lines in $ProcessRow(i)$ after presenting the algorithm.

Let VM be the set of all view managers. In SPA, $nextRed(i, x)$ is a shorthand for the row number of the next *red* entry below $VUT[i, x]$. It is the number of the AL received after AL_i^x from view manager x. If AL_i^x is currently the last AL the merge process has received from view manager x, then $nextRed(i, x) = 0$.

In $ProcessRow(i)$, Line 1 tests whether any action in this row has not yet arrived; if so, the corresponding view cannot be brought to state i (i.e., other ALs in row i cannot be applied). Line 2 checks another condition that must be satisfied before we apply the ALs in row i: for each received AL_i^x, previous lists from VM_x must have been already applied. This ensures that lists from the same view manager are applied to the warehouse in the order they are generated by the manager. If row i passes both tests, actions in this row can be applied to views stored at the warehouse. Note that applying actions in one row may trigger the application of actions in later rows. Line 5 finds all such later rows.

EXAMPLE 5.4.2 (Simple Painting Algorithm)
This example shows how SPA handles ALs and guarantees warehouse view maintenance with *complete* consistency. Suppose there are three views: $V_1 = R \bowtie S$, $V_2 = S \bowtie T$, and $V_3 = Q$. Notice that V_3 is disjoint with V_1 and V_2. Let there be three source updates: U_1 on S, U_2 on Q and U_3 on T. Assume the merge process receives $REL_1, AL_1^2, REL_2, REL_3, AL_2^3, AL_3^2, AL_1^1$, in that order. We use time t_i to refer to the changes in the VUT (rather than to changes in the views). Let times t_0, t_1, t_2, t_3 and t_4 correspond to the receipt of $REL_1, AL_1^2, REL_2, REL_3$ and AL_2^3. We show the VUT changes starting from time t_4.

Time t_4: AL_2^3 arrived

	V_1	V_2	V_3	WT_i
U_1	w	r	b	$\{AL_1^2\}$
U_2	b	b	r	$\{AL_2^3\}$
U_3	b	w	b	ϕ

$\Longrightarrow$ Time t_5: WT_2 applied

	V_1	V_2	V_3	WT_i
U_1	w	r	b	$\{AL_1^2\}$
U_2	b	b	g	ϕ
U_3	b	w	b	ϕ

$\Longrightarrow$ Time t_6: Row 2 purged

	V_1	V_2	V_3	WT_i
U_1	w	r	b	$\{AL_1^2\}$
U_3	b	w	b	ϕ

$\Longrightarrow$ Time t_7: AL_3^2 arrived

	V_1	V_2	V_3	WT_i
U_1	w	r	b	$\{AL_1^2\}$
U_3	b	r	b	$\{AL_3^2\}$

Time t_8: AL_1^1 arrived

$\Longrightarrow$

	V_1	V_2	V_3	WT_i
U_1	r	r	b	$\{AL_1^1, AL_1^2\}$
U_3	b	r	b	$\{AL_3^2\}$

Time t_9: WT_1 applied

$\Longrightarrow$

	V_1	V_2	V_3	WT_i
U_1	g	g	b	ϕ
U_3	b	r	b	$\{AL_3^2\}$

Time t_{10}: WT_3 applied

$\Longrightarrow$

	V_1	V_2	V_3	WT_i
U_1	g	g	b	ϕ
U_3	b	g	b	ϕ

Time t_{11}: Row 1,2 purged

$\Longrightarrow$

V_1	V_2	V_3	WT_i

At time t_5, $WT_2 = \{AL_2^3\}$ is applied to V_3 because all expected ALs in row 2 have been received. Notice that at this time, actions in the first row have not yet been applied. However, AL_2^3 can be applied because the merge process knows from the VUT that all entries prior to row 2 in the third column of the VUT are *black*, meaning that the first update is irrelevant to V_3. At times t_9 and t_{11}, WT_1 and WT_3 are applied respectively.

At times t_6 and t_{10}, rows 2 and 1 are purged from the table, respectively. We also purge all corresponding ALs after they are applied to the warehouse views. Often, the warehouse only needs to work with a small VUT as a result of purging. Although theoretically, the total number of rows in the VUT could be as many as the total number of updates, the actual number is small in a system where no view manager is a bottleneck. When all ALs of all source updates received by the merge process have been processed, all VUT entries will be empty. □

5.4.3 Submitting View Maintenance Transactions to the Warehouse

There is still one remaining problem with algorithm SPA as just presented. In Example 5.4.2, WT_1 is applied at time t_9 and WT_3 is applied at time t_{11}. When the transactions are submitted to the warehouse, it is possible that the warehouse DBMS will commit WT_3 before WT_1. If so, the state of view V_2 will be invalid because it will reflect update U_3 but not U_1.

Let $VS(WT_i)$ be the set of views that WT_i will update; it can be obtained directly from the VUT. We say that transaction WT_j *depends* on WT_i if $j > i$ and $VS(WT_j) \cap VS(WT_i) \neq \phi$. To satisfy MVC, two dependent warehouse transactions need to commit in the order in which they were submitted.

There are a few solutions to this problem; each may be appropriate in different scenarios. The most straightforward way is to execute the warehouse transactions sequentially — only submit one to the warehouse after the previously transaction has committed. This solution is good when the warehouse transactions are small and the transaction overheads are low. The merge process can also choose to only sequence dependent transactions instead of all transactions, that is, only delay a transaction when some transaction it depends on has not committed. If the warehouse DBMS can provide transaction dependency capabilities, an alternative solution is for the merge process to submit transactions with dependency information and let the warehouse DBMS handle the execution sequence.

When transaction overhead is high, the merge process can batch several WT_is and submit them to the warehouse as one batched warehouse transaction (BWT). A BWT contains all ALs in a set of WTs that are ready to be submitted to the warehouse. The merge process needs to make sure that if WT_j depends on WT_i, then all ALs in WT_i appear before all ALs in WT_j when they are batched together. A parallel DBMS at the warehouse may be able to execute some of the actions in a BWT in parallel, as long as the execution is equivalent to a serial one.

Dependencies exist and need to be handled between BWTs, just as for WTs. Therefore, on one hand, batching eliminates the dependency problems between WTs within a batch; on the other hand, batching may effectively create dependencies between WTs that were independent. For example, suppose the merge process is ready to send WT_1, WT_2, WT_3 in that order, and both WT_2 and WT_3 depend on WT_1. By batching WT_1, WT_2 into BWT_1, the merge process need not handle the dependencies between WT_1 and WT_2. On the other hand, now WT_3 depends on BWT_1 (which contains WT_1), so WT_3 cannot commit before BWT_1. Thus, now WT_3 and WT_2 cannot be executed in parallel, unless the warehouse can handle transaction dependencies. Choosing the right size and the right transactions to batch together are optimization problems that we do not address here. Finally, note that batching only yields strong consistency at the warehouse rather than complete consistency, because each BWT may advance the warehouse state by more than one.

5.4.4 Properties of SPA

The Simple Painting Algorithm is complete according to the MVC definition given in Section 5.2. The proof of completeness is given in Appendix F. The completeness of SPA guarantees that when SPA is applied to an initially consistent warehouse state, it generates a complete warehouse state sequence. That is, the warehouse views reflect every single change of the source base data in the correct order.

Another important property of the SPA algorithm is that it applies action lists *promptly*, i.e., it does not delay actions unnecessarily. Not all MVC complete algorithms have this property. For instance, we could devise an algorithm that waits until all actions about all source updates (U_1 to U_f) arrive, then applies $WT_1, \ldots WT_f$ to the warehouse in that order. This algorithm is also complete under MVC, but is clearly not a desirable one because it unnecessarily delays actions. From the way we constructed algorithm SPA, it is easy to see that it is *prompt* and that no other algorithm can deliver updates earlier than SPA without possibly causing inconsistencies.

5.5 Painting Algorithm (PA)

In this section we develop a merge algorithm for underlying view managers that are strongly consistent but not necessarily complete (e.g., Strobe view managers). Strongly consistent view managers are usually more efficient and easier to implement than complete view managers in a distributed warehouse environment. The resulting Painting Algorithm (PA) is strongly consistent and prompt.

As mentioned in Section 5.3, a strongly consistent view manager sends a sequence of ALs: $AL^x_{i_1}, AL^x_{i_2}, \ldots, AL^x_{i_f}$ to the merge process. $AL^x_{i_j}$ brings view V_x from state i_{j-1} to state i_j. Since one AL can represent several intertwined source updates, SPA breaks down, as shown in the following example.

EXAMPLE 5.5.1 (Strongly consistent view management)
As in Example 5.4.1, suppose there are three views $V_1 = R \bowtie S$, $V_2 = S \bowtie T \bowtie Q$, and $V_3 = Q$. Let there be three source updates: U_1 on S, U_2 on Q and U_3 on S. Assume the merge process first receives REL_i for $i = 1, 2, 3$. Then it receives AL_3^1 to update V_1 for both U_1 and U_3, i.e., there is no separate AL_1^1. At this point, SPA makes $VUT[3,1].color$ equal to *red*. If we do not make $VUT[1,1].color$ *red* too, then we will never apply row 1, so let us assume we do make $VUT[1,1].color$ *red*. Later, the merge process receives all other ALs corresponding to U_1 and U_2, as shown in the table entries below. At this time, SPA would apply rows 1 and 2 because all entries are either *red* or *black*. However, this would be incorrect: since AL_3^2 has not arrived yet, we cannot apply row 3, and since AL_3^1 and AL_1^1 are combined, we cannot apply row 1 either. And if we do not apply row 1, then we cannot apply row 2. Thus, we see that with intertwined updates, the consistency algorithm must be more careful.

	$V_1(R,S)$	$V_2(S,T,Q)$	$V_3(Q)$	WT_i
$U_1(S)$	r	r	b	$\{AL_1^2\}$
$U_2(Q)$	b	r	r	$\{AL_2^2, AL_2^3\}$
$U_3(S)$	r	w	b	$\{AL_3^1\}$

□

There are two major differences between PA and SPA. First, in PA, receiving an action list in row i may cause actions in the previous rows to be applied. In the above example, the receipt of AL_3^2 will cause actions in the previous two rows to be applied. Second, it is not guaranteed that each view will go through each state, i.e., PA is strongly consistent but not complete. In Example 5.5.1, all three views will be brought into state 3 directly, skipping states 1 and 2. The strongly consistent property of the underlying view manager algorithms makes it impossible to develop an MVC algorithm that guarantees completeness.

We also use the *ViewUpdateTable* (*VUT*) in the Painting Algorithm. In addition to the *color* field, we add a second field $VUT[i,j].state$ to indicate the next state of a view after applying the related action lists. In the above Example 5.5.1, when the merge process receives AL_3^1, it fills in the state of entry $VUT[1,1].state = 3$. This shows that U_1 is intertwined with U_3. At this time, even if all other actions corresponding to U_1 have been received, they cannot be applied unless all actions in WT_3

can be applied as well. Procedure $ProcessRow(i)$ now has a boolean return value, indicating whether ALs in row i and all related rows can be applied to warehouse views. An integer list $ApplyRows$ is used to remember the possible rows whose actions will be applied.

Procedure $ProcessRow(i)$ may recursively call itself to process rows both prior to and after i. It is therefore possible that $ProcessRow(i)$ is called again for the same i. Line 1 and Line 3 in the procedure make sure that the recursive calls terminate. For example, at time t_6 in Example 5.5.2 below, $ProcessRow(3)$ calls $ProcessRow(2)$ which again calls $ProcessRow(3)$. The last call realizes that it is a repeat call because 3 is already in $ApplyRows$, and returns $true$ in Line 1. If $ProcessRow(i)$ fails, actions in row i will not be applied and $ApplyRows$ will be set to empty before the next time the procedure is called.

As in $ProcessRow(i)$ in SPA, Lines 2 and 4 in the procedure check whether any actions in row i have not yet arrived or any actions in previous, relevant, rows cannot yet be applied. Line 5 says that if some entries in row i must be brought to a later state j directly, then all actions in row j must be applied together with the actions in row i. If any actions in row j cannot be applied, neither can the actions in row i. In Line 6, all actions in all rows in $ApplyRows$ are applied to the warehouse in a single transaction. After the actions are applied, Line 9 checks whether any actions in later rows can be applied as a consequence of applying actions related to row i.

Transaction dependency problems also arise in a merge process running PA, and the solutions we described for SPA can be used for PA as well.

The Painting Algorithm is strongly consistent under MVC. The proof that PA guarantees strong consistency is similar to the proof that SPA is complete, and is omitted here. Similar to SPA, PA also guarantees promptness.

Algorithm 5.5.1 (Painting Algorithm)

The main body of PA is the same as SPA, except that when REL_i is received, the merge process also sets $VUT[i,x].state = 0$ for all $V_x \in VM$. The two procedures are defined as follows.

Procedure $ProcessAction(AL_i^x)$
For all $i' \leq i$ such that $VUT[i',x].color = white$, let $VUT[i',x].color = red$ and $VUT[i',x].state = i$.
Let $ApplyRows = \phi$, call $ProcessRow(i)$ and ignore the return value.
End Procedure

Procedure $ProcessRow(i)$: boolean
Line 1: If $i \in ApplyRows$, return $true$.
Line 2: If $\exists x$, $VUT[i,x].color = white$, then return false.
Line 3: Add row i to $ApplyRows$.
Line 4: For all x such that $VUT[i,x].color = red$, for all $i' < i$ such that $VUT[i',x].color = red$, call $ProcessRow(i')$. If any of the return values is $false$, then return $false$.
Line 5: For all x such that $VUT[i,x].state > i$, call $ProcessRow(VUT[i,x].state)$. If any of the return values is $false$, then return $false$.
Line 6: For each $j \in ApplyRows$, for each $x \in VM$, if $VUT[j,x].color = red$, then let $VUT[j,x].color = gray$.
Line 7: For all rows $j \in ApplyRows$, apply all actions in WT_j together as a single warehouse transaction.
Line 8: Let $ApplyRows = \phi$.
Line 9: For all i,x such that $VUT[i,x].color = gray$, if $nextRed(i,x) \neq 0$, call $ProcessRow(nextRed(i,x))$.
Line 10: Purge rows whose entries are all either $black$ or $gray$ from the VUT, and return $true$.
End Procedure
End Algorithm

EXAMPLE 5.5.2 (Painting algorithm)

This example shows how the Painting Algorithm works. We have the same three views as in the last example. There are three source updates: U_1 on S, U_2 on Q, and U_3 on Q. Assume the merge process receives the following information in order: REL_1, REL_2, REL_3, AL_1^2, AL_3^2, AL_2^3, AL_1^1, AL_3^3. Here we show the changes in the VUT entries.

Time t_0: RELs received

	$V_1(R,S)$	$V_2(S,T,Q)$	$V_3(Q)$	WT_i
$U_1(S)$	(w,0)	(w,0)	(b,0)	ϕ
$U_2(Q)$	(b,0)	(w,0)	(w,0)	ϕ
$U_3(Q)$	(b,0)	(w,0)	(w,0)	ϕ

$\Rightarrow$ Time t_1, t_2: AL_1^2, AL_3^2 arrived

	V_1	V_2	V_3	WT_i
U_1	(w,0)	(r,1)	(b,0)	$\{ AL_1^2 \}$
U_2	(b,0)	(r,3)	(w,0)	ϕ
U_3	(b,0)	(r,3)	(w,0)	$\{ AL_3^2 \}$

$\Rightarrow$ Time t_3, t_4: AL_2^3, AL_1^1 arrived

	V_1	V_2	V_3	WT_i
U_1	(r,1)	(r,1)	(b,0)	$\{ AL_1^2, AL_1^1 \}$
U_2	(b,0)	(r,3)	(r,2)	$\{ AL_2^3 \}$
U_3	(b,0)	(r,3)	(w,0)	$\{ AL_3^2 \}$

$\Rightarrow$ Time t_5: Row 1 applied

	V_1	V_2	V_3	WT_i
U_1	(g,1)	(g,1)	(b,0)	ϕ
U_2	(b,0)	(r,3)	(r,0)	$\{ AL_2^3 \}$
U_3	(b,0)	(r,3)	(w,0)	$\{ AL_3^2 \}$

$\Rightarrow$ Time t_6: AL_3^3 arrived

	V_1	V_2	V_3	WT_i
U_2	(b,0)	(r,3)	(r,0)	$\{ AL_2^3 \}$
U_3	(b,0)	(r,3)	(r,3)	$\{ AL_3^2, AL_3^3 \}$

$\Rightarrow$ Time t_7: Rows 2,3 applied

	V_1	V_2	V_3	WT_i
U_2	(b,0)	(g,3)	(g,2)	ϕ
U_3	(b,0)	(g,3)	(g,3)	ϕ

The information about all three updates has been received and the VUT is initialized at time t_0. At time t_1, AL_1^2 is received. Since $VUT[1,1].color = white$, $ProcessRow(1)$ returns $false$. No views can be updated at this time. At time t_2, $ProcessRow(3)$ returns $false$ and no views can be updated. At time t_3, in order to evaluate $ProcessRow(2)$, the warehouse needs to evaluate $ProcessRow(1)$(Line 4 in the procedure). The latter returns $false$. At time t_4, $ProcessRow(1) = true$ so WT_1 is applied. Line 9 causes $ProcessRow(3)$ to be called but it returns $false$. Row 1 is purged from the table. At time t_6, the merge process invokes $ProcessRow(3)$ and then $ProcessRow(2)$(Line 4). The latter returns $true$. Notice that here it recursively calls $ProcessRow(3)$ again but gets a $true$ return value since row 3 has already been added to $ApplyRows$. Actions in both WT_2 and WT_3 are now applied together as a single transaction. □

5.6 Distributing the Merge Process

The merge process (*MP*) in the previous sections may become a bottleneck as the system scales up — when the number of views defined in the system increases, or when the base relation update frequency increases. In this section we propose techniques that distribute the merge computations. To determine which proposed technique is best, i.e., to determine whether the merging should be distributed and how to distribute it, is a sophisticated optimization problem. The factors that affect the decision include the number of views in the system, the update frequency and update patterns, the cost of message passing, the cost of temporarily storing the action lists, and the overhead of running a merge process. In the following discussion we give examples to show different merging alternatives.

Basically, a merge process can be split into several ones. The most straightforward way of splitting is to first partition view managers into groups such that base relations used in the views of one group are disjoint with those used in the views of other groups. Then each group of views is assigned one merge process. Example 5.6.1 illustrates the partitioning and gives a graphical representation of the partitioned system.

EXAMPLE 5.6.1 (Partitioning view managers into disjoint groups)
Consider a system with the three views in Example 5.4.2: $V_1 = R \bowtie S$, $V_2 = S \bowtie T$ and $V_3 = Q$. Actions related to updates on S need to be coordinated between VM_1 and VM_2, but actions related to Q only involve VM_3. Thus, the three view managers can be partitioned into two groups: $G_1 = \{VM_1, VM_2\}$ and $G_2 = \{VM_3\}$. Figure 5.3 shows the partitioned system using a structure called the *View Merge Tree* (*VMT*). Each node in a *VMT* represents either a view manager, a merge process, or the warehouse; each edge indicates where to forward action lists from the child node. The warehouse is the root node of the tree. In Figure 5.3, each group of view managers uses one merge process to coordinate the actions for the group.

Each *MP* in the generated *VMT* collects *AL*s from its children as if they were the only view managers in the system. Then the *MP* runs SPA or PA according to the type of underlying view managers, just as in the previous sections. Later it forwards the *AL*s to the warehouse in a warehouse transaction. □

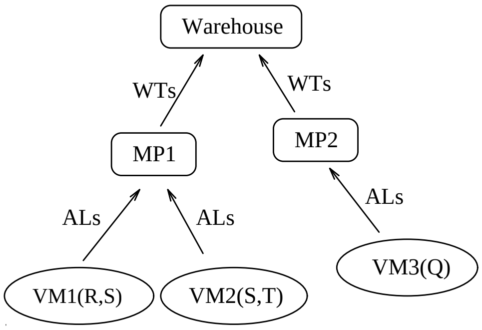


Figure 5.3: VMT after partitioning view managers into groups

One advantage of the partitioning is that action lists are coordinated only when necessary — when they are from view managers that share common base data. Another advantage is that different MPs can run different merge algorithms.

Although all actions from view managers in the same group need to be coordinated eventually, a MP may be further distributed into a hierarchy of MPs when necessary. In this scenario, a set of ALs may be first combined into one AL, then the combined AL is merged with other ALs. Figure 5.4 shows two possible distributions of the merge processes for the three views: $V_1 = R \bowtie S, V_2 = S \bowtie T$, and $V_3 = T \bowtie Q$.

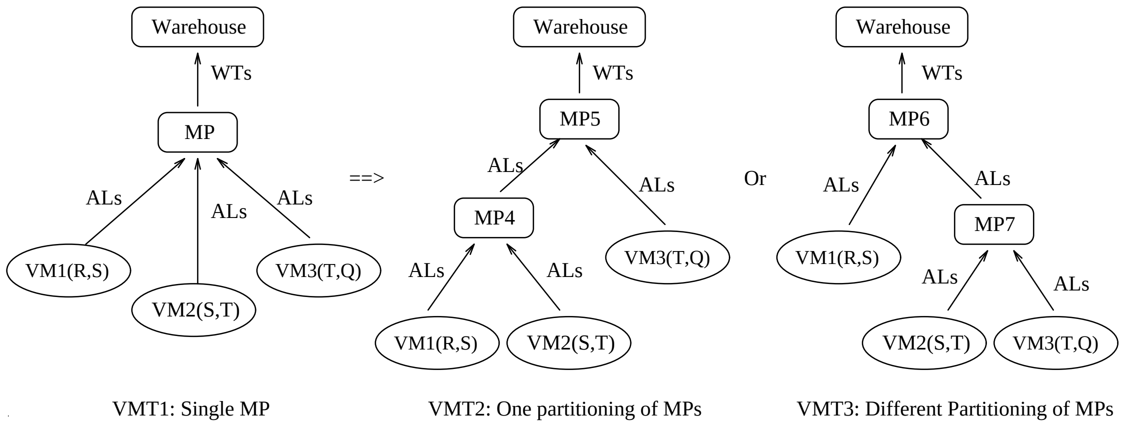


Figure 5.4: Distributing a merge process

In VMT_2 and VMT_3 in Figure 5.4, each merge process functions essentially the same as before, collecting and coordinating action lists. However, one MP node may be a child of another MP node. The child MP, for example, MP_4 in VMT_2 will modify its algorithm to output ALs instead of warehouse transactions(WTs). Since a MP may send out ALs just like a view manager, we assign each MP an unique id that distinguishes it from other view managers and MPs. In the example, MP_4 will combine all actions from VM_1 and VM_2 using SPA or PA. When it is ready to send WT_i, it will instead combine all ALs in WT_i into an action list AL_i^4 and forward AL_i^4 to MP_5. From the point of view of MP_5, MP_4 behaves as if it were an ordinary view manager "VM_4". MP_5 coordinates "VM_4" with actions from VM_3. Any MP whose direct parent is the warehouse, e.g., MP_5, still sends transactions to the warehouse as before.

When distributing the merging, it is important to find a good VMT for the system. First, this VMT should be correct, meaning that if two views V_i and V_j share a base relation, then ALs from VM_i and VM_j must be merged by some MP before being delivered to the warehouse. Then, a chosen VMT should yield good performance. The three VMTs in Figure 5.4 may illustrate the variation in system performance when using different VMTs. If the number of views is small and the traffic caused by passing ALs is low, a "flat" VMT such as VMT_1 works efficiently. In reality, a MP needs to be distributed only when the number of views in the system is so large that the MP cannot handle all actions quickly enough to keep on with the incoming flow. When the MP must be distributed e.g., if the MP in Figure 5.4 cannot handle ALs from more than two view managers, then for certain scenarios VMT_2 may be a better choice than VMT_3, and vice versa. For example, if the update frequency on relation Q is much higher than update frequencies on any other base relations, then VMT_2 is better because Q updates only need to go through one level of merging.

Finding an optimal or a good VMT is a complicated optimization problem. The first step involves deciding what comprises a "good" system: one with the minimum number of messages exchanged? one with the minimum total data transfer? or one with the minimum average time before updates are reflected in views? Next, according to the actual base relation update pattern and frequency, the load limit of a MP must be determined, i.e., what is the proper fan-in of a MP node in the VMT before it saturates. Then, heuristics may be used to pick a good VMT for the system. Since the number of all VMTs is exponential in the number of view managers, it may not be feasible to test all possibilities. Due to space constraints, we will not get into optimization details here.

Finally, a merge process is a logical concept. A *MP* can reside anywhere physically: with the warehouse, with a view manager or as a separate process. For example, it is possible to combine one merge process with each view manager, such that each *VM*/*MP* pair does two things: compute the changes to this view, and merge its own actions with actions it obtains from other view managers. Combining a merge process together with a view manager may save some communication, for example, some ALs may be directly passed to the warehouse without forwarding stop.

5.7 Summary and Algorithm Extensions

In this chapter we presented algorithms to enforce multiple view consistency. Inter-view consistency ensures that warehouse applications can correctly access data from multiple views, and is as important as consistency is in conventional database systems. As far as we know, we are the first to address the problem of efficiently enforcing consistency across views. Our SPA and PA algorithms make it possible to distribute the incremental view maintenance work over multiple concurrently-executing view managers. Our merge process can then coordinate results from different managers and guarantee data consistency. Further, we showed that the merging work can be itself distributed over a collection of merge processes. Each process can perform part of the merge in a pre-defined but flexible manner.

SPA and PA can be easily adapted to handle different assumptions from the ones we have made up to now. We now discuss two extensions of the algorithms.

5.7.1 Source Transactions and Multiple Sources

A source transaction may update more than one base relation that belongs to more than one view. As a result, even if two views do not share common base data, they may still needs to be maintained consistently. For example, if we have $V_1 = R$ and $V_2 = S$, and a source transaction inserts one tuple into R and one tuple into S, then the new tuples should appear in both views at the same time. In this scenario, MVC means that if sources have transactions (local or global) involving more than one update, then all updates in a transaction should be reflected in either all views or none.

Our definition of MVC in Section 5.2 is based on source states. In Section 4.3 we have defined source states and consistent source states for multiple sources that run transactions. In such an environment, SPA and PA are unchanged except that each occurrence of "update" in the algorithm need to be replaced by "transaction". The merge process receives information about transactions instead of updates, and one row in the *VUT* represents one transaction instead of one update. The relevant views for an update is re-defined to be the relevant view set for a source transaction: REL_i now includes all views that are affected by any update performed by transaction T_i.

5.7.2 Other Types of View Managers

Although we have focused on the most commonly used two types of view managers in our previous discussion, the framework we developed as well as the idea of algorithms can be adapted easily to handle other types of view managers. We give a few examples here:

- A view manager may do periodic refresh instead of incremental maintenance. Such a view manager will appear to the *MP* in our system as if it were an ordinary strongly consistent view manager. The action lists from this view manager will tell the warehouse to delete the entire old view and insert tuples of the new view. The merge process still coordinates the *AL*s and passes them along. If the amount of data passing from the view manager to the warehouse is large, the *MP* can be modified to coordinate transaction commit only, instead of handling all data transfer.

- A view manager may only guarantee the *convergence* of the view it manages. That is, it only guarantees the eventual correctness of the view but not the correctness of intermediate view states. Then the *MP* can just pass along all *AL*s it has received, and also guarantees the *convergence* of the warehouse views. That is, all warehouse views are consistent eventually, although some of them may go through inconsistent intermediate states.

- A view manager may be *complete-N*, that is, it may process N source updates at a time and maintain the view consistently after every N updates. In this scenario, the *MP* can use an algorithm that is similar to SPA, but instead it collects all *AL*s corresponding to every N updates, then forwards them to the warehouse. The warehouse view maintenance is *complete-N* as well.

When there is a combination of different types of view managers in the system, it is always possible to use the merge algorithm corresponding to the view manager guaranteeing the weakest level of consistency. For example, if there are both complete and strongly consistent view managers in a system, a *MP* can always use PA to guarantee strong consistency.

Chapter 6

Performance Study

Although incremental maintenance is a promising method for integrating large amounts of data efficiently, the performance of distributed incremental maintenance systems is not well understood. To study the performance as well as the implementation complexity of the algorithms we designed in the last three chapters, we implemented the algorithms in our warehouse prototype WHIPS (WareHousing Information Project at Stanford). In this chapter we describe this prototype, and present experimental results that provide insights into the incremental operation of a warehouse. In particular, the questions we address include:

- *Costs of incremental view maintenance.* What are the major components in the propagation time of an update to the warehouse? How much data is transferred from the sources and between different warehouse modules? What are the potential system bottlenecks? What system parameters have the greatest impact on performance?

- *Relative performance of different view management algorithms.* Views can be incrementally maintained using different levels of correctness and complexity. Which algorithm works best for a given environment and update scenario? How much more expensive is it to use view maintenance algorithm that is more complex yet provides a higher level of consistency, as opposed to a simpler one with a lower level of correctness? Some algorithms only work if warehouse tables contain all source keys. Do these algorithms perform better? If so, can we quantify the benefits?

- *Batching and sequencing of updates.* Updates can be batched together to reduce overhead. Delaying updates (or batches) so they are not in close proximity can also reduce some overheads. How does batch size and update separation impact performance? What is the impact on response time?

We believe that our experiences and results can provide some useful guidance to future warehouse designers and implementers. This chapter starts with the WHIPS architecture (Section 6.1), our performance criteria and relevant system parameters (Section 6.2). In Sections 6.3 through 6.6 we focus on important performance factors: modularity, keys, consistency, source update scenarios and batching.

Note that in this chapter we only present selected results and highlights. We give more extensive and detailed experimental results in a technical report [ZGM98b]. There, we also present an analytic model that roughly predicts WHIPS performance. This model helped verify the correctness of the system implementation and served as a guide for our performance experiments.

6.1 WHIPS

In this section we describe the WHIPS architecture, and we illustrate its run-time functionality by following the propagation of one source update. WHIPS handles relational Select-Project-Join(SPJ) views, as well as derived views defined by union or aggregate operators. However, this performance study focuses on base SPJ views only.

6.1.1 Architecture

Figure 6.1 depicts the part of the WHIPS architecture that is relevant to our performance study. The key WHIPS feature is modularity. As shown in the figure, the system is composed of distinct modules that communicate with each other. Each module is a separate process. We implemented each module as a CORBA object, using the ILU implementation of CORBA [CJS+94].

We believe that this modular design achieves several important goals. (1) Heterogeneity: WHIPS is able to handle various types of data sources, as may exist in a real data warehousing environment. Any database system can be used as the data warehouse. There are also a variety of view maintenance algorithms that can be used. (2) Distribution and scalability: As the load grows, maintenance work can be distributed among different processes, across multiple machines, and performed concurrently. (3) Implementation flexibility: Each integration module can be implemented and optimized independently. However, as we will see in Section 6.3, this design also bought us performance penalties.

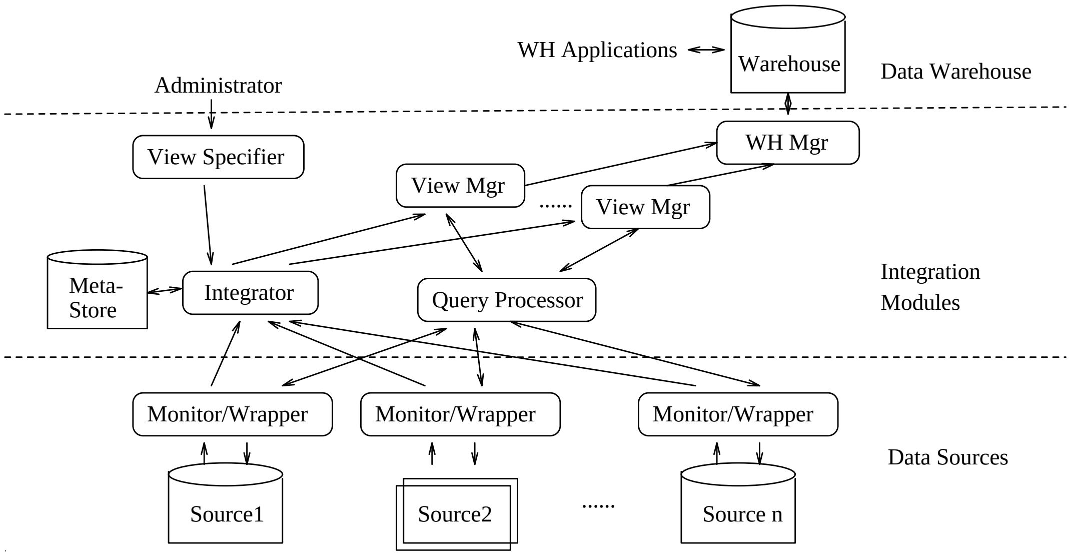


Figure 6.1: The WHIPS system architecture

In general, WHIPS is divided into three layers: the data sources, the integration modules, and the data warehouse. Each data source is encapsulated by a source-specific monitor and wrapper. A *monitor* is responsible for detecting modifications to the source data and notifying the integrator. We have implemented trigger-based monitors for cooperative data sources and snapshot monitors for other sources [LGM96]. A *source wrapper* translates single source queries from the internal WHIPS representation into queries in the native source language. Thus, the wrapper hides the source-specific querying details from all other modules. Data sources in WHIPS are autonomous; the only requirement is that they be capable (through their wrappers) of answering simple select queries. Currently, WHIPS wrappers handle three types of data sources: flat files, Sybase databases, and Mini SQL databases. New types of data sources can be added to the system as long as wrappers and monitors are written for them.

The *integrator* coordinates the different WHIPS components. At initialization, the integrator locates the warehouse manager and creates one or more query processors. When a new source becomes available, the integrator receives a notification from the source's monitor and wrapper. The integrator then stores the source information (such as the source type, schema, query interface) in the *meta-store*, a persistent store for system information. In the current prototype, the warehouse applications may define their views using a subset of SQL extended with constructs for specifying the desired view consistency (e.g., "**with strong consistency**"). When a view is defined, the *view specifier* parses it and sends the corresponding view information to the integrator. The *integrator* then creates one *view manager* to maintain this view at the desired level of consistency. After initialization, the integrator coordinates view maintenance by routing source updates to the appropriate view managers.

Each *view manager* is responsible for maintaining a view, using one of the algorithms presented in Chapter 4. For example, the Strobe algorithm is implemented in a Strobe view manager that maintains a view requiring strong consistency. The view manager first initializes the corresponding view, then receives relevant updates and computes the corresponding view "deltas" (by interacting with the query processor if necessary), and finally sends those view deltas to the warehouse manager for execution. Views can be added and deleted at run time, and view maintenance for the different views can be performed concurrently, possibly using different maintenance algorithms.

The *query processor* basically implements the procedure *source_evaluate*() described in Section 4.4. It receives global queries from the view managers and answers them by posing the appropriate single-source queries to the source wrappers. It then passes the composite global query answers back to the view managers. In our current implementation, a query processor performs distributed query processing using standard techniques such as sideways information passing and filtering of selection conditions to prune the queries it poses to the wrappers [OV91]. There could be more than one query processor if necessary.

The *warehouse manager* receives deltas (lists of the specific inserts and deletes that must be performed to bring the materialized views up to date), and applies them to the views. All modifications received by the warehouse manager in a single message are applied to the warehouse in one transaction, as needed by the view consistency algorithms. Like the source wrapper, the warehouse manager also translates the view deltas to the specific syntax of the warehouse database, thus shielding all other WHIPS modules from the particulars of the warehouse and allowing any database to be used.

The implementation of the view maintenance algorithms is straightforward given the pseudo-code we have presented. We use a *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 relations substituted by known 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.

Although a merge process running both the Simple Painting Algorithm and the Painting Algorithm (Chapter 5) is also implemented in WHIPS to ensure MVC, the performance study presented in this chapter only covers maintenance for single views. Therefore, the merge process was bypassed during the performance study and is not shown in Figure 6.1.

6.1.2 Propagating One Source Update

We now walk through an update propagation to illustrate the WHIPS components at work. Consider the base relations and views defined in Example 1.2.1. Source S

maintains subscription information $S = Subscription(Name, Magazine)$, source P maintains promotion information $P = Promotion(Magazine, Gift)$. The warehouse contains a view defined by $V = \Pi_{Name,Gift}(S \bowtie P)$. The job of the WHIPS integration components is to compute and apply $\triangle V$ corresponding to source updates. In general, each propagation follows the message paths depicted in Figure 6.2. An asynchronous ILU procedure call is used to send each message. Note that in WHIPS, updates may also be propagated in batches (see Section 6.6) that follow the same route.

Say the initial value of the base relations and the view is as shown in Figure 6.2. Let there be a source insertion $U_s = insert(S, [Tom, \underline{Time}])$, which indicates that Tom has subscribed to Time magazine. Update U_s is propagated as follows, where each point of time indicates the system starting or finishing an action.

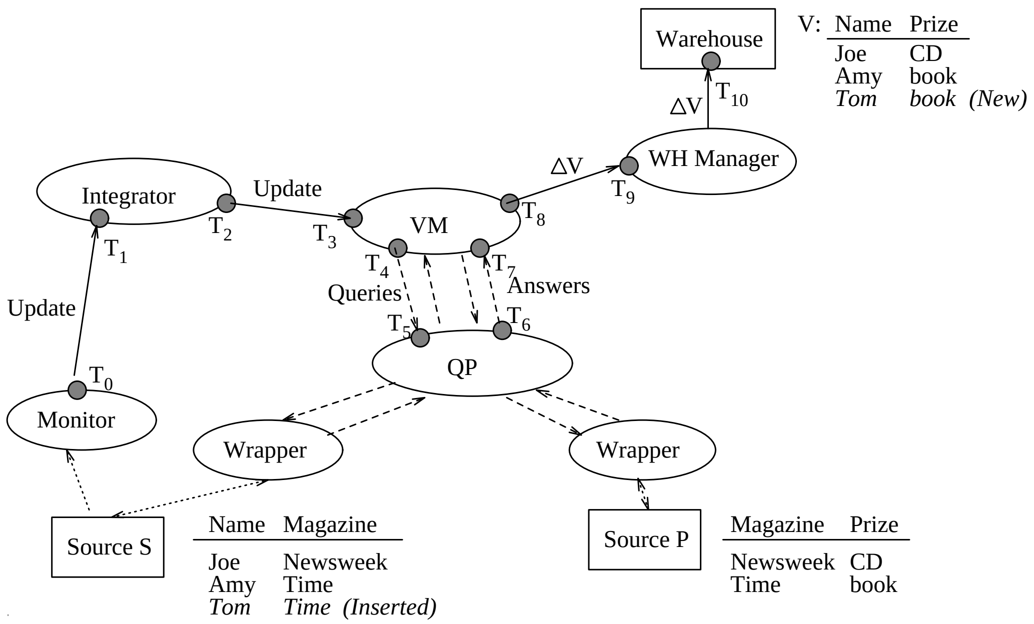


Figure 6.2: Propagating one source update in WHIPS

T_0: Update $U_s = insert(S, [Tom, \underline{Time}])$ is ready at the monitor of source S;

T_1: The integrator receives U_s;

T_2: The integrator forwards U_s to the view manager (VM) for view V;

T_3: This VM receives U_s;

T_4: The VM sends a query $Q_s = \Pi_{Name,Gift}(U_s \bowtie P)$ to the query processor (QP). We call this query an *update query* because it is triggered by a source update;

T_5: The QP receives query Q_s;

T_6: The QP finishes processing Q_s and sends the answer $A_s = [Tom, \mathit{book}]$ to the VM. Between T_5 and T_6, the QP sends a *wrapper query* to the source P wrapper asking for all gifts offered by Time magazine. The QP then composes answer A_s from the wrapper query answer;

T_7: The VM receives answer A_s;

T_8: The VM sends delta view $\triangle V$ to the warehouse manager. Note that $\triangle V$ contains the tuple $[Tom, \mathit{book}]$ obtained from A_s with an indication that it is to be inserted into the view;

T_9: The warehouse manager receives $\triangle V$;

T_{10}: $\triangle V$ appears in the view. That is, the transaction that applies $\triangle V$ to V commits, and the view contains the new tuple $[Tom, \mathit{book}]$.

Not all update propagations go through each of the above steps. For example, some view managers do not send queries to the QP when processing deletions. We let all view deltas, including empty ones, reach the warehouse manager, because this is needed to ensure multiple view consistency. (See Chapter 5. Performance of MVC is not studied here.) Because of compensating queries, there may be more than one pair of queries and answers between the VM and the QP (between T_4 and T_7). In this case, we assume T_4 is when the VM sends the first update query and T_7 is when the VM receives the last answer.

6.2 Performance Criteria and System Setup

In this section we first outline the major performance criteria, then describe the basic setup of our performance experiments. We also present some basic timing results.

6.2.1 Performance Criteria

The main criteria we use is *update propagation time*(UPT). We define the UPT for a specific source update and view as the time elapsed from when the integrator receives the update (T_1 in Figure 6.2) to when the warehouse manager receives the corresponding view deltas (T_9). The time is measured using the Unix function `gettimeofday()`. Since in our experiments all the integration modules are located on the same machine, the UPT is just the time difference between T_9 and T_1. To refer to sub-components of the UPT, we use $t_{i,j}$ to represent the elapsed time between T_i and T_j in Figure 6.2. For example, $t_{5,6}$ is the time between the QP receiving a query Q and sending the corresponding answer, and $t_{6,7}$ is the time between the QP sending the answer and the VM receiving that answer. The full UPT of an update is $t_{1,9}$. A longer time to propagate an update to the warehouse indicates more stale views and an increased possibility of interference between consecutive update propagations.

We did not include in the UPT the time before the update reaches the integrator (before T_1) because $t_{0,1}$ is specific to the connection between the monitor and the integrator. Similarly, we did not include the time to apply $\triangle V$ to the warehouse (after T_9) because $t_{9,10}$ is specific to the database used as the warehouse. Neither of these times is determined by our integration techniques.

For some experiments, we study the system *throughput* — the number of updates processed in one unit of time (one second). For a group of B updates, if T is the elapsed time from when the first update starts at stop 1 to when the last update reaches stop 9, then the system throughput is B/T. Another performance criterion is the number of messages passed between modules during the propagation of one or more updates. The most interesting numbers of messages are the number of queries sent by the view manager to the query processor and the number of source wrapper queries. These two numbers various for different views, for different types of view managers, and they affect UPT substantially.

In general, UPT and throughput are indications of the system efficiency, and the number and size of messages are indications of system load. Other performance criteria (which we do not discuss), such as the initialization time or the network traffic, are often related to the UPT, throughput, and the number and size of messages.

6.2.2 System

For all experiments presented here, both the warehouse views and the source relations are stored in a Sybase database installed on an IBM RS/6000 machine. The WHIPS integration modules run on a DEC Alpha 3000/300LX under OSF/1 3.0 with 125MH CPU, 64MB memory, and 128MB swap space. No other user processes were run on the system during the experiments. Source wrappers are on separate DEC Alpha machines with the same configuration. Any two machines are connected by 10BaseT Ethernet. As we have mentioned, most of the WHIPS system is implemented using C++ and ILU, a CORBA compliant object system developed by Xerox PARC. The entire system is about 19,000 lines of C++ code.

6.2.3 Views and Base Relations

View parameters affect the update propagation include the number of base relations, the sizes of base relations, selectivity (the fraction of tuples that satisfies the select condition(s)), project factor (the size of all projected attributes, which determines the tuple size of the view), and the join factor (the number of tuples in a relation that join with one tuple in a neighboring relation). Given this diversity of factors, it is not easy to obtain a few "representative" scenarios. However, after some experimentation we chose the following "reasonable" views, representing views with different numbers of base relations and different result sizes.

$$
\begin{aligned}
V_1 &= \Pi_{A,B}(R),\\
V_2 &= \Pi_{R.A,R.B,S.B,S.C}(R \bowtie S),\\
V_3 &= \Pi_{R.A,R.B,S.B,S.C,T.C,T.D}(R \bowtie S \bowtie T),\\
V_4 &= \Pi_{R.A,R.B,S.B,S.C,T.C,T.D,U.D,U.E}(R \bowtie S \bowtie T \bowtie U)
\end{aligned}
$$

In the view definitions, R, S, T, U are base relations. In the default setup, the cardinality of the base relations is set to 100 tuples. Each tuple in a relation is 100 bytes, and each projection attribute is 10 bytes. By default, we populate the tables such that one tuple of R joins with exactly one tuple of S, then joins with exactly one tuple of T. This is also true for new tuples; that is, if a new tuple is inserted into

R, and this tuple is to be joined with S to compute $\triangle V$, we set the attribute value such that this new tuple will join with one tuple in S. This fixed join factor reduces variability and makes our results easier to interpret.

We use small relations mainly because there is an ILU upper bound on message size, which limits the sizes of the initial views, as well as of the query answers. To get around these limits we could have ported WHIPS to another CORBA implementation (e.g., Orbix has higher limits on message sizes). We could have also implemented a facility to break up large messages into "packets" to be sent in pipelined fashion. We did not consider this effort worthwhile for our evaluation because most of the performance results of interest are not affected by the base relation size. In particular, since we are studying the performance for *incremental* maintenance, the most important factors in computing $\triangle V$ are the join factors between relations. Also, we are focusing on the performance of WHIPS integration components. Larger base relations may only increase the time to evaluate a source wrapper query if the relation is not properly indexed. Finally, we often want to compare the relative cost of different algorithms or design choices. For example, in Section 6.4.2 we are interested in comparing complete consistency strategies with strong consistency and no consistency. Absolute timings are not very relevant to this comparison.

6.2.4 Update scenarios

In our performance experiments we use an *update simulator* to generate source updates according to specified parameters. The simulator updates the source data and sends the updates to the integrator. Thus, the simulator behaves like a WHIPS monitor. We consider two types of update propagation scenarios: non-interleaving (default scenario), in which the updates are separated by sufficiently long intervals so that the propagation of consecutive updates do not interfere with each other, and interleaving, when there is some update interference. This interference may cause resource conflicts as well as additional compensation work. Thus, the UPT of an update with interference may be longer than in the non-interleaving scenario.

6.2.5 Confidence and Some Basic Timing Results

To measure propagation times we use the average time $\bar{T} = (t_1 + t_2 + \ldots + t_P)/P$ from P repetitive experiments. In other words, each data point in the graphs that follow is an average number obtained from P repetitive experiments. For all the experiments, we have chosen a large enough P so that with a degree of confidence 0.99 we guarantee that the expected value of T is within $\bar{T} \pm 0.1\bar{T}$. Details on our choice of P and the evaluation of confidence intervals are given in [ZGM98b].

There are two basic timing results that are determined by the specific system configuration and are not affected by the view maintenance algorithms. The first one is the time to pass an asynchronous message between two modules. With the described configuration, a message containing 100 bytes of data takes about 0.004 seconds if the two modules are located on the same machine, and about 0.005 seconds if they are located on different machines. The second factor is the time to evaluate a source query, including the time to communicate with the source wrapper, execute the query, and get the results. When the result size is one tuple with 100 bytes of data, the time to execute a selection wrapper query with one selection condition takes about 0.028 seconds. This number remains roughly constant as the base relation ranges from 1 to 1000 tuples. We study the effect of message size on message time further in [ZGM98b]

6.3 Cost of Modularity

WHIPS was designed as a modular system to make experimentation easier. Even though modularity is a requirement, it is important to study the cost of this modularity, to help us interpret the results that follow, and to help us understand the relative costs of the various update propagation stages.

Recall that we defined the UPT of an update as $t_{1,9} = \sum_{j=1}^{j=8} t_{j,j+1}$. Among those times, $t_{2,3}$, $t_{4,5}$, $t_{6,7}$ and $t_{8,9}$ are communication times between WHIPS integration modules. These times are mainly determined by the size of the message. For example, $t_{2,3}$ is the time that the integrator takes to forward a source update to the view manager. It is roughly the same for all single-tuple updates, and it increases with the batch size if source updates are combined. Similarly, $t_{6,7}$ is the time the query processor takes to send a query answer to the view manager, and it is mainly determined by the answer size. Let $t_{commu} = t_{2,3} + t_{4,5} + t_{6,7} + t_{8,9}$, the communication overhead entailed by WHIPS's modular design.

Let $t_{computation} = t_{1,2} + t_{3,4} + t_{7,8}$. During these periods the WHIPS modules process data, e.g., decide the type of an update or check the correctness of a query answer. Let $t_{query} = t_{5,6}$, the time to process a distributed query. The value t_{query} may involve both communication (with source wrappers) and computation (collecting results). The length of t_{query} is determined by the number of wrapper queries, the network speed, and the time it takes sources to execute queries.

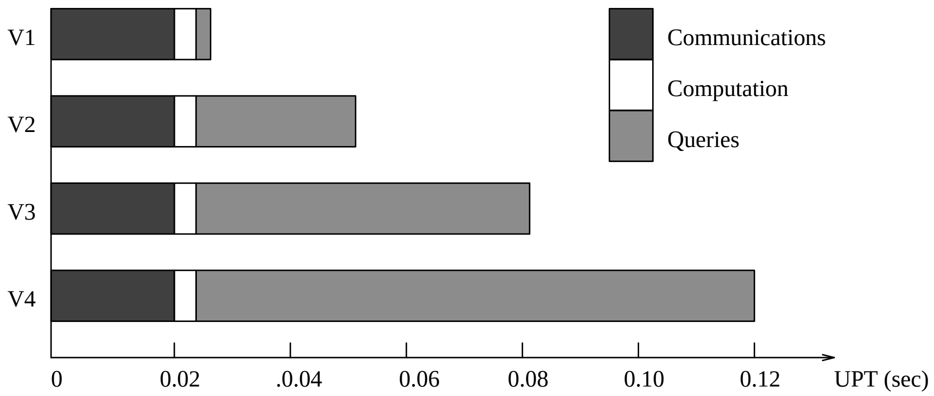


Figure 6.3: Update propagation time breakdown for different views

In Figure 6.3 we show the UPT breakdown for propagating a source insertion in the default setup. The results we present here are obtained using one type of view manager but results for other view managers are almost identical (see Section 6.4.2). In the non-interleaving scenario, the UPTs for the four views given in Section 6.2.3 are 0.025 seconds, 0.053 seconds, 0.083 seconds and 0.120 seconds. As expected, for different views, the t_{commu} times are almost the same, because messages with the same type and almost the same size are exchanged. The $t_{computation}$ times are also similar for all views, and they are shorter than the communication and query times. For V_1, which contains only one relation, no wrapper query is necessary to propagate source updates. Therefore, communication time dominates, and t_{commu} is about 85%

of the total UPT. As the number of base relations in the view increases, the query time quickly increases and becomes the dominant factor. For V_4, t_{commu} is about 17% of the UPT and t_{query} is about 80%. While the absolute communication time remains essentially constant, its fraction of the entire process decreases.

Figure 6.4 shows the update propagation time breakdown for different answer sizes of update queries. The results are for V_3 using algorithm MECA. As expected, in general the average UPT increases with the size of answer relation. This is due to the increase of both the communication time (shown as the shaded area in Figure 6.4) and query time, both of which include transferring the answer relation between modules. For V_3, the inter-module communication time is about 1/3 to 1/2 of the entire UPT. Notice that message transmission time does not increase linearly with message size. For example, there is a significant jump in message times when the message size increases from 20 to 30 tuples. We believe this jump is caused by ILU changing its strategy for packaging messages.

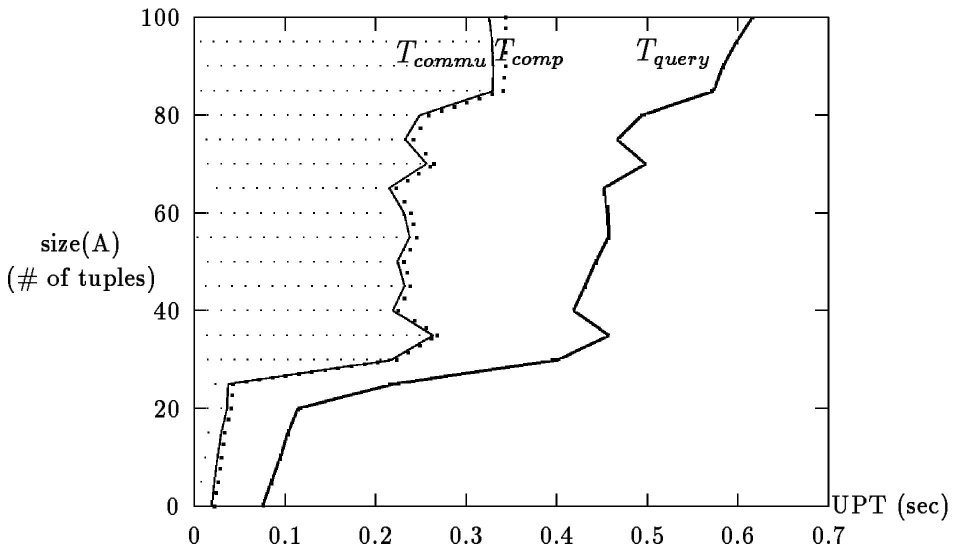


Figure 6.4: Update propagation time breakdown for different answer sizes

There are two important features in Figures 6.3 and 6.4: query times can be very significant for some views, and the communication times are a significant overhead. Let us discuss these in turn.

Query times are high when the sources have to be contacted as part of the maintenance process. These times can be reduced by having faster sources, or by improving the query processor (e.g., through smart join ordering or condition filtering). Query times can be reduced even more by copying all source relations to the warehouse, avoiding going back to the sources entirely. (WHIPS can maintain copies of source data by defining views that are simply the same as source data.) In this case, queries are still performed, but entirely with warehouse data. The benefits of copying source data must be balanced with the costs: increased storage space and a larger number of updates that must be processed. The latter is because the copied source data must be maintained at the warehouse. The results in Figures 6.3 and 6.4 quantify some of the costs and potential savings involved, and hence can be useful for determining if it is worthwhile building a better query processor or copying more source data at the warehouse.

The communication costs can be reduced by using a more efficient CORBA implementation, or by coding the message passing directly. Another option is to combine all the integration modules, including the warehouse manager, into the same process, i.e., put all the integration modules close to the warehouse. With this approach, different integration modules may share a memory structure to exchange information. We believe that using separate modules is better for a testbed such as WHIPS, where ease of implementation is important, and that a combined approach would be more effective for a high performance system. This choice between tightly and lose integration of components is one faced by many system designers.

An option for reducing communication overhead while still maintaining modularity is to give each view manager its own query processor. Thus, each view manager and its instance of the query processor would share a process, and data exchanges between them would be efficient. This approach is especially beneficial when there are many queries from the view manager to the query processor, e.g., when updates frequently interleave and the view manager sends many compensating queries. The main disadvantage is that it limits our ability to do load balancing for query processing.

In summary, if the efficiency of update propagation is critical, then a modular design based on CORBA is probably not be the best choice. If it is feasible to copy all source data to the warehouse, this can significantly reduce query times and should be considered. Of course, the types of views being maintained play an important role in these decisions, and the results shown in this section may help quantify the costs and benefits.

6.4 View Maintenance Algorithms

WHIPS implements six view maintenance algorithms, divided into two groups, as shown in Table 6.1. *Key algorithms* assume that the projection list in the view definition contains all key attributes from the base relations. *Duplicate algorithms*, including compensating algorithms, can maintain all Select-Project-Join views, with or without key information. WHIPS has available three view maintenance algorithms for each group, with different consistency levels. In the table, MCCA is the multi-source version of algorithm CCA (Section 3.5).

	Key Algorithms	Duplicate Algorithms
Inconsistent	KeyConventional	Conventional
Strongly Consistent	Strobe	MECA
Complete	CStrobe	MCCA

Table 6.1: Consistency levels of view maintenance algorithms

In Section 6.4.1 we compare the performance of the two groups of algorithms. Then in Section 6.4.2 we compare algorithms that guarantee different levels of view consistency. Finally we discuss the coding complexity of all algorithms implemented.

6.4.1 Keeping Keys in a View

First we compare the performance of key and duplicate algorithms. We specifically compare key algorithm Strobe with duplicate algorithm MECA. In the non-interleaving scenario, the other two key algorithms behave the same as Strobe, and the other two duplicate algorithms behave the same as MECA (see Section 6.4.2).

Figure 6.5 shows the UPT of an update on base relation R.

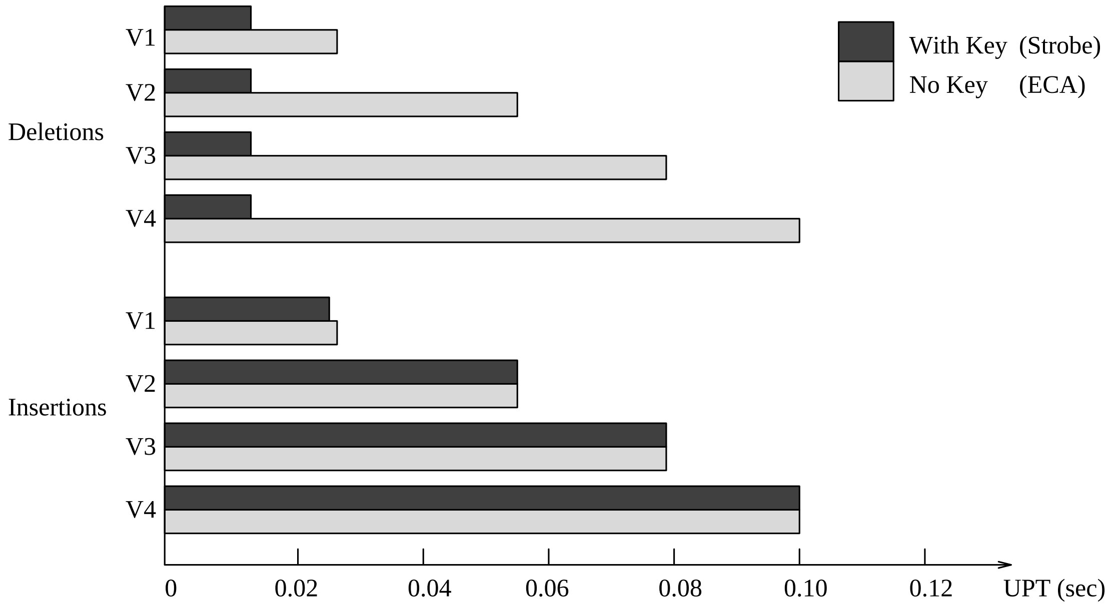


Figure 6.5: Update propagation time for algorithms with and without key information

As we see from the results, for the same view, the UPTs for insertions are similar for both algorithms. This is because they both need to send source queries and obtain answers. The UPT using Strobe is sometimes slightly shorter because Strobe uses a simpler data structure. For deletions, the UPTs are quite different. Duplicate algorithm MECA treats deletions exactly the same as insertions. Strobe, on the other hand, always forwards the deleted key attributes directly to the warehouse and lets the warehouse manager apply a delete action. Therefore for Strobe, the UPTs of all deletions are the same, regardless of the view definition, and they are shorter than those for insertions. (In the rest of this performance study, we focus mainly on duplicate view managers under insertions. This is because the UPT for using other view managers can be inferred from these results.)

Obviously, the benefit of using a key algorithm is the efficient processing of deletions. The higher percentage of deletions in the source update streams, or the longer it takes to propagate insertions (e.g., when the view contains many base relations), the more beneficial it is to use a key algorithm.

When source updates include deletions, maintaining a view using key algorithms also incurs fewer messages between the VM and the QP, and thus generates fewer wrapper queries and less work at the source. To propagate a total of K updates including K_1 insertions, there are K update queries sent from the VM to the QP for duplicate algorithms and K_1 for key algorithms.

It is important to note that the two groups of algorithms maintain views using different operations. Key algorithms maintain their views by applying *actions* to them (recall Section 4.4.2). An *action* either inserts tuples or deletes tuples that satisfy certain conditions. For example, an action can be "Delete from view V_2 where $A = 2$." On the other hand, duplicate algorithms maintain their views by applying *plus deltas* and *minus deltas* to them (recall Section 3.3.2). That is, before maintaining a view, the tuples to be inserted into the view are stored in a temporary "plus table," and the tuples to be deleted from the view are stored in a "minus table." When the view is ready to be maintained, tuples in the plus table are inserted into the view and tuples in the minus table are deleted from the view, using duplicate operations.

Whether it is more efficient to use actions or delta tables depends largely on whether the underlying database supports duplicate semantics, and on how many actions/deltas can be combined and applied to the view in a batch. For example, performing a delete action for which the key attribute value is given may be very efficient (e.g., given an index over that key). On the other hand, deleting one tuple from a view containing duplicates may require specialized procedures when such a deletion is not supported by the underlying database (e.g., Sybase and most other databases). If there is a list of actions to be applied, they have to be applied in the order given, while a list of delta tables may be combined into a single pair of plus and minus tables and applied to the view at once.

6.4.2 View Consistency

Now we study the cost and complexity of achieving different levels of view consistency. We performed experiments under both the non-interleaving scenario and a predictable interleaving scenario. The average UPT results are shown in Figure 6.6.

When there is no source update interleaving, all three types of algorithms maintain

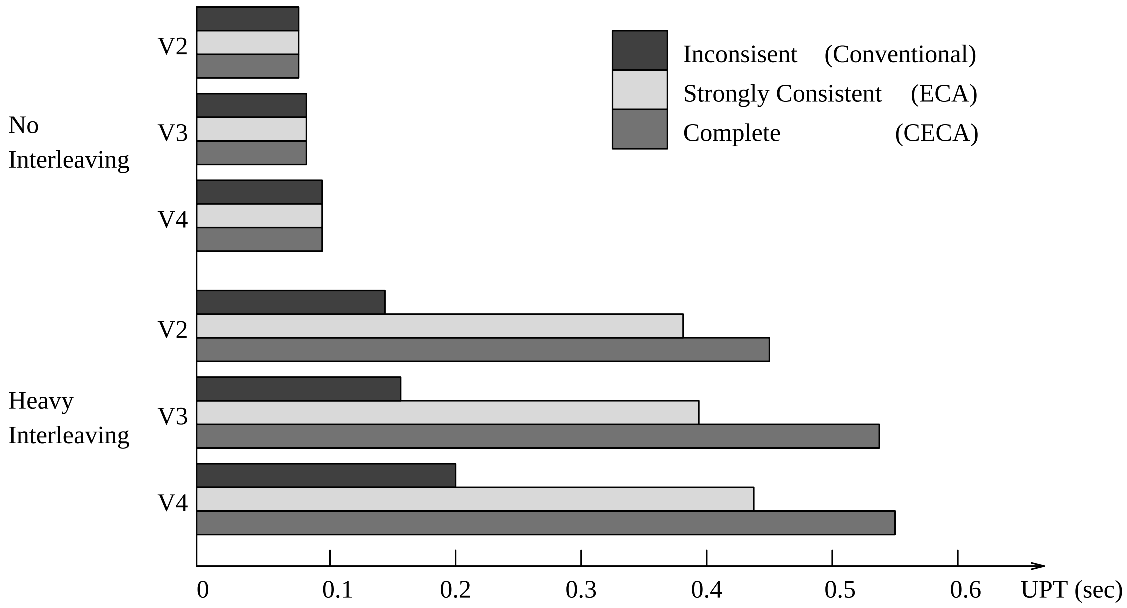


Figure 6.6: Update propagation time for views maintained to different consistency levels

materialized views correctly. The average UPTs for the three algorithms are almost the same. This is as expected since all view managers should send and receive the same messages with the same contents. Between T_7 and T_8, the strong and complete view managers execute more CPU operations due to their more complex logic, but the effect is insignificant. In this scenario, all three algorithms also reflect each single-source update in the view.

In the interleaving scenario, two tuples are simultaneously inserted, one into base relation R and the other into S, and reported to the integrator one after the other. Let these two updates be U_r and U_s. We also make U_r join with U_s and join with one existing T tuple. Update U_s occurs before the propagation of U_r finishes, i.e., before the update query corresponding to U_r is evaluated. Thus U_s interferes with the propagation of U_r. We measure the average UPT of this group of two updates and repeat the experiment enough times to satisfy the specified confidence level. We stress that this interleaving scenario is a pessimistic one because every single update observes interference. In Section 6.5 we study a scenario with stochastic interference.

For the interleaving scenario, the Conventional algorithm may lead to incorrect view maintenance. The UPT for the Conventional algorithm is a lower bound since that algorithm logically processes the updates as if there were no conflicts among them. The UPT increase for the Conventional algorithm between the no-interleaving and interleaving scenarios is due to resource contention; we discuss this further in the following section.

The strong and complete algorithms will send compensating queries to correct potential inconsistencies. Specifically, a compensating query of the form $U_r \bowtie U_s \bowtie T$ is sent to the QP for the two updates U_r and U_s. Since evaluating compensating queries requires communication between the VM and QP, and possibly between the QP and source wrappers, the impact of compensating queries on UPT is substantial. For example, for V_3, the UPT using the MECA algorithm is about 2.6 times longer than the Conventional UPT, and that using MCCA algorithm is about 3.0 times longer than the Conventional UPT. Complete algorithms take the longest time to propagate updates because updates must be processed one by one, while strongly consistent algorithms may combine updates. The increase of the average UPT for complete and strongly consistent view managers is the cost of maintaining data consistency.

Now we look at the number of messages exchanged. Let the total number of updates be K, where K is an even number. Let the number of messages exchanged by modules m_1 and m_2 be $msg(m_1, m_2)$. For the non-interleaving scenario, we have $msg(VM, QP) = K$, $msg(QP, WR) = 2K$ (WR represents all source wrappers) and $msg(VM, WH) = K$ for all three algorithms. For the interleaving scenario and the two compensating algorithms, every pair of updates triggers two update queries and one compensating query, thus $msg(VM, QP) = 3K/2$. Each update query invokes two source wrapper queries, and each compensating query invokes one wrapper query, so the total number of wrapper queries is $msg(QP, WR) = 2 * K + 1 * K/2 = 5K/2$. For the MECA view manager, every two updates are bundled together and invoke only one view-update message, so $msg(VM, WH) = K/2$. For the MCCA view manager, each update invokes one view update message, so $msg(VM, WH) = K$. The number of messages is summarized in the following table.

As we can see, the increase in number of messages is not dramatic. The total

	$msg(VM, QP)$	$msg(QP, WR)$	$msg(VM, WH)$
All three algorithms (non-interleaving) and Conventional (interleaving)	K	2K	K
MECA (interleaving)	3K/2	5K/2	K/2
MCCA (interleaving)	3K/2	5K/2	K

Table 6.2: Number of messages for different view managers and interleaving scenarios

message overhead goes from $4K$ in the non-interleaved case to at most $5K$ with interleaving. Thus, the main costs of guaranteeing consistency when updates interleave are delays and the overhead of additional source queries.

6.4.3 Implementation Complexity

Table 6.4.3 shows the number of lines of code for the view manager algorithms, excluding comments and common code (e.g., initialization). As we can see, the key algorithms are slightly less complex to implement; for example, the implementation of the Strobe view manager is about 70 lines of code shorter than the MECA view manager. Within the same group, the two complete view managers are the most complex, as expected, followed by the strongly consistent view managers, followed by the Conventional view managers.

View Managers	Lines of Code	View Managers	Lines of Code
vmCStrobe.cc	468	vmMCCA.cc	479
vmStrobe.cc	311	vmMECA.cc	381
vmKeyConv.cc	201	vmConv.cc	190

Table 6.3: Coding complexity of different view managers

6.5 Source Update Scenarios

Interleaving updates impact the system in two ways:

1. Resource contention. A message for one update may be blocked if the module it tries to reach is busy processing another update.

2. Generation of extra work. Some of the algorithms, such as MECA or MCCA, compensate for interleaving by sending compensating queries and processing the results of those queries. This creates additional work.

In Section 6.4.2 we focused on the delays caused by the extra work generated by compensating algorithms. In this section we first study the effect of resource contention on UPT and system throughput. We then discuss one experiment where both contention and extra work affect system performance.

6.5.1 Fixed Update Intervals

For each test in this experiment, a sequence of F updates on the same base relation are sent to the integrator, separated by a fixed interval I. The average UPT and the system throughput are then measured. For different tests, a different time interval is used to separate updates. Because all the updates are on the same base relation, there is little extra work generated by the algorithms, and most of the delays are caused by contention, i.e., queuing of messages. Figure 6.7 shows the average UPT and throughput for view V_3 using MCCA algorithm for $F = 10$ and $F = 20$.

As expected, when the update frequency is the highest (update interval I is 0), we have the longest average UPT due to update interference. We also have the highest throughput, meaning that the system is kept busy and tries to process as many updates as possible. Therefore, to maximize system throughput, it is better to propagate updates as fast as they become available, regardless of the delay caused by propagation interference. The average UPT decreases as the update interval grows, and once the time interval is greater than a certain threshold, the average UPT stays almost constant. Let this time interval be I_b. Intuitively, the propagations of consecutive updates do not interfere with each other if they are separated by a time

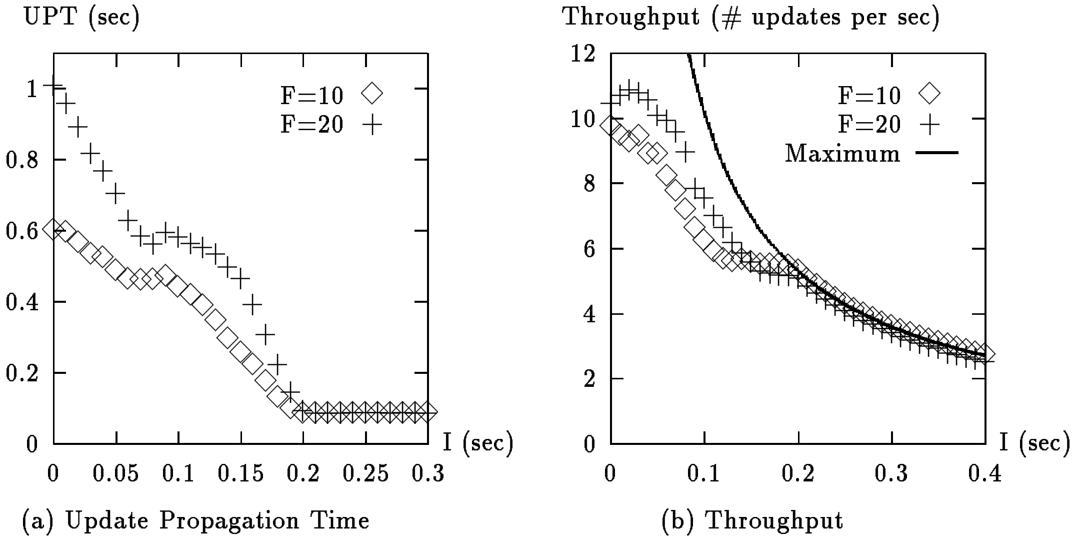


Figure 6.7: UPT and throughput for updates with different update intervals

interval larger than I_b. Let the UPT with I_b intervals be T_b. For view V_3, we have $I_b \approx 0.2$ seconds and $T_b \approx 0.08$ seconds.

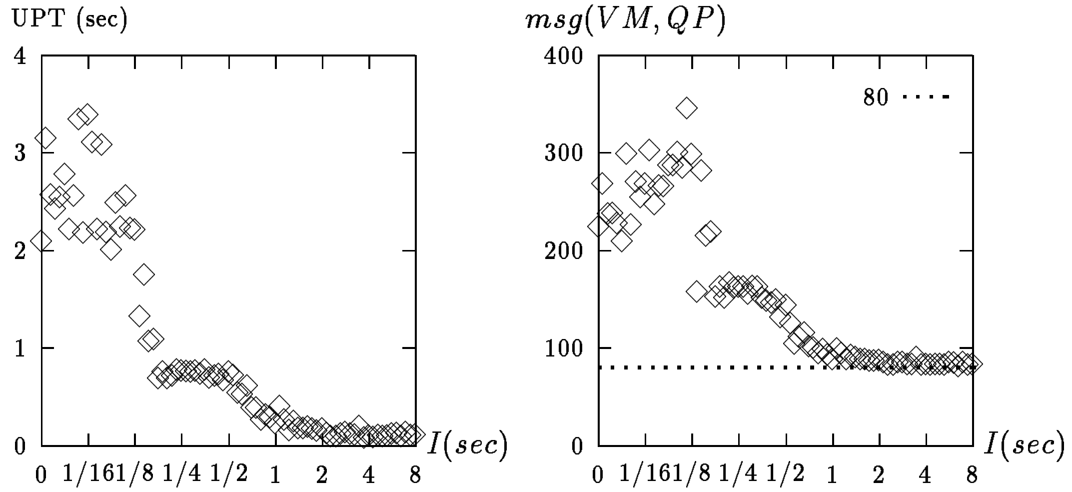
In Figure 6.7(b), the curve "Maximum" shows the maximum possible throughput for updates sent in the specified intervals, which is roughly equal to $1/I$. As expected, when $I > I_b$, all updates are propagated without any delay, and the throughput reaches maximum. When $I < I_b$, because there are delays due to propagation interference, the throughput is less than the maximum possible. We can see that when $I < I_b$, the throughput for $F = 20$ is greater than that for $F = 10$, but the average UPT is also longer.

Keep in mind that each data point in Figure 6.7 represents an average over many executions. Even though our inputs have a very regular pattern, the timing of each execution can vary significantly, depending on how and when messages are queued. In [ZGM98b] we provide a detailed analysis of update propagation times.

6.5.2 Random Update Intervals

So far, we have studied regular update patterns that let us understand the main tradeoffs in incremental update processing. Next we perform a series of experiments with randomly-generated source updates. These experiments provide a more realistic scenario and a better idea of how WHIPS may operate in practice.

Figure 6.8 shows performance results for view V_3. We concurrently generate 40 random insertions into base relation R and 40 random insertions into relation S. Each new tuple has a 1/5 chance to join with an existing tuple in a neighboring relation. The update interval is a random number between 0 and I seconds. Figure 6.8 shows the results for the MCCA algorithm. Each data point represents one run where the 80 updates are processed.



(a) Update Propagation Time (b) Number of queries from VM to QP

Figure 6.8: Performance results for propagations with random update intervals

Figure 6.8(a) shows the average UPT. The general shape of the resulting curve is similar to that in the fixed-interval scenario. When the update interval is relatively long, there is little interference and the UPT is roughly constant ($\approx 0.08sec$). The average UPT increases as the update frequency grows. When I is small, the UPT is quite variable, since the precise ways in which the updates interfere is very unpredictable.

Notice that the point at which updates effectively stop interfering ($I > 1$) is different from the previous experiment (where I_b was 0.2 seconds). This is partly because now updates are separated by times in the interval 0 to I (and not all separated by

I). Another reason is that in the current experiment updates are delayed both because of resource contention and because of compensating queries. To illustrate this, Figure 6.8(b) shows the number of queries sent from the VM to the QP. Of all the queries, 80 are update queries, and the rest are compensating queries. We can see that the query number increases almost directly with the increase in average UPT. There are many compensating queries when the update frequency is high; the actual number depends on how updates interfere with each other.

We also noticed that there is no significant increase in the number of actual wrapper queries (see [ZGM98b] for results). Because the chance of two updates U_r and U_s joining is small (1/200), most of the compensating queries of the form $U_r \bowtie U_s \bowtie T$ result in an empty answer and can be evaluated by the QP locally. Thus, we see a good optimization opportunity for the heavy interleaving scenario: the view manager may screen compensating queries before sending them to the QP. If a query will clearly yield an empty answer (as is the case with $U_r \bowtie U_s \bowtie T$ when U_r and U_s do not join), the view manager can skip it and reduce the compensation load.

6.6 Batching

Source updates can be batched and propagated to the warehouse together. In general, batching updates not only reduces the number of messages, but it also reduces the chance of interference between updates. However, batching may cause updates to wait before being sent, increasing their UPT. Batching can be done at various steps along the update propagation path, i.e., at the source monitor, the integrator, or the view manager. We assume that source monitors collect updates and send them together to the integrator, according to one of the following batching policies.

1. Batch every B updates, where the batch size $B \geq 1$. The $B = 1$ scenario has been discussed in previous sections, where each update is propagated individually. When $B > 1$, some updates may wait at the source monitors to be batched with later updates, and this waiting time would be considered part of the propagation time. As we will find out later in this section, for certain update scenarios, there is an optimal batch size.

2. Batch all updates generated in a certain time interval, e.g., in a minute or in an hour. This policy is useful if it is only possible to propagate updates during certain time windows, so as to not interfere with other activities (e.g., user queries). At the beginning of an update window, it is possible to break up a batch into smaller ones.

3. Batch when the "oldest" update has waited for a certain time. This method may ensure that a view does not become too "stale." It can also be used in combination with Policy 1 to make sure all updates are propagated to the warehouse after waiting for a certain time.

In this section, we will discuss only Policy 1 involving the optimal batch size. Once we have determined the optimal batch size for Policy 1, its value may be used in Policies 2 and 3. For instance, in Policy 3 we may wait to collect the optimal number of updates, unless the first update is delayed too long.

There are two major goals for batching. The first is to reduce the workload. Batching reduces the number of messages, so it is most useful when messages are expensive. Batching may also reduce the number of operations done by the integration module. For example, to compute $\triangle V = \triangle R \bowtie S$ where there are n updates in $\triangle R$, batching the n tuples will enable the S source to perform one join instead of n joins. In general, the larger the batch size, the more the workload is reduced. Of course, considering view freshness requirements and message size limitations, there is a practical limit on the largest reasonable B.

The second major goal of batching is to speed update propagation by reducing the interference among update propagations. The following experiment shows how batching reduces the average UPT compared with individual updates. It also examines the optimal batch size under certain update scenarios. We study view $V_2 = R \bowtie S$ using algorithm MCCA. A total of K updates on base relation R are grouped into batches of size B, and sent to the integrator with a fixed interval I between batches. Figure 6.9 shows the average UPT results for $K = 20$ and $B = 1, 2, 5, 10$. As an example, when $B = 2$, there are 10 batches of updates, each containing 2 tuples. These 10 batches are sent to the integrator separated by interval I.

Figure 6.9 shows that each batch size is optimal for some update frequency. In general, when the update frequency is high (I is small), updates bundled in larger batches are propagated faster. When the update frequency is low, smaller batches are better. When I is large enough such that the propagation of updates do not interleave ($I \geq I_b$), sending updates individually (B=1) is the fastest way to propagate them.

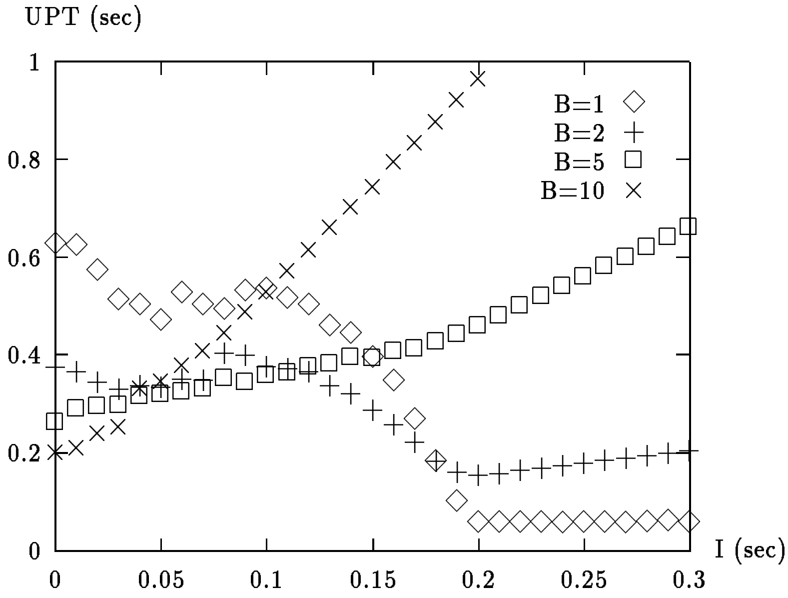
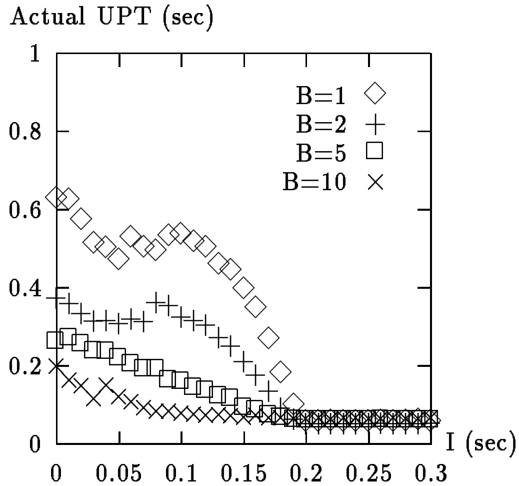
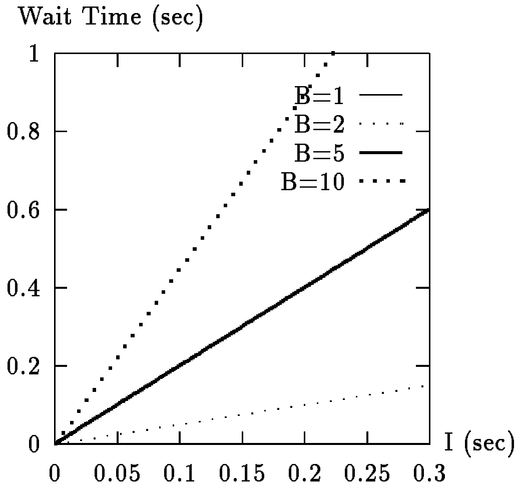


Figure 6.9: Average UPT for different batch sizes

The average UPT shown in Figure 6.9 can be broken into two components: the actual update propagation time and the wait time. (The latter is the average time updates wait at the source before being batched and sent to the integrator.) These two components are shown separately in Figure 6.10(a) and Figure 6.10(b). For the actual update propagation time only, when $I < I_b$, a larger batch size is always better, since there are fewer messages and less contention. When $I > I_b$, the actual propagation time for all B from 1 to 10 is similar. For the wait time, the source waits until B updates are available before sending them together, giving the average wait time for batch size B as $I * (1 + 2 + \ldots + (B - 1))/B = (B - 1) * I/2$. The wait time increases linearly with the update interval as well as with B. So when I is long, a larger B causes updates to wait longer.



(a) Actual update propagation time



(b) Wait time

Figure 6.10: Breakdown of average UPT for batching

When we consider both the actual propagation time and wait time, the former dominates when I is short, and a larger batch size gives shorter UPT. One special case is when $I = 0$ (all updates are ready to be propagated). This is the case when updates are generated by the same source transaction or updates accumulate during non-propagation times. Batching them in the largest possible batch size is often the best. In addition, one needs to consider the non-linear increase of message passing time pointed out in Section 6.3. Thus, it may be advisable to keep the batch size limited, so that message sizes are below those that cause a jump in transmission time. When I is long, wait time becomes the dominant factor and smaller batches may perform better. In particular, when $I \geq I_b$, sending the updates individually is the most efficient. In reality, optimal batch size or batch window may be determined according to the expected frequency of updates.

6.7 Summary

In this chapter we studied the performance of WHIPS, an incremental warehouse maintenance testbed. Performance depends on a variety of factors, including hardware parameters, the base relations and views defined, the update frequency and batch size, relation selectivities and join factors, and the maintenance algorithm used. It is thus hard to make particular performance predictions. Nevertheless, we believe that our experiments shed some light on the major design tradeoffs faced in incremental maintenance. In particular:

- We evaluated the costs of a modular system design, by studying update propagation and quantifying the cost of using CORBA as an infrastructure (Section 6.3). We concluded that CORBA is quite expensive, but offers great design and implementation flexibility.
- We evaluated the delay caused by sending queries to data sources (Section 6.3). The results can help us decide whether to copy base data at the warehouse, and how beneficial it may be to optimize global query evaluation.
- We compared the relative performance of different view maintenance algorithms and studied the cost of maintaining view consistency (Section 6.4).
- We analyzed the behavior of WHIPS under a spectrum of source update scenarios, quantifying the impact of update interleaving (Section 6.5).
- We presented results that provide insights to system designers. For example, we discussed how to determine optimal batch size (Section 6.6), how to send updates (Section 6.5, 6.6), which query processor optimizations are crucial (Section 6.3, 6.4, 6.5), and what is a superior method for evaluating compensating queries (Section 6.4, 6.5).

Chapter 7

Maintaining Graph-structured Views

Relational views are useful for controlling data access, specifying contents of caches (or remote copies), and other data management tasks. In this and the next chapter we study how to extend this view concept and the associated maintenance algorithms to what we call a *graph-structured database* (GSDB). Informally, a GSDB is a collection of "objects" that may contain "pointers" (graph edges) to other objects. A GSDB can represent Web pages, documents, XML data, data integrated from heterogeneous data sources, or other semi-structured information; it can also represent graph data structures such as a PERT chart, or a circuit design.

In this chapter, we formally define the notion of virtual and materialized views for GSDBs (Section 7.3). With these definitions, the result of a view definition on a GSDB is another GSDB, making it possible to define views on views and to query views in the same way GSDBs are queried. (This same property holds for relational views and makes them especially useful.) We present a simple language for defining graph-structured views (GS views), and mechanisms for restricting access to objects "within" a view (Section 7.3). We also discuss implementation issues related to querying and materializing graph-structured views. We present an algorithm for incrementally maintaining simple materialized views (Section 7.4). The algorithm takes as input a view definition and a sequence of updates to the base data, and propagates the changes to the copied data, querying the base data when necessary. Notice that different from the scenarios we studied in Chapters 3 to 5, this incremental maintenance algorithm assumes that it has control of the base data. That is, this algorithm is analogous to the Conventional relational algorithm. The reason that we develop such an algorithm is because there is no known incremental maintenance algorithm for graph-structured views. In Section 7.5, we briefly discuss incremental view maintenance in a warehousing environment, where the base data is stored at sources that export change information but are unaware of the view definitions. In this case, queries to the base data may be expensive and may lead to inconsistency results as in our relational setting.

7.1 Introduction

We start by giving an example of graph-structured database.

EXAMPLE 7.1.1 (A graph-structured database)
Figure 7.1 shows a graph-structured database. Each node in the graph represents an object, and each directed edge (pointer) represents a parent-child relationship between two objects. Users can traverse the graph by starting from an object and following the edges. We assume each object in the graph has an object identifier (OID) and some other fields that we will describe in Section 7.2.

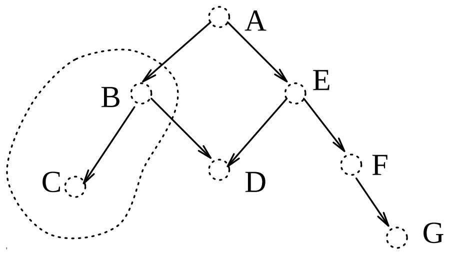


Figure 7.1: A graph-structured database □

Views on a GSDB can be used for at least two tasks: defining cache contents and query filtering. For example, consider a set of interrelated Web pages. For now, assume that each page is an object, and the URLs in pages are the graph edges. Say that a user is interested in all Web pages containing the word “flower” and would like to copy them to his local disk for faster access. Using the constructs we define in this chapter, a user will be able to define a *materialized view* to select the objects (and possibly edges) that should be copied. When the original objects change, the materialized view needs to be updated, and the incremental maintenance algorithm we discuss in Section 7.4 will keep the view up to date (at least for certain simple types of views).

A view can also be used to "filter" the objects that are accessed by a query. For example, a parent may wish to restrict access by his children to a particular subset of Web pages. For this he can define a *virtual view* (not materialized) that contains the allowed Web pages. Similarly, a user may have identified an interesting set of Web pages, and would like to use this set as a starting point for future queries. Again, a virtual view, defined using the constructs we will present, can be used for this purpose.

The data model we use in this chapter is a "lightweight" one, without notions of object classes and inheritance. In particular, in Figure 7.1, objects A, B, ... are indistinguishable as far as the system is concerned. We believe that this makes our model more generic and widely applicable. Furthermore, even if an application requires classes or inheritance, the underlying data will most likely be stored by the database system as a GSDB. For example, the system may identify the class of an object by adding a special system field or tag to each object. (It could also link together all objects in a class.) An object A that inherits from another object B could have a special system pointer to B, for instance. Thus, the same view machinery we discuss in this chapter could be used to cache or control access to the underlying data structures used to implement a more structured model.

Even though the general notion of a graph-structured view is similar to that of a relational view, there are a number of new challenges due to the richer nature of the GSDB model. For example, what exactly is a view on a GSDB? Is it just a collection of objects, or does it also include the edges between the objects? In Figure 7.1 say that our view includes the portion of the graph enclosed by the dotted line. Is the edge from B to C officially part of the view? In this case, perhaps a user that is given access to the view should not see the link from B to D. However, it is not clear how to achieve this, since the user could retrieve the contents of B which somewhere contains the C, D pointers. Saying that the view simply contains objects B and C is simpler (no need to track edges), and if the view is materialized, then one could modify the contents of B so that it no longer contains the pointer to D. However, arbitrary "editing" of the materialized objects complicates view maintenance.

In a materialized view there is also the question of object identifiers. In our example, what should the OID of the copy of B be, and how can we identify the original object? Should the materialized copy of B contain a pointer to object C, or to the copy of C in the materialized view? Whatever the choice for materializing views, there is also the problem of incrementally maintaining them as the original data is modified. This is a much harder problem than for relational views. In particular, if we modify a tuple in a relation, we know that only views defined on that relation may be impacted. In a GSDB, a simple change like adding a link between two objects can dramatically change the set of objects in a view. (This will become clearer once we discuss how views are defined.) Thus, as we will see, incremental view maintenance is more complex and more expensive for GSDBs. As a matter of fact, it may only be feasible for relatively simple views.

7.1.1 Related Work

Our work is based on previous work on materialized view maintenance [GM95], object technology [Cat94], querying object oriented databases [CD92, KKS92, KS95], and semi-structured data models [Suc97, QRUW95]. In the rest of this subsection we specifically compare our graph-structured views to relational views and object views defined using object classes.

Most of the incremental view maintenance work focuses on the relational model (See Chapter 2 for references). graph-structured views are different from relation views in at least three major ways: (1) In a graph-structured view there is no schema to constrain changes to a particular "region;" with relational views, on the other hand, changes to a relation only impact views that refer to that relation. (2) In a graph-structured view, there are relationships among objects (edges in the graph) that need to be preserved in the view. (3) View data may contain pointers and thus "lead access" to base data, and those accesses need to be controlled. Instead of defining incremental view maintenance algorithms for GSDBs, one could instead represent the graph data as relations (e.g., with a relation storing all edges), and then simply use existing relational maintenance algorithms. However, as we discuss in Section 7.4.4, directly using relational algorithms on graph data is not very effective.

Most previous research on object views (e.g., [AB91, BDK92, Run92]) uses object databases with classes. Views are defined by adding and hiding attributes to base classes. In contrast, since there is no class concept in a GSDB, views in a GSDB need to be defined by query expressions (as we do here). Views defined by adding and hiding attributes over object classes are a subset of possible views defined using path expressions [KKS92]. Also, most of the views considered by researchers so far are virtual views. In our work we consider both virtual and materialized views.

There is some recent work on materialized object views by the *MultiView* group at University of Michigan [Run92]. The major difference between their approach and ours, besides that they define views as virtual classes, is that in their system each real value (attribute value or method of an object) has only one single physical copy. A materialized view stores only OIDs (pointers) of the base value. So their view maintenance algorithms do not need to worry about duplicating or propagating values of updated objects. However, the performance advantage of querying materialized views is reduced because each access to an object in a materialized view is decomposed into several accesses to the "real data". For the same reason, their approach is difficult to adapt to a distributed environment. On the contrary, the materialized views we design in this chapter have the ability to duplicate both object pointers and values, and are able to independently maintain the relationship between an object and its duplicate.

Also related are the view maintenance methods proposed in [Suc96]. Their data model is edge-labeled trees without OIDs, which is similar to our model. They use a query language UnQL [BDHS96] to define their views, and use an algebraic approach to maintain the views. That is, they find expressions that can compute delta views corresponding to the changes of base data. However, their approach only works for views that are defined using "join free" queries, and updates that are either concatenation or replacement of two disjointed graphs. In their case, a view is always self-maintainable. In contrast, we define our views using an extension of OQL [Cat94] and designing incremental view maintenance algorithms using common insertion, deletion and update operations. Views defined in this chapter cannot generally be handled by techniques discussed in [Suc96].

Other recent work (also from Stanford) is [AGM+97, AM+98]. The difference is that views defined in this dissertation contain a set of objects only; edges between objects are implicitly kept and can be hidden only by defining multiple views. Views in [AGM+97, AM+98] may specify objects as well as edges to be kept (or discarded) in a view. Using our definition, views and query answers are ordinary databases that can be manipulated by any database operations, just like relational views and queries answers are both relations. In [AGM+97, AM+98] a view is a database, but a query answer is not. As a result, maintaining such views is quite complicated, and maintenance expressions over base data are given in [AM+98].

7.2 Data Model

In this section we introduce our object and database model. For objects, we use the OEM model [PGMW95]. Each *object* contains four fields: an OID, a label, a type and a value. The OID of an object is a universally unique identifier. The label is a string that explains the meaning of the object and does not need to be unique. Each object either has an *atomic* type, such as integer or string, or has a *set* type. The value of a set object is a set of OIDs of other objects. The following example shows a collection of objects.

EXAMPLE 7.2.1 (A collection of database objects)
We enclose each object within a pair of angle brackets and show the OID, label, type and value fields in order.[1] We use indentation as a visual aid to show the hierarchical relationship of the objects.

[1]In our examples we try to use meaningful OIDs; in general, they can be arbitrary.

```
< ROOT, person, set, {P1, P2, P3, P4} >
    < P1, professor, set, {N1, A1, S1, P3} >
        < N1, name, string, 'John' >
        < A1, age, integer, 45 >
        < S1, salary, dollar, 100,000 >
        < P3, student, set, {N3, A3, M3} >
            < N3, name, string, 'John' >
            < A3, age, integer, 20 >
            < M3, major, string, 'education' >
    < P2, professor, set, {N2, S2, P4} >
        < N2, name, string, 'Sally' >
        < ADD2, address, string, 'California' >
    < P4, secretary, set, {N4, A4} >
        < N4, name, string, 'Tom' >
        < A4, age, integer, 40 >
```

In this database, object *ROOT* (we refer to an object by its OID) is an example of a *set* object. It has four children objects, representing two professors (i.e., with those labels), one student and one secretary. Notice that the subobjects of one professor are structured differently from those of the other professor. Object $A1$ is an example of an atomic object. We use *label*(O) to refer to the label of object O and use *value*(O) to refer to O's value. For example, $label(P2) = professor$ and $value(P2) = \{N2, S2\}$.

This set of objects can also be represented by the graph in Figure 7.2. For an atomic object, we omit the type since it can be inferred by its value. For a set object, we show the OID and label within brackets, and show its value by outgoing edges. □

Next we introduce the notion of a database in this data model. A *graph-structured database(GSDB)* is an object whose set value contains the OIDs of all objects in this database. Thus, a database is simply a way to group objects together. Objects can be grouped into a database for various reasons, for example, because they are semantically related, they are frequently accessed together, or they are physically located at the same site. If all the objects in Example 7.2.1 compose a database `PERSON`, then the object `PERSON` is:

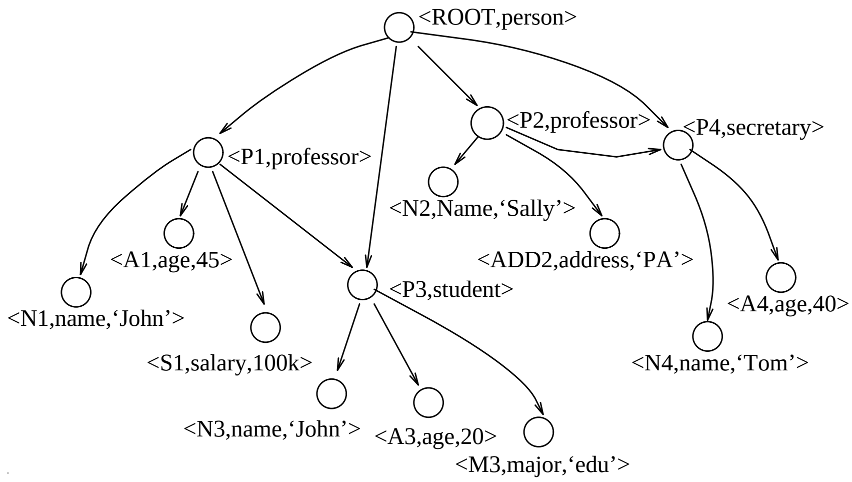


Figure 7.2: A GSDB is a collection of objects

```
< PERSON, database, set,
    {ROOT, P1, P2, P3, N1, A1, S1, N2, ADD2, N3, A3, M3, P4, N4, A4}>
```

Notice that a database is not a special type of object; it is simply a conceptual aid and will be helpful when we discuss views.

We have selected this particular data model because it is simple, and yet will let us study the important view management issues. Furthermore, we believe that the model and the algorithms we will present easily generalize to other models. For example, objects with multiple fields can be represented in our model by several objects. For instance, a multi-field employee object `< name:'Joe', salary:50k >` can be represented as

```
< E1, employee, set, {N1, S1} >
    < N1, name, string, 'Joe' >
    < S1, salary, dollars, 50k >
```

As another example, fixed format records can be represented in our model by repeating the field names (as labels) in each object. For instance, say that a schema defines the first field of a record to be a name and the second field to be a salary. Then the record `<'Joe', 50k>` can be represented by the same object above. As a third example, some graph-structured data models (e.g., [BDHS96],[QRUW95]) have values or labels on edges. These models can be mapped to ours [BDHS96].

Incidentally, note that our model is a conceptual one. The actual implementation could differ. For example, the OID field may not be stored in the record, and could be inferred from the record's location on disk. Repeated labels or values could be compressed in various ways. The database objects we defined may or may not be physical objects; for instance, objects in a database could simply be those contained in a particular region of a disk.

We now define some GSDB terminology that will be used in this chapter. A *path* is a sequence of zero or more object labels separated by dots: $p = l_1.l_2 \ldots l_n$. For example, $professor.student$ is a path. We use $N.p$ to denote the set of objects that can be reached following path p from object with OID N. If $N_2 \in N_1.p$, then N_1 is an *ancestor* of N_2, and N_2 is a *descendent* of N_1. In this case, the first label in p is the label of one of N_1's direct children, and the last label in p is the label of N_2. If $N_2 \in N_1.p_1$ and $N_3 \in N_2.p_2$, then $N_3 \in N_1.p_1.p_2$. In the Figure 7.2 example, node $A1$ is a descendent of $ROOT$ and can be reached from node $ROOT$ following path $professor.age$, that is, $A1 \in ROOT.professor.age$.[2]

A *path expression* is a regular expression of paths. For example, $*$, $professor.*$ and $professor.?$ are path expressions. A path is also a (constant) path expression. We say that a path p is an *instance* of path expression e if the wild cards in e can be substituted by paths to obtain p. For any path expression e, we define $N.e$ to be the union of all objects in $N.p$ for all instances p of e.

We allow set operations on objects of type set. In particular, let $S1$ and $S2$ be two set objects. We define $union(S1, S2)$ to be an object whose value is $\{value(S1) \cup value(S2)\}$, and define $int(S1, S2)$ to be an object whose value is $\{value(S1) \cap value(S2)\}$. We assume that these resulting objects have an arbitrary unique OID and take on the label of $S1$[3]. These operations are mainly used to manipulate database objects and query answers, although they could apply to any set objects.

Here we use a simple but representative language that lets us study view management issues. Its basic syntax is:

[2]We use the terms node and object interchangeably.

[3]Typically set operations are only meaningful when objects $S1$ and $S2$ have the same label.

```
SELECT    OBJ.sel_path_exp Y                    (7.2.1)
WHERE     cond(Y.cond_path_exp)
[WITHIN   DB1 ]
[ANS_INT  DB2 ]
```

A query answer is also an object, with format `<ANS, answer, set, value(ANS)>`, where *value*(ANS) is a set of OIDs. To evaluate the above query without the last two optional clauses, the system considers all objects in `OBJ.sel_path_exp`. For each object Y in this set, the system checks if it satisfies condition *cond*(Y.*cond_path_exp*). Boolean function *cond*() accepts a set of atomic objects, and returns *true* if one of those object values satisfy the condition. When the condition is true, Y is placed in `value(ANS)`. For example, the query `SELECT ROOT.professor Y WHERE Y.age > 40` will return `<ANS, answer, set, {P1}>` as answer. To write a query, the user must provide an entry point(OID) like $ROOT$. This entry point could be obtained from a previous query, or it could somehow be known to the user. A database name `DB` can also be used as the entry point. Using `DB.?` means that the search starts at all objects in `DB`.

Notice that the above query can span multiple databases. In Example 7.2.1, say that all objects are in database `D1` except for $A1$ which is in database `D2`. In this case $P1$ points to a "remote" object. The query `SELECT ROOT.professor Y WHERE Y.age > 40` will still return the same answer, since the query is insensitive to the "location" of objects. (Contrast this to relational queries where the relation a tuple belongs to is important.) The two optional clauses we now define attempt to control the "scope" of a query. As far as we know these clauses are not part of traditional object-oriented query languages, but as we will see, they will be useful for posing queries on views.

The `WITHIN DB1` clause limits the search to a single database `DB1`. The effect is that any OIDs that are not in `DB1` are completely ignored by the query. For our example where all nodes are in `D1` except $A1$, then our sample query with the clause `WITHIN D1` will have an empty result.

The clause `ANS_INT DB2` specifies that the answer object should be intersected with the `DB2` object to yield the final answer. This means that the answer objects are constrained to be in DB2, but the evaluation of the WHERE clause can follow remote pointers. For the example in Figure 7.2, if all nodes are in D1 except $A1$, then the query with ANS_INT D1 will return <ANS, answer, set, {P1}>. However, if all nodes except P1 are in D1, the same query will return an empty set. The ANS_INT clause is mainly used to restrict a query answer to objects within a certain database. As we will see in the next section, we can use this clause to restrict user queries to return only objects within certain views.

Finally, we want to stress that we have selected a simple language, not because other features are not useful, but because these other features are not necessary for discussing simple view management. Features such as FROM clauses (e.g., in OQL [Cat94] or Lorel [AQM+97]), multiple paths in SELECT clauses, or multiple conditions in WHERE clauses could easily be added to our language. However, some of these features would make incremental view management (Section 7.4) more complex.

7.3 Views and Materialized Views

As discussed in Section 7.1, defining views on GSDBs introduces some challenging problems regarding the structure of a view, how it can control access, and how it can be materialized. However, now that we have defined query answers to be the same as databases, we can think of views as other databases, available for querying. However, the situation is still different from a relational context because a query answer simply contains a set of OIDs that are not meaningful without the original data. In a relational view, on the other hand, the view contains all the relevant data. Another difference with relational views is that with the richer nested object model, view processing and view maintenance algorithms are substantially different.

7.3.1 Virtual Views

We start by defining *virtual views* to be the results of queries, as illustrated by the following example.

EXAMPLE 7.3.1 (A view on GSDB)

Suppose that we are interested in all persons named 'John' from the database PERSON. We write:

```
define view VJ as:                                              (7.3.2)

        SELECT    ROOT.* Y
        WHERE     Y.name = 'John'
        WITHIN    PERSON
```

In this case, objects $P1$ and $P3$ are selected, so $value(VJ) = \{P1, P3\}$. The view `VJ` is an object `<VJ, view, set, value(VJ)>`. Notice that we used a new label `view` for the view object, although this label will generally not be seen by queries. □

In general, a virtual view is defined by a *view definition query*. A graph-structured view (GS view) is an object `<V, view, set, value(V)>` where $value(V)$ is the set of OIDs returned by the view definition query. We say that an object O is in a view V if O is in $value(V)$. We also refer to the objects that are examined during query evaluation for a view as the *base objects* of the view; if those objects belong to some databases, we refer to them as *base databases*.

Virtual views can be used by queries in two ways. First, views can be used to constrain query results by using an `ANS_INT` clause. For example, query

```
        SELECT    ROOT.professor Y                              (7.3.3)
        ANS_INT   VJ
```

will return $\{P1\}$ as its answer. Object $P2$ is in $ROOT.professor$, but was excluded from the query answer because it is not in $value(VJ)$.

In our example the `ANS_INT` clause was added by the query writer. We can also envision an authorization system where user queries are automatically expanded to include `ANS_INT` or `WITHIN` clauses for the union of views the user is authorized to access. This way users would only be able to access authorized data (or retrieve authorized data). Since views can be changed, it is easy to dynamically modify the privilege of a user.

A second way virtual views can be used is as "starting points" for queries. For instance, view `VJ` defined in Example 7.3.1 contains the set of person objects containing subobjects with value 'John'. If we are interested in the age of those persons named 'John', we do not need to write the full query. Instead we can write the query `SELECT VJ.?.age`, which gives us all subobjects of objects in view `VJ` with label `age`. Thus, views can be used as important intermediate results that can be further processed by follow-on queries on the views. This can clearly make the follow-on queries much simpler.

Our view concept is quite simple: views only contain sets of objects, and do not "restructure" the objects in any way. Yet, by defining views on views, one can essentially add new structure that may be useful to applications. To illustrate, consider the views defined on the objects in Figure 7.2:

```
define view PROF as:                                        (7.3.4)
        SELECT    ROOT.*.professor Y
define view STUDENT as:                                     (7.3.5)
        SELECT    PROF.?.student Y
```

In the original database, professor and student objects were reachable from $ROOT$, in any possible depth. Now we have created a database `PROF` that only contains professors, and another one `STUDENT` that contains only their students. (A student who is not a subobject of some professor would not be included in `STUDENT`.) Queries can now access this new "professor–student" hierarchy by using the views as starting points (e.g., by starting at `STUDENT` we only reach information on students with professors), or by constraining the queries with the views (e.g., by using a `ANS_INT` clause we restrict our answer to students with professors).

7.3.2 Materialized Views

A materialized view is a stored copy of the objects in a view. The materialized copy of a view can be stored either physically close to the base databases, or at a remote site. Views are typically materialized to improve query performance and data availability, e.g., data that is expected to be accessed frequently is included in the materialized view. However, materialized views can also be used for access control and as query starting points, just as virtual views.

For a view V, each base object in $value(V)$ has a *delegate* object in the corresponding materialized view. A delegate object is a real object with the same label and type of its original object. For now, we assume that a delegate has the same value as the original object; we will discuss some variations later. A delegate object will have a new OID as described in the follow paragraph.

We need to relate the delegate of an object back to its original. Keeping this relationship is crucial for view maintenance. We use the OIDs of view objects to record this relationship. For a base object O, the OID of O's delegate object in a materialized view is obtained by concatenating the view OID with O. For example, in a materialized view MV, the delegate of base object $P1$ has OID $MV.P1$. This approach is similar to semantic OIDs as used in [PAGM96, HY90, KKS92].

Since object identifiers in a materialized view have meaning, they cannot be changed arbitrarily. Some storage systems may not allow semantic OIDs (e.g., the system assigns its own OIDs). In such cases, the view management system can generate a table that maps the local OIDs used by the storage system to the semantic OIDs needed for view management.

A materialized view is again an ordinary GSDB, i.e., it is an ordinary object **<MV, mview, set, value(MV)>**, where MV is the OID of the view and $value(MV)$ contains the OIDs of delegate objects. Our next example illustrates a materialized view.

EXAMPLE 7.3.2 (A materialized view)

The following expression defines the materialized copy of the view in Example 7.3.1.

```
define mview MVJ as:                                    (7.3.6)
        SELECT    ROOT.* Y
        WHERE     Y.name = 'John'
        WITHIN    PERSON
```

We use the keyword mview to specify that the view is materialized. The materialized view MVJ is shown in Figure 7.3. Each object in $obj(VJ)$ has a delegate in the materialized view. The view object MVJ is also shown in the figure. □

Whether a view is materialized or not should not affect query results, as long as values in delegates are the same as in the original object. For example, a query posed to MVJ should return the same results as when the query is posed to VJ. Notice that

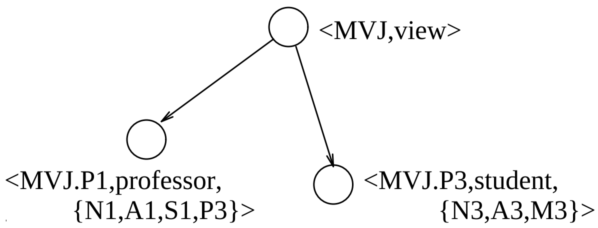


Figure 7.3: The materialized view MVJ

making delegates contain the same value as their original means that delegates may contain OIDs of base objects. For example, $N1$ in view `MVJ` is an OID of an object in database `PERSON`. A user query on `MVJ` can access $N1$ if the query allows access to remote objects (e.g., no `WITHIN MVJ` clause).

Relationships (edges) between objects in a materialized view are implicitly kept. For example, $P3$ is a child object of $P1$ in database `PERSON`. In `MVJ`, $value(MVJ.P1)$ contains $P3$. It is possible for the system that implements this view to detect the existence of $MVJ.P3$ in the view, then change $P3$ in $value(MVJ.P1)$ to $MVJ.P3$. We call this operation *swizzling the edge* from $MVJ.P1$ to $MVJ.P3$. It means changing a base OID to the OID of its delegate. Swizzling should not affect the results of queries.

The system or its manager may decide to swizzle edges to improve performance. In particular, there are two scenarios where swizzling is very useful. One is when the materialized view is stored at a site different from the base databases. In this case edge swizzling may enhance query performance by allowing local access to the referenced objects. A second scenario is when a query uses a materialized view `MV` as a starting point and also contains a `WITHIN MV` clause. In this case swizzling makes it easier to enforce the `WITHIN MV` clause. For example, a query `SELECT MVJ.professor.student WITHIN MVJ` should return $MVJ.P3$ in the answer object. If edge swizzling is done, it is easy to check that the edges traversed are in `MVJ` (i.e., check that pointers start with the `MVJ` identifier). Without swizzling, when the system decides to follow the link in $MVJ.P1$ to $P3$, it must then check if the delegate for $P3$ is in `MVJ`. Since it is (i.e., $MVJ.P3$ exists), then $MVJ.P3$ is added to the answer.

Since a materialized view is stored independently from the base data, it is possible to "manually" change the object values without affecting base objects. However, this has to be done with care since queries on the modified view may give different results from the original materialized view (or from the equivalent virtual view). Furthermore, changing the view in an arbitrary way makes it impossible to maintain the view automatically.

Nevertheless, there may be cases where modification of materialized views could be useful. For example, say we chose to swizzle all edges in a materialized view `MV`, and then remove all remaining base OIDs that appear in delegate objects. As a result, any later user query using objects in `MV` will be restricted to access only `MV` objects. This "access control" is similar to attaching a `WITHIN MV` clause to all queries, but is not identical. (With the `WITHIN` clause, a query can still retrieve a view object that contains OIDs of base objects.) A second use of view modification could be to add timestamps or other auxiliary information to delegate objects. Queries can then refer to this auxiliary information, something they could not do on the equivalent virtual view.

Notice that if a remote site defines several views that share common objects, it may end up with multiple delegates for the same base object. Sharing delegate objects of interrelated views is an interesting idea that we plan to investigate further.

7.3.3 Querying Views

Conceptually, a view is the same as an ordinary database, and objects in a view can be queried just like those in a GSDB. How the system processes a query on a view depends on whether the view is materialized or not. If the view is materialized, then the system accesses the stored objects and there is really no difference over normal query processing. (A materialized view must be kept up to date as the base data changes. This is the topic of the following section.)

When a view is virtual, one way to process a query over the view is to rewrite the query into an equivalent query that uses base objects only, just like query rewriting for relational views. However, since we lack an algebraic representation of queries, brute force rewriting may result in huge queries that are difficult to optimize.

Another method for answering queries on virtual views is to materialize the view when it is queried. Then the query is evaluated using the materialized view. The problem with this method is that the view could contain a large number of objects and the query could access a small number of them, thus resulting in a lot of wasted work.

In summary, we make view definition relatively simple. For virtual views, simplicity makes query rewriting feasible. For materialized views, as we will see, simplicity makes maintenance feasible. More complex views and relationships may be broken into multiple simple stages (multiple simple views).

7.4 Incremental Maintenance Algorithm

An incremental view maintenance algorithm produces changes to a view given the changes to base objects. In this section we develop an algorithm for graph-structured views in a centralized system, where the base databases and the materialized view reside at the same site. In this environment, the view maintenance algorithm has direct access to the base data. In the next section we will consider how to adapt the algorithm to a distributed warehousing environment. We first briefly review basic updates in a GSDB, then describe the type of view that we will develop a maintenance algorithm for. We then design the incremental view maintenance algorithm, illustrate it by examples and discuss why this new algorithm is necessary.

7.4.1 Basic Updates of GSDB

We consider three types of basic updates on a GSDB. Let N, N_1 and N_2 be OIDs.

1. $insert(N_1, N_2)$ adds the OID of N_2 into $value(N_1)$, N_1 must have a *set* type; object N_2 becomes a child of N_1 after this operation.

2. $delete(N_1, N_2)$ removes OID N_2 from $value(N_1)$, assuming that N_2 was a child object of N_1. (If no objects point to N_2 any more, N_2 may be garbage collected. However, we do not discuss garbage collection here.)

3. $modify(N, oldv, newv)$ changes the value of atomic object N from $oldv$ to $newv$.

The insertion and deletion operations add and delete edges in the base database graph. There are other possible update operations on a GSDB, for instance, the creation of a new object. Creating a new object that is not pointed at by any other object will have no impact on any queries, hence it has no effect on any views. Adding a new object O to a database DB can be modeled as $insert(DB, O)$. Most of other update operations on a GSDB can be represented as a series of base updates. For example, changing the value of a set object can be treated as insertions and deletions of its components. Changing the value of a label is also possible but not considered here.

7.4.2 Simple Materialized View

We focus on a group of simple views, where the view definition query involves only tree paths instead of path expressions. We believe that studying this simple type of view illustrates the fundamental problems of view maintenance. A few examples of those problems are: When does a base update cause a materialized view to be changed? How can one determine which object(s) are to be inserted into or deleted from a view? Is incremental view maintenance more efficient than recomputing the entire view? In Section 7.4.5 we briefly discuss how to relax our assumptions and maintain more general views.

In this section, a materialized view `MV` is defined by the following expression.

```
define mview MV as:                                   (7.4.7)
      SELECT   ROOT.sel_path Y
      WHERE    cond(Y.cond_path)
```

In the above expression, the selection path starts from object $ROOT$. The two paths sel_path and $cond\_path$ each contains a sequence of object labels without wild cards. Function $cond()$ is as defined in Section 7.2; it returns $true$ if any of the object values in $Y.cond\_path$ makes the condition true. Let $view\_path = sel\_path.cond\_path$.

As stated above, we assume that the base database has tree structure. This assumption is not crucial but simplifies the algorithm. The materialized view `MV` contains a set of delegate objects. From Section 7.3, we know that a new view object will be generated for each view defined. View object `MV` has type *set* and its components are all the delegate objects in this view.

EXAMPLE 7.4.1 (Basic update and corresponding view maintenance)
Say we are interested in professors who are younger than 45. We define a view `YP` using the objects in Example 7.2.1.

```
define mview YP as:                                          (7.4.8)

        SELECT    ROOT.professor Y
        WHERE     Y.age ≤ 45
```

The current materialized view is shown on the left hand side of Figure 7.4. The view `YP` includes a view object `YP` and an object $YP.P1$ which is the delegate of base object $P1$.

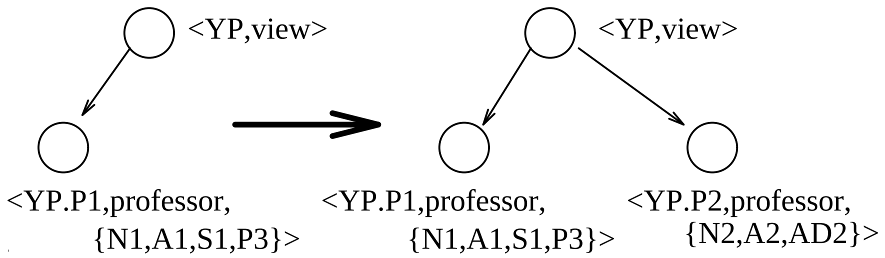


Figure 7.4: Change of the materialized view `YP` corresponding to a base update

Assume that an update $insert(P2, A2)$ occurs where $A2$ is an atomic object `<A2, age, 40>`. Now $P2$ becomes an object that satisfies the condition in the definition of view `YP`. Therefore, $YP.P2$ should be inserted into `YP`. Figure 7.4 shows the correct view after processing this update. □

Notice that in a GSDB, labels of an object can be arbitrary and can be repeated. In this example, object $P2$ does not have a child object with label *age* until $A2$ is inserted. In other cases, one object may have two or more subobjects with the same label. Therefore there might be more than one "derivations" of a view object. The maintenance algorithm needs to consider all those scenarios.

7.4.3 Incremental Maintenance Algorithm

There are two parts of an incremental view maintenance procedure. First, the view maintenance algorithm must decide whether there are any base objects whose delegates should be inserted into the view or removed from the view, due to a source update. We call these objects *affected view objects*. Second, for base objects whose delegates are in the view, when their values are changed, the corresponding delegate objects must also be changed. For example, when a child object is removed from an object O and O's delegate is in a view, then the child's OID should also be removed from the delegate object. However, for the second issue, one can always perform the same change on the delegate objects given the corresponding base change. Thus, in the following discussion, we will address the first issue only and refer to it as the view maintenance problem.

When base objects are changed, intuitively, either some new delegate objects need to be inserted into the view, or some existing delegates need to be deleted from the view. The task of an incremental view maintenance algorithm is to discover those delegates. With our model of materialized views, a view is just a collection of objects, so the maintenance algorithm need not worry about maintaining edges.

We first define a few constructs used by the algorithm. In a database with tree structure, it is clear that there is at most one path between two objects, so we use $path(N_1, N_2)$ to represent the path from N_1 to N_2. This path starts from the label of one of N_1's direct children and ends with the label of N_2. If N_1 is not an ancestor of N_2, then $path(N_1, N_2) = \phi$. Let p_1, p_2 be two paths: $p_1 = l_1 l_2 \ldots l_m$ and $p_2 = k_1 k_2 \ldots k_n$. We say $p_1 = p_2$ if $m = n$ and $l_i = k_i$ for all $1 \leq i \leq n$.

Definition: Define $ancestor(N, p)$ to be the ancestor object Y of N that satisfies $path(Y, N) = p$. If there does not exist such an object, $ancestor(N, p) = \phi$. □

Definition: Define $eval(N, p, cond)$ to return the set of objects in $N.p$ that make $cond(N.p)$ true. This function returns ϕ when no such object satisfies the condition $cond(N, p)$. □

In Example 7.4.1, $eval(P1, age, cond) = \{A1\}$ because $A1$ is in $P1.age$ and the value of $A1$ satisfies the condition given in the view definition, i.e., $value(A1) \leq 45$.

When a delegate for object Y is created and added to a view V as part of view maintenance, we must copy the contents of Y and make the OID of the new object $V.Y$. For objects with type set, we assume that OIDs in $value(V.Y)$ are not swizzled, i.e., the values in $value(V.Y)$ are base OIDs.

Definition: Define $Vinsert(VN_1, VN_2)$ to be the operation that creates delegate object VN_2 and inserts it into $value(VN_1)$. We assume if an object with OID VN_2 was already a child of VN_1, then the insertion will be ignored and VN_1 stays the same. □

Definition: Define $Vdelete(VN_1, VN_2)$ to be the operation that removes object VN_2 from $value(VN_1)$. It is the same as the *delete* operation but it operates on view objects. If VN_2 is not a child of VN_1, then nothing happens to VN_1 after the operation. (Again, VN_2 may be garbage collected if necessary.) □

Algorithm 7.4.1 shows how to identify objects whose delegate is to be inserted into or deleted from `MV`. There is more than one way to identify those objects. Since we intend to develop algorithms that will eventually apply to a distributed warehousing architecture, the algorithm we provide here isolate the computations that need access to the base databases from those that can be done without base data. Specifically, the operations that may need to examine base data are encapsulated into functions $path(ROOT, N)$, $ancestor(N, p)$ and $eval(N, p, cond)$.

The algorithm is triggered once by each update on the base objects. In a centralized environment, view maintenance can be performed by the same transaction as the triggering update, so the algorithm uses the the base databases right after the triggering update and before any further updates.

To locate objects whose delegates are potentially affected (inserted into or deleted from the view) by a base update, Algorithm 7.4.1 examines *sel_path*, *cond_path* and the updated objects. In the algorithm, Y represents an object whose delegate is potentially affected, and X is Y's atomic descendent on which the condition is tested. After the algorithm locates Y, it tests whether the original condition that makes

[4]The argument of *cond*() here is a value; the function returns *true* if that value satisfies the condition specified.

Algorithm 7.4.1 (Incremental maintenance of GS view MV)

$\triangleright$ When $insert(N_1, N_2)$ occurs:
If $view\_path = path(ROOT, N_1).label(N_2).p$ where p is an arbitrary path then
  Let $S = eval(N_2, p, cond)$;
  For all $X \in S$ do $Vinsert(MV, MV.Y)$ where $Y = ancestor(X, cond\_path)$.

$\triangleright$ When $delete(N_1, N_2)$ occurs:
If $view\_path = path(ROOT, N_1).label(N_2).p$ where p is an arbitrary path then
  Let $S = eval(N_2, p, cond)$;
  For all $X \in S$, let $Y = ancestor(X, cond\_path)$;
  If $p = p_1.cond\_path$ where p_1 is an arbitrary path
    then do $Vdelete(MV, MV.Y)$;
    else if $eval(Y, cond\_path, cond) = \phi$, then do $Vdelete(MV, MV.Y)$.

$\triangleright$ When $modify(N, oldv, newv)$ occurs:
If $path(ROOT, N) = view\_path$
then
  Let $Y = ancestor(N, cond\_path))$;
  If $cond(newv)$[4],
    then do $Vinsert(MV, MV.Y)$;
    else if $(cond(oldv)$ and $eval(Y, cond\_path, cond) = \phi)$,
      then do $Vdelete(MV, MV.Y)$.

End Algorithm

$MV.Y$ appear or not appear in the view has been changed because of this recent update. If so, $MV.Y$ is inserted into or deleted from the view as appropriate. Notice that when a deletion occurs and Y is an ancestor of N_1, we cannot simply delete $MV.Y$ from the view, because other descendents of Y may also make the condition true. This is a result of our non-unique label assumption. In this case, the condition on Y is reexamined. Figure 7.5 shows the relationship between the paths and objects used by Algorithm 7.4.1. The left two figures correspond to two possible objects layout scenarios when the base update is an insertion or a deletion, the rightmost figure corresponds to the base update $modify()$.

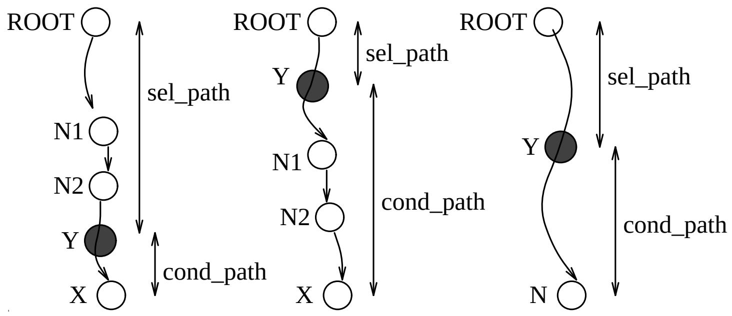


Figure 7.5: Objects and paths referred by the view maintenance algorithm

EXAMPLE 7.4.2 (Incremental view maintenance)
The following example illustrates the execution of Algorithm 7.4.1 on the scenario of Example 7.4.1.

Step 1: Update $insert(P2, A2)$ occurs.

Step 2: Since $path(ROOT, A2) = professor.age$, $sel\_path = professor$ and $cond\_path = age$, it is true that $view\_path = path(ROOT, P2).label(A2).p$ where $p = \phi$.

Step 3: Let $S = eval(A2, \phi, cond) = \{A2\}$, object $A2$ is in S because $value(A2) = 40 < 45$. So the insertion of $A2$ will enable one of $A2$'s ancestor to be inserted into the view.

Step 4: Let $Y = ancestor(A2, age) = N2$, do $Vinsert(YP, YP.N2)$. After the insertion, node $YP.N2$ appears in the view as a child of `YP`. □

To prove that Algorithm 7.4.1 is correct, we need to show that starting from an initially correct materialized view, the view will be consistent with the base data after processing each update. That is, the delegates of all view objects are in `MV`, and there are no extra objects in `MV`. The details of the correctness proof are omitted here.

7.4.4 Discussion

An important issue is the cost of incremental maintenance, especially as compared to recomputation of the entire view. In Algorithm 7.4.1, the major cost lies in evaluating functions such as $ancestor(N, p)$ that may involve access to the base databases. However, recomputing the view also involves access to the base databases. Thus, the cost of each approach actually depends on the specifics of each scenario, such as the size of the databases, the type of view, the cost of query processing and the index structure of base databases. For example, if the base database has an "inverse index" such that from each node we can find out its parent, then evaluating $ancestor(N, p)$ is straightforward. If there does not exist such an index, evaluating the same function may require a traversal from $ROOT$ to N. In general, incremental maintenance will be superior to recomputing the entire view if the view contains many delegate objects (in which case recomputation will be very expensive), and updates only impact a few, easily identifiable objects.

Another question is whether using our incremental view maintenance algorithm is better than using a relational model to represent both base data and views and then applying existing relational view maintenance algorithms. It is possible to represent objects of a GSDB in a relational fashion by "flattening" the object tree, as shown in the following example.

EXAMPLE 7.4.3 (GSDB Relational representation)

We can represent a GSDB using three tables. The first table contains the OIDs and labels of all objects, the second table contains the OIDs of all objects of type set and their children OIDs, and the third one contains the OIDs of all atomic objects and their values. For simplicity, we assume that the VALUE attribute of the third relation can hold different data types (it is a union type). The three-table representation of database `PERSON` in Example 7.2.1 is the following.

OID	LABEL
ROOT	person
P1	professor
P2	professor
P3	student
...	...

PARENT	CHILD
ROOT	P1
ROOT	P2
ROOT	P4
P1	N1
...	...

OID	TYPE	VALUE
N1	string	'John'
A1	integer	45
S1	salary	100k
N2	name	'Sally'
...	...	...

Using this relational representation, one could in principle incrementally maintain the view, but there are disadvantages. First of all, with the relational representation, a single object update can involve multiple tables. For example, an insertion of an atomic object needs to modify all three tables. This would require several invocations of the relational incremental maintenance algorithm, and could lead to inconsistencies when only some of the updates are reflected on the materialized view. For example, it would be incorrect to have a tuple (A, B) in the PARENT-CHILD table without having both A and B in the

OID-LABEL table.

Furthermore, a view defined using paths as in Section 7.3 needs to be defined by a Select-Project-Join expression with (many) self-joins. Incrementally maintaining such a view is not trivial, and we believe it could be more expensive to evaluate because the "path semantics" are hidden in the relations. □

7.4.5 Extensions of the View Maintenance Algorithm

We just provided a procedural algorithm for maintaining a group of simplified views to illustrate the fundamental problems. However, note that relaxing some of the restrictions we imposed on the view definition in Section 7.4.2 is easy. For example, handling views with more than one select path or more than one condition is straightforward. On the other hand, relaxing the following two assumptions (which we believe is important to do for some applications) is not simple:

- Allow the *sel_path* and *cond_path* to be general path expressions with wild cards. To maintain this type of view, the maintenance algorithm needs to be able to test path containment for general path expressions. For example, any path p is contained in path expression $*$. If a view is defined by expression `SELECT ROOT.*`, then any insertion of a $ROOT$'s descendent node will cause delegate objects to be inserted into the view.

- Allow base databases to be arbitrary graphs. The maintenance algorithm will be similar to Algorithm 7.4.1, except that now there may be more than one path between two objects. Therefore, the actual implementation of the algorithm, e.g., computing $ancestor(Y, p)$, is more difficult. We will study views whose base data are arbitrary graphs in the next chapter.

7.5 Maintaining GS Views in a Warehouse

When a materialized view is stored at a data warehouse that is separated from the data sources, the warehouse can apply the same algorithm. However, when only the warehouse (and not the data sources) knows the view definition, evaluating functions such as $ancestors(N, p)$ or $eval(N, p, cond)$ needs to be done by communicating with source databases.

The most straightforward way for the warehouse to implement those functions is to send queries back to the data sources. The warehouse (or the source wrappers) needs to translate the functions into one or more source queries and integrate the answers. If the source can evaluate any queries required to evaluate those functions, then the warehouse can directly apply Algorithm 7.4.1 in Section 7.4. When a source can only support some simple querying interface, then the warehouse can decompose the evaluation of a function into multiple simple queries, integrate the answer and evaluate the target functions.

EXAMPLE 7.5.1 (Realizing a function using source queries)
Assume that a source accepts queries with format `fetch` X `where` $func(X)$, and returns all objects X (their OIDs and values) from this source that satisfy conditions specified in $func(X)$. Also, assume the warehouse knows that the source can evaluate function $path(X, Y)$. Then, the warehouse may translate the evaluation of $ancestor(Y, p)$ into `fetch` X `where` $path(X, Y) = p$. For $eval(N, p, cond)$, the warehouse may use `fetch` X `where` $path(N, X) = p$ to obtain all objects in $N.p$, then test condition $cond()$ on those objects locally. □

There are two major problems with querying the sources. First, it is expensive. If the capability of a source is very limited, evaluating one function may involve many complex interactions, involving query and answer translation. Furthermore, sending queries and answers consumes time and network bandwidth, and leads to poor availability if a source is down.

Second, querying the source, especially multiple times to evaluate the same query, may result in inconsistent data. Just like when maintaining relational views, when the source is autonomous, source updates may interfere with query evaluation and resulting in inconsistent query results. Following is an example of inconsistent view maintenance. The example shows that two objects appear in the view at the same time while they should not.

EXAMPLE 7.5.2 (Inconsistent view maintenance)
Let A, B, C and D be source objects shown in Figure 7.6. Let the view definition be `SELECT ROOT.a Y WHERE Y.b >10`. Initially, A was a child of $ROOT$ but B was not. Object A satisfies condition of view V, so $MV.A$ is in MV at the warehouse.

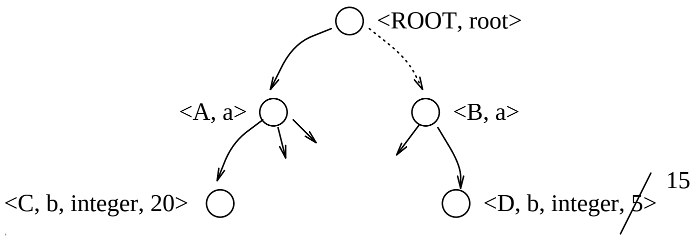


Figure 7.6: The base database before and after the given source updates

Let there be three source updates: $U_1 = addedge(ROOT, B)$, $U_2 = delete(ROOT, A)$, $U_3 = modify(D, 5, 15)$. Object B does not satisfy view condition before U_3 modifies D, and does afterwards. We can see that $MV.A$ and $MV.B$ should not appear in the view simultaneously because $MV.A$ should be removed from the view after processing U_2, and $MV.B$ should be inserted into the view only after processing U_3. However, when the warehouse receives U_1 and starts processing, the test of whether B satisfy the view condition is done by sending a query Q back to the source. If U_3 occurs before the evaluation of query Q, then the answer to Q will include object B. So $MV.B$ is inserted into the view. At this time the view contains both $MV.A$ and $MV.B$, which is inconsistent with any source database state. □

Analogous to the relational approaches, the view maintenance anomaly problem may also be solved by compensating or using unique OID information (like keys). However, due to the lacking of rigid structure, it is even more complicated to develop algorithms that can correctly maintain a graph-structured view. For example, to compensate a query result, the algorithm may need to conctruct a graph from subsequent updates and decide whether the query result would have been different if this "update graph" did not exist in the base database. Given the complexity of accessing source data and correcting maintenance anomalies, it will be greatly benificial if a view can be maintained without refering back to the data sources, i.e., if the view is *self-maintainable*. We address this issue in the next chapter.

Chapter 8

Self-Maintainability of GS Views

As we pointed out in the last chapter, there are several disadvantages to accessing source data when maintaining a materialized view: It takes time, it requires source capability of answering certain queries, it may be complicated due to the composition of query answers, and it may run into consistency problems. Therefore, it is desirable to perform view maintenance without accessing base data. Intuitively, we say that a materialized view V is *self-maintainable* if V can be incrementally maintained without referencing the base data.

In this chapter, we study the self-maintainability issues for graph-structured materialized views. We start in Section 8.1 by introducing our framework and comparing the problems and solutions with those in previous research. We define self-maintainability and related terminologies in Section 8.2 and present self-maintainability test in Section 8.3. Sections 8.4 addresses how the maintainer may evaluate the test conditions, and provide some "rules of thumb" that can be used to simplify the evaluation. Section 8.5 presents one type of view that is always self-maintainable, and, base on that, discusses how to keep auxiliary data to help maintain a view.

8.1 Introduction

We model our view maintenance process in the following environment. The base data is stored at a data source, the materialized views are stored at a data warehouse. As shown in Figure 8.1, a module called view *maintainer* receives the base updates and maintains the materialized views[1]. The maintainer may use a *self-maintainability test* (SMT) to decide whether an update is self-maintainable with respect to a view. During the test, the maintainer may access auxiliary information such as indices stored for the purpose of view maintenance, and may also query the view. After deciding the self-maintainability of an update, the maintainer can maintain the view using a view maintenance algorithm, with or without querying the base data.

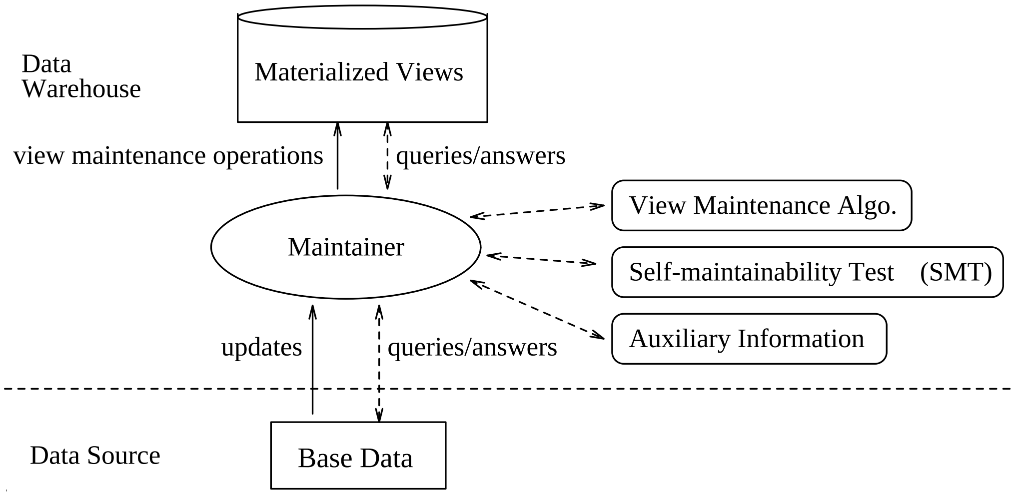


Figure 8.1: Conceptual view maintenance environment

Self-maintainability has been studied in the relational context but not for graph-structured data, mostly because maintaining graph-structured materialized views is a new and complicated problem. We now address the differences between the self-maintainability issues of these two types of views, and present related research.

[1]A maintainer is similar to a view manager in previous chapters. We chose a different name because it is used to focus on a different aspect of view maintenance — maintainability instead of consistency.

8.1.1 Self-maintainability of Relational and GS views

The view definitions for relational data are different from those for graph-structured data: Relational views are defined by relational algebra expressions or SQL statements, while graph-structured views can be defined using path expressions or a specialized query language over graphs. As a result, view maintenance techniques vary substantially for these two data models, as do the approaches to decide whether a view is self-maintainable.

Besides the definitions and maintenance algorithms, there are at least three other major differences between the self-maintainability issues of relational views and GS views. The first is in the update information: For the relational model, the update information usually contains the updated tuple, but representing update information for the graph-structured data model is more complicated. In OEM, each object has an object identifier (OID), a label, and a value. If an object O is involved in an update, for example, does the update information include the OID of O, its label, its value, or all of them? How about O's children and descendent objects? The same view may or may not be self-maintainable when different update information is available.

The second complication of solving the self-maintainability problem for graph-structured views is the lack of "scope restrictions" on base updates. In particular, the unit of relational updates is a tuple, while in a GSDB, a simple change like adding a link between two objects can potentially connect two graphs. If we modify a tuple in a relation, we know that only views defined on that relation will be affected. In a GSDB, however, adding a link between two objects can dramatically change the set of objects in a view.

The third difference is in the actual procedure of deciding whether a view is self-maintainable. For the relational model, the tests for self-maintainability are often represented by standard relational operators (e.g., [TB88]) or query containment rules (e.g., [Huy96]). Performing relational operations and testing query containment have been studied separately and thus are not part of the self-maintainability problem. For GSDB model, the tests may involve, for example, checking whether there is a path from object X to object Y in the base database (Section 8.3). There are no standard methods to test such a condition.

8.1.2 Related Work

Discussion of relational view self-maintainability can be found in [TB88, BCL89, QGMW96, GJM96, Huy96]. They mainly focus on SPJ views. They also focus on deciding self-maintainability given a fixed scope of information. As we have mentioned, different approaches must be taken for testing self-maintainability of GS views.

As we have mentioned in last chapter, views defined in [Suc96] using the query language UnQL [BDHS96] are "join-free", and they are self-maintainable when updates are either concatenation or replacement of two disjointed graphs. We defined our views using an extension of OQL and design incremental view maintenance algorithms using common insertion, deletion, and update operations (Chapter 7). These views are not join-free — updating some objects may affect the presence of other objects in a view. Thus, these views are generally not self-maintainable.

8.2 Self-maintainability and Irrelevance

Self-maintainability means that without the base data, the maintainer has all necessary information to maintain a view. Like relational views, whether a GSDB view can be self-maintained depends on what information is available to the maintainer. Table 8.1 shows the possible information that can be used for view maintenance, most of which are discussed separately in the specified sections.

The first three categories of information are almost always necessary for view maintenance, the rest are additional information that may be used to help determine self-maintainability. In practice, whether some information can be used depends on whether the maintainer is able to obtain the information, and whether it is cost effective to use it. For example, in Section 8.5 we discuss what auxiliary data can be stored for a view such that the auxiliary data together with the view are always self-maintainable.

In general, with Φ as a subset of information specified in Table 8.1, we give the following definition:

	Relational	Graph Structured
basic information	view definitions	view definitions
	updated tuples	OIDs of updated objects
	type of base update	type of base update
additional information	—	additional upd. info. (Section 8.4)
	base relation schema	base database info. (Section 8.4)
	base key info. [GJM96]	unique label info. (Section 8.4)
	constraints [TB88, QGMW96]	constraints
	auxiliary views [QGMW96]	auxiliary views (Section 8.5)
	view contents [Huy96]	view contents (Section 8.5)

Table 8.1: Information that can be used to maintain a materialized view

Definition: An update U is *self-maintainable* with respect to a view V given information Φ if, corresponding to U, V can be maintained (i.e., the new V can be obtained) using information in Φ only. We also say that the view V is *self-maintainable* with respect to update U given Φ. □

If V is self-maintainable with respect to all updates, we simply say V is *self-maintainable.* We want to re-emphasize here that there are two aspects of view self-maintainability. First, there is the question of whether a view can always be self-maintained regardless of the base update. This view property may be decided when the view is defined. Second, with respect to a specific update, there is the issue of whether a view can be self-maintained. This decision must be made at run time when the update is available.

A notion related to self-maintainability is *relevance*, which indicates whether a view is affected by a base update. Intuitively, an update U is irrelevant to a view V if V stays the same when evaluated over the base database before and after update U.

Definition: An update U is *maintenance irrelevant* to a view V given information Φ if it is possible to decide from Φ that V stays the same when evaluated over the base database before and after update U. □

Detecting irrelevant updates may save a lot of unnecessary work to maintain views unaffected by base updates. Obviously, for the same update, view, and Φ, maintenance irrelevance implies self-maintainability, but not vice versa. Just like deciding self-maintainability, deciding maintenance irrelevance depends on Φ. For example, if the label of an updated object is known, and the label is not used in any of the paths in the definition of view V, then one can determine that this update is irrelevant to V. On the other hand, if only OIDs of updated objects are available, one is not able to tell whether the same update affects the view.

Finally, we want to point out that most of the research on self-maintainability of relational views first gives the set of information used (e.g., updated tuples and keys), then defines self-maintainability as "maintainable." Implicitly, their maintainable means maintainable given the this fixed set of information. Therefore, the definitions in different previous papers categorize different set of updates as self-maintainable, e.g., maintainable given key information, or maintainable given view content. We believe that the information used to maintain the view should explicitly be given by the definition.

8.3 Self-maintainability and Irrelevance Test

In this section we develop a self-maintainability test, SMT. This test is performed by the maintainer (recall Figure 8.1) to determine if an update is maintenance irrelevant, self-maintainable, or not self-maintainable. For SMT, the maintainer must evaluate certain conditions (given below). Some of the conditions refer to the state of the base data. When the maintainer is unable to evaluate those conditions, the update is not self-maintainable. However, in other cases, the maintainer may know enough (from the information that came when the update was propagated, or from its available information Φ) to evaluate the condition and determine irrelevance or self-maintainability of the update.

Notice that in this Section we only describe the SMT, and we will address how the test can be performed in the next Section. A "smarter" maintainer with more knowledge may perform SMT more efficiently, however, efficiency is not our major goal for developing SMT.

8.3.1 Notation

We study self-maintainability of simple views defined in Section 7.4.2. However, we assume that the base database can be an arbitrary directed graph. As a result, there may be multiple paths between two objects. Let $path(Y, X) = \{p | X \in Y.p\}$; it contains the set of paths that can lead from Y to X. If Y is not an ancestor of X, then $path(Y, X) = \phi$. If Y and X are the same object, then $path(Y, X) = \{\lambda\}$. We define $path(Y, X).path(Z, W) = \{p_1.p_2 | p_1 \in path(Y, X), p_2 \in path(Z, W)\}$. For a path p, $\lambda.p = p$.

By default, we assume that when the maintainer receives notification of a base update (Section 7.4.1), it always receives the OIDs of the updated base objects (N_1 and N_2) and the type of the update (insert, delete, or modify). Additional update information such as labels of updated objects may also be sent to the maintainer. We will discuss them in Section 8.4.

8.3.2 Testing Maintenance Irrelevance

Figure 8.2 graphically illustrates what SMT is trying to discover. As shown in Section 7.4, the figure illustrates scenarios where the view may be affected. In particular, the view may be affected when the updated link between N_1 and N_2 (either added or removed) is on a path from $ROOT$ to one of the objects X that satisfy the select condition of the view definition. Removing the N_1, N_2 link may mean that one of the Y objects that were selected by the view should no longer be in the view. Similarly, adding link between N_1 and N_2 may mean that a new Y should be added. Note that the candidate Y object may follow N_1, N_2 on the path from $ROOT$ to X (Scenario 1 in the figure), or may precede N_1, N_2 (Scenario 2). The situation for modify is similar to Scenario 2, where Y can only be on the path from $ROOT$ to N_2. Let $view\_path = sel\_path.cond\_path$.

Our first SMT condition checks if there is a path from $ROOT$ to the updated object(s) that is relevant to the view.

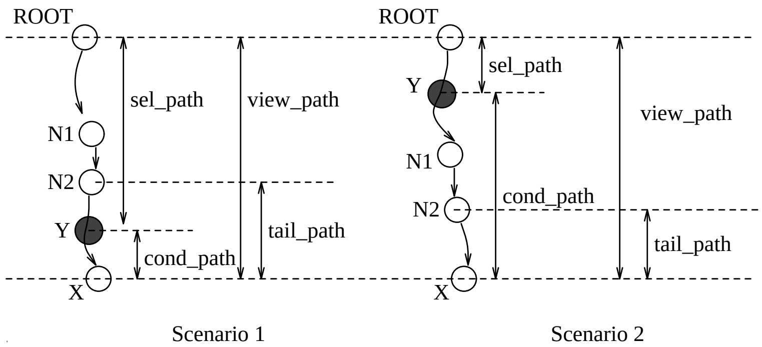


Figure 8.2: Updated base objects and different paths

Condition (a) In the base database, right after the update completed,

- For an insert or a delete: There exists a path $p \in path(ROOT, N_1)$ such that $p.label(N_2)$ is a prefix of $view\_path$.
- For a modify: There exists $p \in path(ROOT, N_2)$ such that p is a prefix of $view\_path$. □

In a graph-structured database, there may be more than one path from $ROOT$ to N_1. Any one of those paths may make Condition (a) true. If the maintainer can ascertain that Condition (a) was false at the base database, then it knows that the update is irrelevant to the view.

If the maintainer knows that the condition was true, the maintainer can check for the rest of the path, from N_2 to object X (see Figure 8.2). To do so, for each $p \in path(ROOT, N_1)$ we define a $tail\_path$ to be the portion of the $view\_path$ that was not traversed to reach N_2. That is, for each $p \in path(ROOT, N_1)$ that makes Condition (a) true, $view\_path$ should equal $p.label(N_2).tail\_path$. If $view\_path = p.label(N_2)$, then let $tail\_path = \lambda$. (For a modify update, $view\_path$ should equal $p.tail\_path$). If there is more than one p, there is more than one $tail\_path$. Let T be the set of all $tail_path$s. For example, if $view\_path = a.b.a$, and $label(N_2) = a$, then if $ROOT$ can reach N_2 through both path a and path $a.b.a$, we have $tail\_path = b.a$ or $tail\_path = \lambda$.

The second SMT condition checks whether the select condition is satisfied by some descendents of N_2.

Condition (b) In the base database, right after the update completed,

- For an insert or delete: there exists $tail\_path \in T$ and an object X such that $X \in N_2.tail\_path$ and $sel\_cond(X) = true$.
- For a modify, there exist a $tail\_path \in T$ such that $tail\_path = \lambda$ and $sel\_cond(oldv) \neq sel\_cond(newv)$. □

If the maintainer knows that Condition (b) is false, then the update is again irrelevant to the view. If both Conditions (a) and (b) are true, then the update potentially affects the view and the maintainer needs to evaluate the SMT Conditions (c) and (d).

Sometimes the maintainer is unable to evaluate Conditions (a) or (b) from its information in Φ. In this case, we say that the outcome of the condition evaluation is *unavailable*. For example, when the value of X is unknown, the maintainer cannot evaluate $sel\_cond(X)$. If the maintainer cannot evaluate Condition (b), then the view is not self-maintainable. If the maintainer cannot evaluate Condition (a), then it is impossible to obtain T. However, the maintainer can still evaluate Condition (b) for each potential $tail\_path$ that could be in T. For example, if $view\_path = professor.student.major$, then there are only three potential $tail_path$s: $major$, $student.major$, or $professor.student.major$. If Condition (b) is false for all potential $tail_path$s, then the update is maintenance irrelevant to the view. However, if Condition (b) is true or unavailable for any of the $tail_path$s, the update is not self-maintainable.

Table 8.2 summarizes the decisions after evaluating Conditions (a) and (b). Remember that maintenance irrelevance implies self-maintainability.

The following example shows how the maintainer can use SMT Conditions (a) and (b) to determine whether an update is irrelevant to a view. It also shows that the maintainer may make different decisions when different information is available.

Condition (a)	Condition (b)	Decision
False	Any value	Maintenance irrelevant
True	False	Maintenance irrelevant
True	Unavailable	Not self-maintainable
True	True	Evaluate Conditions (c) and (d)
Unavailable	(all) False	Maintenance irrelevant
Unavailable	(any) Unavailable	Not self-maintainable
Unavailable	(any) True	Not self-maintainable

Table 8.2: Summary of decisions on Conditions (a) and (b)

EXAMPLE 8.3.1 (Testing Maintenance Irrelevance)
Consider a simple view `VSMITH` defined as `SELECT ROOT.professor Y, WHERE Y.name = 'smith'`. Consider a base update $delete(ROOT, Z)$. Assume the maintainer receives label information for updated objects and $label(Z) = professor$. For view `VSMITH`, $view\_path = professor.name$. Since $path(ROOT, ROOT) = \lambda$ and $label(Z)$ is a prefix of $view\_path$, Condition (a) is true and we have $T = \{name\}$. If the maintainer has no information about Z's sub-objects, then the maintainer cannot decide whether Condition (b) is true or false, and the update is not self-maintainable. On the other hand, if the maintainer knows that object Z has no sub-object with label *name*, then $Z.name = \phi$, Condition (b) is false, and the update is maintenance irrelevant to the view according to Table 8.2. □

8.3.3 Testing Self-maintainability

When both Conditions (a) and (b) are true, the update is relevant to the view, meaning that potentially there are Y objects (Figure 8.2) to be inserted into or deleted from the view. Condition (c) checks whether the maintainer can obtain all those Y objects from Φ.

Condition (c) We define **Y** as the set of objects that are potentially affected by the update:

- For an insert or a delete: Let $S = \{X | X \in N_2.tail\_path$, where $tail\_path \in T$ and $sel\_cond(X)\}$. S refers to the base database right after the update completed.

For all $X \in S$,

let $\mathbf{Y1} = \{\mathrm{Y}\ | sel\_path \in path(ROOT, N_1).label(N_2).path(N_2, Y)$ and $cond\_path \in path(Y, X)\}$,

let $\mathbf{Y2} = \{Y | sel\_path \in path(ROOT, Y)$ and $cond\_path \in path(Y, N_1).label(N_2).path(N_2, X)\}$.

Let $\mathbf{Y} = \mathbf{Y1} \cup \mathbf{Y2}$.

- For a modify: Let $\mathbf{Y} = \{\ Y | sel_path \in path(ROOT, Y)$and $cond\_path \in path(Y, N_2)\}$.

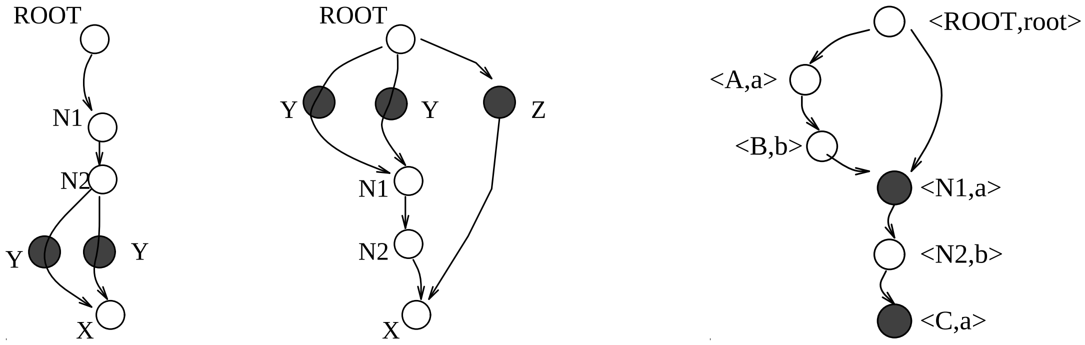
Condition (c) is true if for an insert, OIDs, labels and values of all objects in $\mathbf{Y}$ can be obtained from Φ; for a delete, OIDs of all objects in $\mathbf{Y}$ can be obtained from Φ. (A modification is treated as an insert if not $sel\_cond(oldv)$ but $sel\_cond(newv)$; it is treated as a delete if $sel\_cond(oldv)$ but not $sel\_cond(newv)$.) □

In Condition (c), $\mathbf{Y1}$ contains Ys in Scenario 1 and $\mathbf{Y2}$ contains Ys in Scenario 2. Here we want to stress two issues related to the evaluation of Condition (c). First, there may be more than one Y object corresponding to one X, as shown in Figure 8.3(a), and the maintainer must find all of them. The maintainer has to be especially careful with objects like Z in Figure 8.3.3, which also satisfies $sel\_path \in path(ROOT, Z)$ and $cond\_path \in path(Z, X)$ but whose existence in the view is not affected by the current update.

Second, when there are multiple $tail_path$s, the maintainer has to find all corresponding Ys to ascertain that (c) is true. Sometimes for the same view and the same update, there are Ys in both Scenario 1 and 2, as shown in Example 8.3.2.

EXAMPLE 8.3.2 (Multiple potential view objects corresponding to one source update)

Let the view be defined as `SELECT ROOT.a.b.a Y, WHERE Y.b = 'value'`. In the base database shown in Figure 8.3(b), both object N_1 and C are potential Ys. In particular, since both $a \in path(ROOT, N_1)$ and $a.b.a \in path(ROOT, N_1)$, we have $T = \{a.b, \lambda\}$ and $\mathbf{Y} = \{C, N_1\}$. C is a potential Y object in Scenario 1 and N_1 is a potential Y object in Scenario 2. □



(a) More than one Y object corresponding to one X

(b) Both Scenario 1 and 2 occur for the same view and update

Figure 8.3: Possible object layouts when evaluating Condition (c)

If the maintainer cannot obtain any Y object specified in Condition (c), then (c) is false and this view cannot be maintained. If (c) is true and the base update is an insertion, then the view can be maintained by inserting the delegate objects of Ys into the view. If the base update is a delete, however, we are not done yet. For a delete, the maintainer must know whether Y's delegate object should stay in the view after the deletion. Object Y may stay in the view because it can be reached from $ROOT$ via another path (for Scenario 1), or because it has more than one "derivation" due to other descendent objects (for Scenario 2). For instance, for the view `SELECT ROOT.professor Y WHERE Y.student.major = 'math'`, if a professor object has two students majoring in math and one of them graduates, this professor object should remain in the view. But if the professor has only one math student object who was just removed, the professor object should be removed from the view too. Remember that for modify, when $sel\_cond(oldv)$ but not $sel\_cond(newv)$, it is treated as a delete in Scenario 2.

Let $\mathbf{Y}$, $\mathbf{Y1}$,$\mathbf{Y2}$ be the collections of affected view objects defined in Condition (c). We first define a Condition (d') that indicates whether a single object $Y \in \mathbf{Y}$ should stay in the view after a delete.

Condition (d') In the base database, right after the delete, for $Y \in \mathbf{Y1}$, it is still true that $sel\_path \in path(ROOT, Y)$; for $Y \in \mathbf{Y2}$, there still exists $X \in Y.cond\_path$ and $sel\_cond(X) = true$. □

For a $Y \in \mathbf{Y}$, if the maintainer can decide that (d') was true, then Y stays in the view. If (d') was false, Y should be removed from the view. If the maintainer cannot make a decision on (d'), then it does not know whether Y should stay in the view, and hence the update is not self-maintainable. The last SMT Condition (d) should be evaluated for all deletes.

Condition (d) For all $Y \in \mathbf{Y}$, Condition (d') can be evaluated. □

If Condition (d) is true, i.e., Condition (d') is either true or false for all $Y \in \mathbf{Y}$, then the maintainer can decide whether a Y should stay in the view for all $Y \in \mathbf{Y}$. Therefore this update is self-maintainable. If Condition (d) is false, then there is at least one $Y' \in \mathbf{Y}$ such that Condition (d') is unavailable for Y', and the update is not self-maintainable. Table 8.3 summarizes the decisions after evaluating Conditions (c) and (d):

Condition (c)	Condition (d)	Decision
False	Any value	Not self-maintainable
True (insertions)	Any value	Self-maintainable
True (deletions)	True	Self-maintainable
	False	Not self-maintainable

Table 8.3: Summary of decisions on Conditions (c) and (d)

8.3.4 Discussion

The SMT we just developed is instance specific — it makes a decision for a given update and a view. It can be used at run time to decide whether a view is self-maintainable for an update. It can also be used to prove the self-maintainability of a view (e.g., Example 8.3.3) or a type of view (e.g., complete views in Section 8.5), by proving that tall updates will pass or fail specific tests.

EXAMPLE 8.3.3 (A self-maintainable view)
In this example we show that a simple view `PROF` defined as `SELECT ROOT, ROOT.professor Y` is self-maintainable given labels of updated objects. We do so by showing that all updates are either maintenance irrelevant or self-maintainable to this view. For this view, $view\_path = sel\_path.cond\_path = professor$. Consider an arbitrary update U:

- When U is a modify, then U is maintenance irrelevant to the view, because the view has no select conditions and thus Condition (b) is false.
- When U is an insert or a delete,
  - If $N_1 \neq ROOT$, then since $label(N_1).label(N_2)$ has at least two labels and cannot be a prefix of $view\_path$, Condition (a) is false and U is maintenance irrelevant.
  - If $N_1 = ROOT$ and $label(N_2) \neq professor$, then (a) is false because $label(N_2)$ is not a prefix of $view\_path$. Therefore, U is again maintenance irrelevant.
  - If $N_1 = ROOT$ and $label(N_2) = professor$, then (a) is true and $tail\_path = \lambda$. Since $X = N_2 \neq \phi$, Condition (b) is true. Since $cond\_path = \lambda$, we have $Y = X = N_2$, so Condition (c) is also true.
    * When U is an insert, it is self-maintainable to the view.
    * When U is a delete, $ROOT$ may reach N_2 via other paths. However, by examining the value of $ROOT$, it is possible to detect whether $ROOT$ has more than one links to N_2 and thus give a true or false answer to (d). Therefore, the update is also self-maintainable.

Since all possible updates are either maintenance irrelevant or self-maintainable, view `PROF` is self-maintainable. □

The SMT is complete and sound. That is, for given information Φ, the maintainer can use SMT to detect all self-maintainable updates, and any self-maintainable update will be detected. We do not formally prove the completeness and soundness here. Intuitively, evaluation of the four conditions cover the process of maintaining a view given an update. If an update U is categorized as irrelevant by the SMT, then the maintainer needs to do nothing to maintain the view. If U is categorized as self-maintainable, Condition (c) must be true, so the Ys can be obtained and the view can be maintained. If U is categorized as not self-maintainable, there must be a decision or piece of data that is essential to the view maintenance that is missing from Φ.

SMT conditions can be simplified when the underlying base database has tree structure. For a tree structured database, there is at most one path from one object to another. Therefore, there is at most one path from $ROOT$ to N_1 or N_2, and there is at most one $tail\_path$ per update. Furthermore, there is at most one Y corresponding to each X. As an example, the set of potentially affected view objects $\mathbf{Y}$ in Condition (c) can be obtained by the following expressions.

- For an insert or a delete: Let $S = \{X | X \in N_2.tail\_path \text{ and } sel\_cond(X)\}$. S should be evaluated in the base database right after the update completed. Let $\mathbf{Y} = \{Y | Y = ancestor(X, cond\_path), \forall X \in S\}$.
- For a modify: Let $\mathbf{Y} = \{ancestor(N_2, cond\_path)\}$.

Locating the ancestor of an object may be easier and faster than locating an object that lies on one of many potential paths between two objects (as required by Condition (c)), especially in the case where the parent pointers of base objects are given. Therefore, if the base database is tree-structured, the maintainer may choose to simplify the SMT conditions before testing them.

In the next section we will discuss the mechanics of evaluating the SMT conditions given additional information about the base updates, the base database, or the view.

8.4 Performing the SMT

Relational self-maintainability tests are specified by standard relational operators (e.g., [TB88]) or query containment rules (e.g., [Huy96]), thus there is no need to discuss how to perform the tests. In contrast, performing SMT may involve, for example, checking whether there is a path from one object to another in the base database. There are no standard methods to test such a condition.

In general, the more knowledge the maintainer has about the updated objects or the base data, the more updates can be self-maintained. In this section we show how additional information can be used to either evaluate or simplify the evaluation of the SMT conditions. Remember that by default the maintainer only has the OIDs of the updated objects. In the following we first consider options on additional update information provided to the maintainer as part of the update notification, then briefly address the use of additional knowledge on base databases.

8.4.1 Label and Value Information

It is often relatively easy to send the labels and values of the updated base objects to the maintainer. Example 8.4.1 shows how SMT Condition (a) can be evaluated given label information.

EXAMPLE 8.4.1 Using update label information to determine irrelevance Say we have a view `SELECT ROOT.professor.student` and an update $U = insert(ROOT, Z)$. If the maintainer has the label of Z and $label(Z) = secretary$, then $label(Z)$ is not part of $view\_path$, hence, Condition (a) is false and U is irrelevant to the given view. □

We develop ad hoc rules called *SM rules* for the maintainer to perform SMT automatically. The following Rule (a1) applies to the case of Example 8.4.1.

Rule (a1) Using label information, if $label(N_1).label(N_2)$ is not a sub-path of $view\_path$, then Condition (a) is false. (Hence, the update is maintenance irrelevant to the view.) ◇

An SM rule is labeled as Rule (kn) if it is the nth rule and is related to the evaluation of Condition (k). We can see that an SM rule specifies some qualifications on Φ; if satisfied, the maintainer can directly obtain the result of an SMT condition. Providing SM rules is one way to test SMT conditions for a subset of updates or views.

8.4.2 Descendent Information

The maintainer may be provided with the OIDs, labels, and values of N_1 and N_2, plus the OIDs, labels and values of all N_2's descendent objects. In many scenarios, descendent information is essential to evaluate Conditions (b) to (d) and ensure that an update is self-maintainable.

Locating all descendent objects of N_2 may be difficult in a graph-structured base database. Fortunately, only a limited number of layers of N_2's descendents, not all, are necessary to evaluate Conditions (b) to (d). This number is equal to the size of the largest *tail_path* in T. Let $L = maximum(|t|)$ where $t \in T$. Because *tail_paths* are only known after evaluating Condition (a) and are update specific, the maintainer may set the required levels of descendent objects to $|view\_path|$, an upper bound of L. We use $descendent(N_2, L)$ to represent the first L levels of N_2's descendent objects (their OIDs, labels and values).

Here are a few example SM rules on how descendent information can be used. Since evaluating Condition (b) uses objects in $N_2.tail\_path$, we have

Rule (b2) Objects in $descendent(N_2, L)$ are sufficient to evaluate Condition (b). $\diamond$

After evaluating Condition (a) the maintainer can decide whether each $Y \in \mathbf{Y}$ is on the path from N_2 to X, or on the path from $ROOT$ to N_1, by comparing the corresponding *tail_path* with the *cond_path*. When Y is on the path from N_2 to X (Scenario 1 in Figure 8.2), if all descendents of N_2 are given to the maintainer, then Y can be obtained because Y itself is a descendent of N_2.

Rule (c3) If all $Y \in \mathbf{Y}$ are in Scenario 1, then if $descendent(N_2, L)$ is known, Condition (c) is true. $\diamond$

Also, because all Y's descendent objects are descendent objects of N_2 as well, we have the following rule:

Rule (d4) If Y is in Scenario 1, then if $descendent(N_2, L)$ is known, the maintainer can decide whether Condition (d') is true or false and thus, Condition (d) is true. $\diamond$

8.4.3 Root-path Information

In this case, the maintainer is provided with the OIDs and labels of N_1 and N_2, plus $path(ROOT, N_1)$. For a tree-structured base database, it is usually simple to send root-path information to the maintainer, since the base update transaction probably traversed the base database until it reached the updated objects. Therefore, it may record the object labels along the path and report them as part of the update. However, for graph-structured data there may be more than one path from $ROOT$ to N_1 and all of them need to be reported to the maintainer. SM rule (a5) tells us that Condition (a) can be evaluated using root-path information:

Rule (a5) If $path(ROOT, N_1)$ is given as part of the update information, Condition (a) can be evaluated by path comparison. $\diamond$

8.4.4 Restricted Base Updates

Knowledge of base update restrictions can also be used to simplify the evaluation of SMT conditions. The following are a few example scenarios:

- The base database only performs updates of a certain type. For example, many databases are append only, and for those databases, Condition (d) does not need to be evaluated.

- The base database only updates atomic objects. Remember that a set object with an empty value (i.e., no children) is also considered an atomic object. In a tree structured database, an insert or a delete of a subtree can be represented as a sequence of inserts or deletes of atomic (leaf) objects. One example of only deleting atomic objects is the UNIX delete file command `'rm -r dir'`, which removes each file in a directory first, then removes the empty directory.

  Rule (b6) When N_2 is atomic, if $\lambda \notin T$, Condition (b) is false; if $\lambda \in T$ and $value(N_2)$ is known to the maintainer, the maintainer can test Condition (b) by evaluating the select condition on the value of N_2. $\diamond$

  Since it is often easy to test the select condition on a given object, the maintainer may evaluate Condition (b) before (a), and only evaluate (a) when the result of (b) is true.

- The base database only updates objects with a certain label. For example, say a materialized view reflects stock data for a given portfolio. For this view, only changes of stock quotes are propagated incrementally into the view, and other changes (e.g., to the portfolios) can be reflected by a periodic refresh every evening. Then, the maintainer only needs to evaluate updates with label *quote* against SMT. In this scenario, for example, if this label is not part of the *view_path* of a view, then according to Rule (a1), the all updates are maintenance irrelevant.

8.4.5 Knowledge of Base Database

Schema information (e.g., DTDs of XML), unique labels and other constraints on the base database are all optional for a graph-structured database. When they are available, they can be used to simplify the evaluation of SMT conditions. For example, SMT conditions can be simplified when the underlying base database has tree structure.

We give one example here on how knowledge of base database can be used to facilitate SMT evaluation. More detailed discussion can be found in [ZGM99]. In this example, we represent the additional knowledge on base data using *neighboring-label information (NLI)*. NLIs are expressions defined to represent the local structural information of a GSDB. They tell us what patterns of labels may appear together and which do not. The expressions may be defined using keywords like **ALWAYS**, **NEVER**, and **UNIQUE**.

EXAMPLE 8.4.2 (Using NLI to evaluate SMT conditions)
Let the view definition be `SELECT ROOT.professor.student` and assume object $ROOT$ has unique label $root$. Let the update be $U = insert(P, S)$. Assume the maintainer receives the labels of updated objects: $label(P) = professor$ and $label(S) = student$.

Let the base database information be given in an expression `ALWAYS(`$root$`)`.$professor$, meaning that all professor objects are first-level children of $ROOT$. According to this information, the maintainer knows that Condition (a) is true because P is a child of $ROOT$, and $label(P).label(S)$ is a prefix of the $view\_path$. The maintainer obtains the corresponding $tail\_path = \lambda$, and Condition (b) is true since there exists an element of $S.tail\_path$, namely S itself, that satisfies the select condition (which is always true in this example). Condition

(c) is also true because $Y = S$ and $\mathbf{Y}= \{\mathrm{S}\}$ is available to the maintainer. Thus, update U is self-maintainable. However, if the maintainer does not know `ALWAYS(`*root*`).`*professor* to hold, then P can appear without being a first-level child of $ROOT$, and the same update may or may not affect the view. □

We conclude this section by pointing out that update or base database information can be combined to perform more powerful test, as we did in Example 8.4.2. More sophisticated SM rules can also be developed for combined information.

8.5 Complete Views and Auxiliary Data

In this section we first describe one type of view, a *complete view*, that is always self-maintainable. Then using complete views, we show how to keep auxiliary data in order to maintain an arbitrary view without accessing base data.

8.5.1 Complete Views

We first give the definition of complete views, then discuss some of their properties.

Definition: For a given base database, given object O and path p, define a *complete view* $CV(O, p)$ as a view that contains all objects that can be reached from O using a prefix of p. We abbreviate $CV(ROOT, p)$ as $CV(p)$. □

EXAMPLE 8.5.1 (A complete view)

The complete view $CM = CV(professor.student.major)$ contains all objects that are reachable from $ROOT$ through path $professor.student.major$. Figure 8.4 shows the base database given in Section 7.2 and the view CM over this database. The view CV contains (the delegate of the) five objects shown in the shaded area. Because λ is a prefix of $view\_path$, $ROOT$ itself is in the complete view. □

Notice that a complete view cannot be defined by the simple view definition given in Section 7.4.2. However, a complete view can be seen as the union of simple views. For example, view CM in Example 8.5.1 can be defined by four `SELECT` statements.

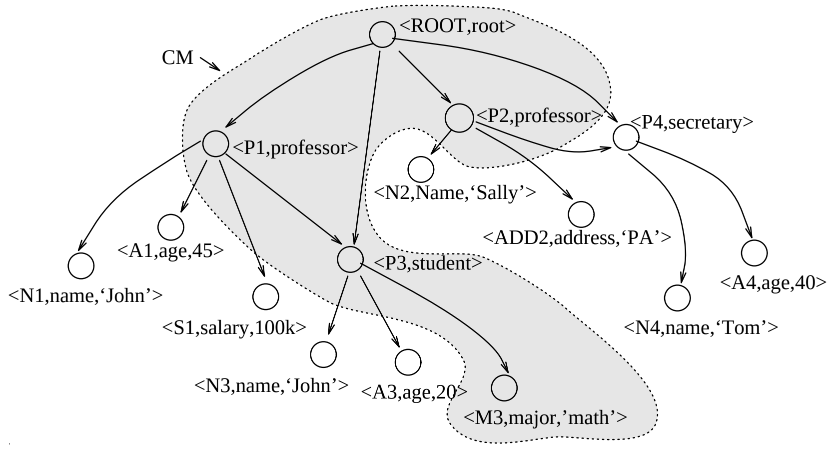


Figure 8.4: Complete view **CM**

We use D_{-U} to denote the database D before an update U occurs, and D_{+U} to denote the same database right after U occurs. We use $V[D_{-U}]$ and $V[D_{+U}]$ to denote the extensions of view V over database D at the corresponding state. For a given update, let T be the set of *tail_path*s defined in Section 8.3.2. The following theorem states the maintenance properties of complete views.

Theorem 8.5.1 *For a complete view $C = CV(view\_path)$ and an update U,*

- *If U is a modify, then U is maintenance irrelevant to C.*
- *If U is a delete, then U is self-maintainable to C given $C[D_{-U}]$, the view content before the update.*
- *If U is $insert(N_1, N_2)$, then U is self-maintainable with respect to C given $C[D_{-U}]$ and*
  $CV(N_2, tail\_path)[D_{+U}]$ for all $tail\_path \in T$. □

The correctness proof of Theorem 8.5.1 is given in Appendix G.1. Intuitively, a complete view C contains all objects that are reachable from $ROOT$ via certain paths. A modify does not change an object's reachability from $ROOT$, and thus does not change C. After a base deletion, N_2 and some of N_2's descendent objects may need to be removed from C. However, which objects should be removed can be decided from $C[D_{-U}]$. For insert, $C[D_{-U}] \cup CV(N_2, tail\_path)[D_{+U}]$ (for all $tail\_path \in T$)

is $C[D_{+U}]$, which is the new view content after the update (Appendix G.1). For all three cases, an update is maintainable for view C.

The purpose of studying complete views is that they can be stored as auxiliary data to maintain other views. The following theorem states the relationship between a complete view and the self-maintainability of a view with the same *view_path*, assuming the maintainer has the basic information in Table 8.1.

Theorem 8.5.2 *For a simple view V and an update U,*

- *If U is a modify or a delete, $CV(view\_path)[D_{-U}]$ contains all objects sufficient and necessary to maintain view V with respect to U.*
- *If U is an insert, $CV(view\_path)[D_{+U}]$ contains all objects sufficient and necessary to maintain view V with respect to U.* □

The proof of correctness of Theorem 8.5.2 is given in Appendix G.2. Theorem 8.5.2 can be used by the maintainer to decide what auxiliary information to store in order to make a view self-maintainable. We discuss this issue next.

8.5.2 Keeping Auxiliary Data

The bottom line is that to maintain view V with $view\_path = sel\_path.cond\_path$, the maintainer may need all the objects in $C[D_{-U}]$ and $C[D_{+U}]$ where $C = CV(view\_path)$. These objects can be found in the existing view, sent along with update information, or stored as auxiliary views. (Of course, an object in both $C[D_{-U}]$ and $C[D_{+U}]$ should preferably be stored or sent only once.) In the following we discuss two options on what to store and what to send. Notice that the auxiliary structures themselves (together with the view and update) must be self-maintainable.

1. The maintainer stores all objects that may potentially appear in the view or on a path that will be examined during the evaluation of SMT conditions. (Objects that are already in the current view need not be stored again.) All objects in $C[D_{-U}]$ are such objects. Actually, any object with a label that appears in *view_path* may potentially be examined. So for this method, if the base

database is B, then the auxiliary view AV is the union of all such view defined by "`SELECT B.l Y`", where l is a label appearing in *view_path*. (A database, like a view, is also an object that can be used in a query [ZGM98a].) With respect to the three types of base updates given in Section 7.4.1, AV is itself self-maintainable. (Actually, AV is maintenance irrelevant.)

For this method, the basic information in Table 8.1 is enough to maintain view V. But this method may result in keeping more information than necessary. For example, for a view `NAMES` defined as `SELECT ROOT.professor.name`, the maintainer needs to store all professor and name objects (even if they are not currently reachable from $ROOT$.) There is no need for the maintainer to keep a secretary object, for example, but it needs to keep the names of the secretaries. Although it does not make sense semantically, one could potentially insert a link from a professor object to the name sub-object of a secretary.

2. The maintainer stores $C[D_{-U}]$, that is, the content of the view C before the update. This information is sufficient to maintain view V with respect to delete and modify. For inserts, the maintainer also needs $CV(N_2, tail\_path)[D_{+U}]$ for all $tail\_path \in T$ to maintain V. In practice, the base site may simply send $descendent(N_2, L)$ (L is the maximum length of all $tail_path$s) or $descendent(N_2, \infty)$ (all descendents of N_2), supersets of necessary information.

   For example, to maintain the view `NAMES` above, name objects that are only reachable from secretary objects do not need to be stored. If they were to appear in the view, they can be obtained from update information.

   From Theorem 8.5.1, the auxiliary view $C[D_{-U}]$ is also self-maintainable given the update information. The current view content, $V[D_{-U}]$, contains a subset of objects in $C[D_{-U}]$. If the maintainer is allowed to access the current view content, it may not need to store those objects again as auxiliary data.

Of course, the maintainer may choose to store no auxiliary data and let the base site report all necessary information with updates. The cost of storing and maintaining the auxiliary information versus the cost and feasibility of querying base data determines which approach to use. Heuristics can be used to determine this relative cost. For example, if the expected size of $CV(view\_path)$ is substantially smaller than the size of the base database, it may be beneficial for the maintainer to store $CV(view\_path)$.

8.6 Summary

As semistructured and XML data become commonplace in database systems, it becomes important to understand how graph-structured materialized views can be efficiently maintained. In this chapter we have studied this maintenance problem, and in particular, self-maintainability of graph-structured views. To our knowledge, this is the first research effort on this topic. We presented a self-maintainability test that decides the relevance of updates, and we showed how the test can be performed. We proved that complete views are self-maintainable, and we discussed how to keep auxiliary data to help maintain a view.

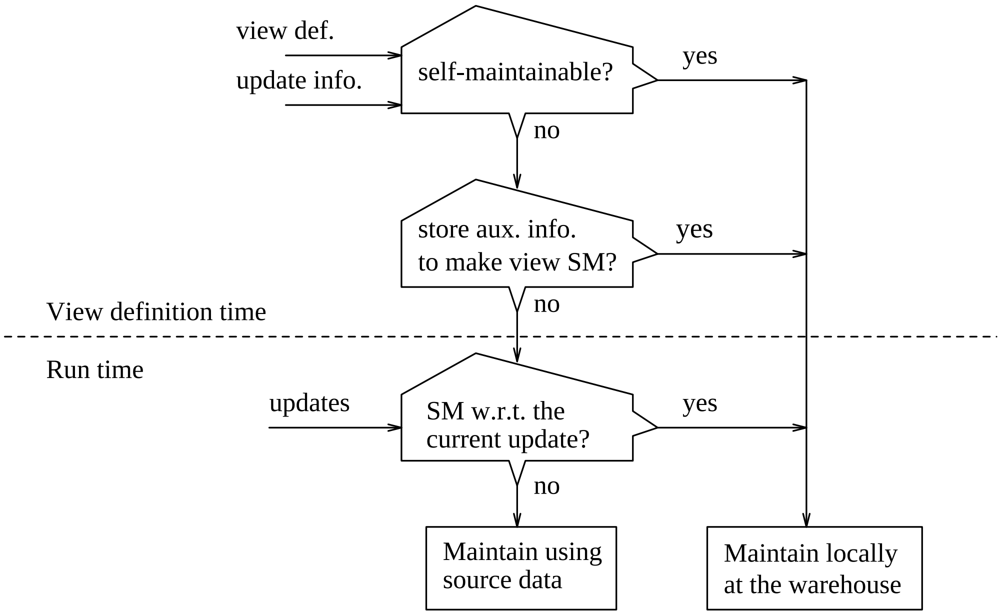


Figure 8.5: Using SMT to maintain materialized views

Techniques described in this chapter can be used to facilitate view maintenance, as shown in Figure 8.5. When a view is defined, the maintainer can decide whether it is self-maintainable. If not, the maintainer may decide whether to store auxiliary data or acquire more information about the base database to make the view self-maintainable. At run time, self-maintainable views can be maintained without accessing base data. For other views, the maintainer can decide whether a given update is irrelevant or maintainable. SMT (or a set of SM rules) is used in all three decision stages. If an update is not self-maintainable, then an algorithm accessing base data (e.g., one described in [ZGM98a]) must be used.

We assumed that in the simple view defined in Section 7.4.2, both the *sel_path* and the *cond_path* are constant paths. In fact, when they are path expressions containing wild cards (e.g., $*$, $professor.*$ and $professor.?$), the four SMT conditions described in Section 8.3 can still be used to determine the self-maintainability of the view. However, when evaluating the test conditions, all possible instances of a path expression must be considered. In particular, computing *tail_path*s becomes more complicated. For example, for a view defined as `SELECT ROOT.* Y WHERE Y.major = 'math'` and a base update that modified an object with label *major*, we have $T = \{*, *.major\}$. In general, computing T requires path containment techniques, which we do not discuss further. Our definition and theorems for complete views also apply directly to views defined using path expressions.

Chapter 9

Conclusion and Future Work

Materialized views copy selected or summarized data of interest to improve query performance and data availability. They have become a focus of study recently for two major reasons: the enormous interest in data analysis (warehousing, OLAP, mining) and the increasing need to improve the performance of complex queries.

In this dissertation we have studied consistency issues for maintaining materialized views in a data warehouse. We defined correctness for materialized views, and the levels of correctness that capture the correspondence between the base data and the derived views. We designed a family of algorithms that incrementally and consistently maintain materialized views in a data warehouse. We presented the implementation of the algorithms in the WHIPS prototype and some performance results. As mentioned in Chapter 1, although we framed our discussion in a data warehousing environment, our consistency definitions can apply to general derived data. We also addressed the issues of defining and maintaining materialized views over graph-structured data.

There are still many interesting problems in the area of maintaining materialized views in a distributed environment. In particular, our algorithms were designed mainly with a focus on correctness, not efficiency. Many optimizations can be done to improve the performance of the algorithms. For example, there are many smarter ways to evaluate compensating queries (such as evaluating them using previous query results, or combining common subexpressions of queries), or the warehouse could copy selected base data to facilitate view maintenance. Other than maintaining individual views defined directly on base relations, there are many issues related to consistently maintaining a hierarchy of views at the warehouse. One such issue is the concurrency control between the view maintenance operations and warehouse user transactions; that is, how to allow users to read consistent, current views while concurrently performing view maintenance.

Research on specifying and maintaining graph-structured views has so far been preliminary. With the increased amount of data available online, mostly in hierarchical instead of relational or strongly-typed objected-oriented format, handling views over this data becomes very important. There are many unsolved problems in this area, for example: Do we hide or expose edges? How does one define and maintain partially materialized views, for example, views that materialize a few levels of objects and leave the rest as pointers back to base data? We also need to justify whether our view definition is reasonable, whether our view maintenance algorithm is useful, and how to improve the view maintenance efficiency.

We conclude the dissertation by answering two frequently asked questions, and raising one related question.

1. Is the view maintenance anomaly problem particular to join views?

   No. The view maintenance anomaly was caused by the need to fetch more data from autonomous data sources. Self-maintainable views, such as select views, and duplicate-preserving project and union views, can be maintained without going back to the data sources, thus maintenance causes no anomalies. Maintaining other views, such as views defined using an aggregate operator, will encounter similar anomaly problems because obtaining source data while performing maintenance is necessary.

2. Can the ideas and solutions presented in this dissertation apply to views defined using other operators?

   Yes, as long as the view itself is *incrementally maintainable*. We say a view is incrementally maintainable if it satisfies the following condition:

   Let Q be a query with format $Q = V\langle U_1, U_2, \ldots U_k\rangle$ where $U_1, U_2, \ldots, U_k (k \geq 0)$ is a set of arbitrary source updates. (When $k = 0$, $Q = V$.) Then, for any update U_i where ss_{i-1} and ss_i are the source states before and after U_i occurs respectively, $Q[ss_i] = Q[ss_{i-1}] + Q\langle U_i\rangle[ss_i]$.

Intuitively, incrementally maintainable means that the view deltas can be obtained by applying the view definition to the source deltas ($V[ss_i] = V[ss_{i-1}] + V\langle U_i\rangle[ss_i]$). Lemma A.0.2 in Appendix A shows that SPJ views are incrementally maintainable. If a view is incrementally maintainable, then the algorithms developed in this dissertation directly apply.

Therefore, the right question is whether a view defined using a new operator can be incrementally maintained even in centralized environments, when locking base data is permitted.

If a view is not incrementally maintainable, e.g., an aggregate view, new algorithms may need to be developed. Similar ideas, like using compensation or keys, can be applied. In WHIPS, we provide a simpler and cleaner, but maybe less efficient method to handle aggregate views by defining the SPJ part as a base view and the aggregate part as a derived view.

Finally, we end with a challenging question suggested by the research in this dissertation. Assume that a *test scenario* includes a set of base relations, a view, one or more base updates and one event interleaving scenario (e.g., update U_i occurs before answer A_j is computed at source s). Then, is it possible to design a finite set of test scenarios, such that the consistency level of any given view maintenance algorithm can be determined by running the algorithm against this test set?

For instance, Examples 3.3.1 and 3.3.2 in Chapter 3 can be two test scenarios. There a view is defined using three base relations, there are three source updates, and there are different assumptions about the interleaving of updates and queries. For both examples, the Conventional algorithm ran into view maintenance anomalies, ECA correctly maintains the view but may bundle view deltas corresponding to more than one update together, and CCA always updates the view corresponding to each source update. From this we can conclude that the Conventional algorithm is inconsistent, ECA is at most strongly consistent, and CCA is at most complete. However, just from the two examples, we cannot determine the correctness levels of

ECA and CCA, because we do not know whether they will ever be inconsistent for a different view or under a different update scenario.

Therefore, we are looking for a complete set of test scenarios such that, after an algorithm processes those scenarios, the lowest consistency level it generates is the the consistency level of the algorithm. For example, if an algorithm maintains the view in a complete way for all of the test scenarios, then this algorithm is complete. Of course, to simplify the test, one can limit the basic constructs of the algorithms to some pre-defined operators, e.g., +, -, the substitution operator $Q\langle U\rangle$, etc. We plan to study this problem further.

Appendix A

Proof of Correctness for ECA Algorithm

In this appendix we provide a correctness proof for the Eager Compensating Algorithm (ECA) (Section 3.3). Theorem A.0.1 shows that ECA is strongly consistent. Before getting to the theorem, we need to establish five lemmas.

First, we claim that not all events will affect outcome of a query evaluation. For example, if a source event se has type S_qu, then the base relations are not modified. Consequently, the result of evaluating another query before or after se are the same.

Formally, let se_i be an arbitrary source event; the source state before se_i is ss_{i-1} and after is ss_i. Let we_i be a warehouse event and the warehouse state before we_i is ws_{i-1} and after is ws_i.

Lemma A.0.1 *Independent events using ECA.*

se_i *has type* $S\_qu_i \Longrightarrow V[ss_{i-1}] = V[ss_i],\ Q[ss_{i-1}] = Q[ss_i]$

we_i *has type* $W\_up_i \Longrightarrow V[ws_{i-1}] = V[ws_i]$

we_i *has type* W_ans_i *with* $UQS \neq \phi \Longrightarrow V[ws_{i-1}] = V[ws_i]$ □

Proof: Obvious from the definition of events (Section 3.1) and events described in ECA. ■

With ECA, the triggering relationship among source and warehouse events is as shown in Figure A.1.

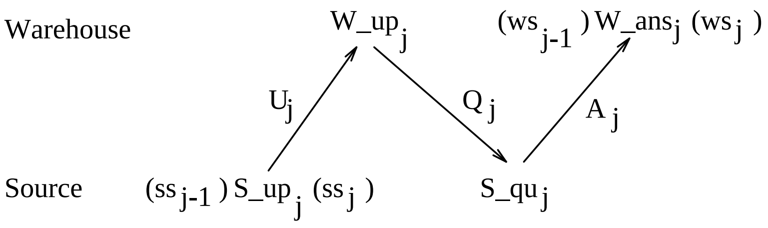


Figure A.1: Triggering relations among events.

Since there is a one-to-one correspondence between updates and the four events they trigger, in the rest of the appendix we will label events by the id of the update that generated them. Specifically, let $U_1, U_2, \ldots$ be the updates that occurred during the execution of ECA algorithm. For each U_j, the events it triggers are S_up_j, W_up_j, S_qu_j and W_ans_j. Also, let Q_j be the query involved in the processing of U_j and A_j be the answer relation of Q_j. According to Lemma A.0.1, at the source only events of type S_up affect query evaluation, so we can assume the source state only change after each S_up event, that is, we can label the source state before S_up_j as ss_{j-1} and label it after S_up_j as ss_j. Similarly, assume warehouse states only change after each type W_ans events.

In the next lemma, we show how a type S_up event affects the evaluation of a query. Assume Q is an arbitrary query generated by ECA and is targeted for execution before a source update U_j (at state ss_{j-1}). However it is actually executed after U_j (at state ss_j).

Lemma A.0.2 $Q[ss_{j-1}] = Q[ss_j] - Q\langle U_j\rangle[ss_j]$ *for any query Q defined by formula 3.1 in Section 3.2.* □

Proof: Consider a term T in Q as defined by formula 3.1: $T = \Pi_X(\sigma_C(\tilde{r}_1 \times \tilde{r}_2 \times \ldots \times \tilde{r}_n))$. Assume U_j is an update on relation r_m. If r_m is not used in T, then $T\langle U_j\rangle = \phi$, and $T[ss_j] = T[ss_{j-1}]$. Therefore we have $T[ss_j] = T[ss_{j-1}] + T\langle U_j\rangle[ss_j]$ holds. If r_m is used in defining T, then for r_l other than r_m, $\tilde{r}_l[ss_j] = \tilde{r}_l[ss_{j-1}]$. And $\tilde{r}_m[ss_j] = \tilde{r}_m[ss_{j-1}] + tuple(U_j)$. Then

$$\begin{aligned}
T[ss_j] &= \Pi_X(\sigma_C(\tilde{r}_1[ss_j] \times \tilde{r}_2[ss_j] \times \ldots \times \tilde{r}_m[ss_j] ... \times \tilde{r}_n[ss_j])) \\
&= \Pi_X(\sigma_C(\tilde{r}_1[ss_j] \times \tilde{r}_2[ss_j] \times \ldots \times (\tilde{r}_m[ss_{j-1}] + tuple(U_j)) \ldots \times \tilde{r}_n[ss_j])) \\
&= \Pi_X(\sigma_C(\tilde{r}_1[ss_{j-1}] \times \tilde{r}_2[ss_{j-1}] \times \ldots \times \tilde{r}_m[ss_{j-1}] \ldots \times \tilde{r}_n[ss_{j-1}])) +
\end{aligned}$$

$$\begin{aligned} & \Pi_X(\sigma_C(\tilde{r}_1[ss_j] \times \tilde{r}_2[ss_j] \times \ldots \times tuple(U_j) \ldots \times \tilde{r}_n[ss_j])) \\ = \quad & T[ss_{j-1}] + T\langle U_j \rangle [ss_j]. \end{aligned}$$

Since $Q = \sum_i T_i$ defined by Formula 3.2 is a sum of terms, we have $Q[ss_j] = Q[ss_{j-1}] + Q\langle U_j \rangle [ss_j]$. Thus $Q[ss_{j-1}] = Q[ss_j] - Q\langle U_j \rangle [ss_j]$. ∎

Lemma A.0.2 states a very important property of query Q. Intuitively, Lemma A.0.2 says that the intended answer of Q, $Q[ss_{j-1}]$, can be obtained by the difference of the actual answer, $Q[ss_i]$, and the answer of a compensating query, $Q\langle U_j \rangle [ss_j]$. Remember that a query in ECA (and LCA, CCA) is generated by substituting one or more base relations using updated tuples. If for all such query, Lemma A.0.2 holds, then the corresponding view is said to be *incrementally maintainable* (Chapter 9). All incrementally maintainable views can be maintained by algorithms presented in this dissertation, because Lemma A.0.2 is used to prove the correctness of all algorithms and it is the only one that refers to the property of the view.

In ECA, if $UQS(W\_ans_j) = \phi$, then the warehouse updates the materialized view MV. The following lemma states that when the warehouse updates MV, the last answer it gets(A_j) is the intended one for Q_j .

Lemma A.0.3 $UQS(W\_ans_j) = \phi \Longrightarrow A_j = Q_j[ss_j]$. □

Proof: Assume there exists an S_up_m, such that $S\_up_j < S\_up_m < S\_qu_j$. Then from the triggering rule described in Section 3.1 we know that $W\_up_j < W\_up_m < W\_ans_j$, and $W\_ans_j < W\_ans_m$. Therefore, $Q_m \in UQS(W\_ans_j)$ and this implies that $UQS(W\_ans_j) \neq \phi$. This is a contradiction, therefore all source events between S_up_j and S_qu_j(if there are any) must be of type S_qu. By Lemma A.0.1 and our labeling convention, the source state right before S_qu_j is still ss_j, and A_j is evaluated at this state, $A_j = Q_j[ss_j]$. ∎

Definition: An event of type W_ans updates the materialized view at the warehouse when its UQS is empty. For a warehouse event we, W_ans_i is its *Previous-Updating-Event* if $W\_ans_i < we$, $UQS(W\_ans_i) = \phi$, and $\not\exists W\_ans_m$ such that $W\_ans_i < W\_ans_m < we$ and $UQS(W\_ans_m) = \phi$. □

The next lemma shows that for two consecutive updates on the warehouse view, if the first one updates the view into a consistent state, then so does the second one.

Lemma A.0.4 *Assume $UQS(W\_ans_j) = \phi$ and W_ans_i is the Previous-Updating-Event of W_ans_j. Then $V[ws_i] = V[ss_i] \Longrightarrow V[ws_j] = V[ss_j]$.* □

Proof: We know that the updates occurring between U_i and U_j (if any) are U_{i+1}, U_{i+2}, ..., U_{j-1}. Before updating MV at event W_ans_j, the $COLLECT$ set is $COLLECT = \sum_{l=i+1}^{j} A_l$ since the last time $COLLECT$ was set to empty was at event W_ans_i. Furthermore we have $V[ws_j] = V[ws_i] + COLLECT$.

Now we can reduce the proof that $V[ws_j] = V[ss_j]$ assuming $V[ws_i] = V[ss_i]$ to the proof of the following equation with $COLLECT = \sum_{l=i+1}^{j} A_l$:

$$V[ss_i] + COLLECT = V[ss_j] \tag{A.1}$$

The proof of Equation A.1 is done by induction on the number of updates between U_i and U_j, which is $m = j - i - 1 \geq 0$.

1. Base case: $m = 0$.

   There is no update between U_i and U_j so U_i and U_j are consecutive updates. Since $UQS(W\_ans_j) = \phi$, we have $Q_j = V\langle U_j \rangle - \phi = V\langle U_j \rangle$.

$$\begin{aligned} V[ss_i] + A_j &= V[ss_i] + Q_j[ss_j] && (Lemma\ A.0.3) \\ &= V[ss_i] + V\langle U_j \rangle[ss_j] && (ECA) \\ &= V[ss_j] && (Lemma\ A.0.2) \end{aligned}$$

   Thus, Equation A.1 holds for the base case.

2. Induction step:

   Induction hypothesis: Assume Equation A.1 holds for all $m < c$.

   When $m = c$, we have c updates between U_i and U_j. Let us refer this scenario with c updates as Scenario S. In Scenario S, some of the queries in $Q_{i+1} \ldots Q_{j-1}$ are evaluated before U_j occurs at the source, some after.

   Let

$$Q_{before} = \sum_{\substack{l=i+1 \\ Q_l \notin UQS(W\_up_j)}}^{j-1} Q_l \quad \text{and} \quad Q_{after} = \sum_{\substack{l=i+1 \\ Q_l \in UQS(W\_up_j)}}^{j-1} Q_l$$

Correspondingly, let

$$A_{before} = \sum_{\substack{l=i+1 \\ Q_l \notin UQS(W\_up_j)}}^{j-1} A_l \quad \text{and} \quad A_{after} = \sum_{\substack{l=i+1 \\ Q_l \in UQS(W\_up_j)}}^{j-1} A_l$$

So

$$COLLECT = A_{before} + A_{after} + A_j \tag{A.2}$$

Let Scenario S' be exactly the same as Scenario S except U_j and all four corresponding events do not occur. In Scenario S' we label all events and states with a prime. Thus the updates are $U'_1, U'_2, \ldots$, the queries are $Q'_1, Q'_2, \ldots$, and so on. Many of the S' events and states correspond to the ones in S. In particular, $V' = V$(we have the same view), and $\forall l, i \leq l \leq j-1$, $U'_l = U_l$, $Q'_l = Q_l$ and $ss'_l = ss_l$.

Since U_j is absent in Scenario S', $UQS(W\_ans'_{j-1}) = \phi$. There are $c - 1 < c$ updates between U_i and U_{j-1}. By induction hypothesis we have

$$V'[ss'_i] + COLLECT' = V'[ss'_{j-1}] \tag{A.3}$$

In Scenario S', $COLLECT'$ is the sum of answers A'_i though A'_{j-1}. Let us divide the answers into two sets:

A'_{before} = answers to the queries that constitute Q_{before}

A'_{after} = answers to the queries that constitute Q_{after}

Thus, $COLLECT' = A'_{before} + A'_{after}$, Equation A.3 can be written as

$$V[ss_i] + A'_{before} + A'_{after} = V[ss_{j-1}] \tag{A.4}$$

A_{before} are those answers computed before U_j in Scenario S, they are computed at the same state in Scenario S', thus $A_{before} = A'_{before}$.

A_{after} contains those answers that were computed after U_j occur in Scenario S. Since $UQS(W\_ans_j) = \phi$, by similar arguments to those in Lemma A.0.3, we know that all answers in A_{after} are computed at state $ss_j = state(S\_up_j)$:

$$A_{after} = Q_{after}[ss_j] \tag{A.5}$$

In Scenario S', answers in A'_{after} must be computed after U_{j-1} occurs, otherwise they will not be evaluated after U_j in Scenario S. So $A'_{after} = Q_{after}[ss_{j-1}]$, and Equation A.4 becomes:

$$V[ss_i] + A_{before} + Q_{after}[ss_{j-1}] = V[ss_{j-1}] \tag{A.6}$$

Furthermore, we know that $Q_j = V\langle U_j\rangle - Q_{after}\langle U_j\rangle$ from Algorithm ECA. This gives us $Q_j[ss_j] = V\langle U_j\rangle[ss_j] - Q_{after}\langle U_j\rangle[ss_j]$. From Lemma A.0.2 we have that $Q_{after}[ss_{j-1}] = Q_{after}[ss_j] - Q_{after}\langle U_j\rangle[ss_j]$. Subtracting the latter from the former equation we obtain

$$Q_{after}[ss_j] + Q_j[ss_j] = Q_{after}[ss_{j-1}] + V\langle U_j\rangle[ss_j] \tag{A.7}$$

The intuitive meaning of Equation A.7 is that if the queries in Q_{after} and Q_j are evaluated in state ss_j as in Scenario S, the result will be the same as evaluating those queries in Q_{after} in state ss_{j-1} as in Scenario S', and adding the term $V\langle U_j\rangle[ss_j]$ that reflects the effect of U_j on the view.

Combining all of our results we can show that

$$\begin{aligned}
& V[ss_i] + COLLECT \\
= \;& V[ss_i] + A_{before} + A_{after} + A_j && (\ A.2) \\
= \;& V[ss_i] + A_{before} + Q_{after}[ss_j] + A_j && (\ A.5) \\
= \;& V[ss_i] + A_{before} + Q_{after}[ss_j] + Q_j[ss_j] && (Lemma\ A.0.3) \\
= \;& V[ss_i] + A_{before} + Q_{after}[ss_{j-1}] + V\langle U_j\rangle[ss_j] && (\ A.7) \\
= \;& V[ss_{j-1}] + V\langle U_j\rangle[ss_j] && (\ A.6) \\
= \;& V[ss_j] && (Lemma\ A.0.2)
\end{aligned}$$

This shows that Equation A.1 holds in Scenario S, and completes the induction proof. As stated earlier, Equation A.1 is sufficient to prove Lemma A.0.4. ∎

Lemma A.0.5 *Assume* $V[ws_0] = V[ss_0]$, *then* $UQS(W\_ans_j) = \phi \Longrightarrow V[ws_j] = V[ss_j]$ *for any* j. □

Proof: The proof is done by induction on j.

1. Base case: $j = 1$.

   Assume there is an "empty update" U_0 which occurs before any updates and its four events (which are also empty) are processed before any other events. After S_up_0 the source state is ss_0 and after W_ans_0 the warehouse state is ws_0. Since there are no pending queries when W_ans_0 is processed, W_ans_0 is the *Previous-Updating-Event* of W_ans_1.

   Since $V[ws_0] = V[ss_0]$ and $UQS(W\_ans_1) = \phi$, by Lemma A.0.4 we have $V[ws_1] = V[ss_1]$.

2. Induction step: Induction hypothesis: the lemma holds for any $j < c$ $(c > 1)$.

   When $j = c$, let W_ans_i be the *Previous-Updating-Event* of W_ans_j. Since $i < c$, by induction hypothesis we have that $V[ws_i] = V[ss_i]$. Then from Lemma A.0.4 we can prove that $V[ws_j] = V[ss_j]$.

■

Theorem A.0.1 *Algorithm ECA is strongly consistent for a finite sequence of updates $U_1, U_2, \ldots, U_k$.* □

Proof: Assume that $V[ws_0] = V[ss_0]$, that is, the warehouse is in a consistent state with the source before any events occur. Consider two arbitrary warehouse events: $ws_i < ws_j$. We first show that using ECA, there exist source states ss_m and ss_n such that $ss_m \leq ss_n$, $V[ws_i] = V[ss_m]$ and $V[ws_j] = V[ss_n]$.

Let us first consider ws_i. If ws_i has type W_ans and $UQS(ws_i) = \phi$, then let $ws_{i'} = ws_i$. Otherwise, let $ws_{i'}$ be the Previous-Updating-Event of ws_i. From Lemma A.0.1, we know that $V[ws_i] = V[ws_{i'}]$ since there is no event between $ws_{i'}$ and ws_i which may update MV. Since $ws_{i'}$ is always an event of type W_ans with $UQS(ws_{i'}) = \phi$, we know from Lemma A.0.5 that $V[ws_{i'}] = V[ss_{i'}]$ where $ss_{i'}$ is the source state right after the triggering update of $ws_{i'}$ occurs.

Similarly for ws_j, there exists a warehouse event $ws_{j'} \leq ws_j$ which has type W_ans and $UQS(ws_{j'}) = \phi$. Further more, there exists source state $ss_{j'}$ such that $V[ws_{j'}] = V[ss_{j'}]$.

Since $ws_i < ws_j$, we have $ws_{i'} \leq ws_{j'}$. From the event ordering rules we know there must be $ss_{i'} \leq ss_{j'}$ as well. So there exists $ss_m = ss_{i'} \leq ss_n = ss_{j'}$ such that $V[ws_i] = V[ss_m]$ and $V[ws_j] = V[ss_n]$.

Now we show that ECA is convergent. Notice that $UQS(W\_ans_k) = \phi$ since U_k is the last update. Lemma A.0.5 gives $V[ws_k] = V[ss_k]$. At the source, the last update is S_up_k and after that, all events(if any) have type S_qu. So ss_k is the final source state. At the warehouse, W_ans_k is the last event so ws_k is the final warehouse state. Thus, ECA is convergent and hence strongly consistent. ∎

Appendix B

Proof of Correctness for ECA-Key Algorithm

In this appendix we show that ECA-Key (Section 3.3.4) is strongly consistent. Consider a warehouse event we_k where $UQS(we_k) = \phi$. The warehouse state right after this event, $state(we_k) = ws_k$. Let F be the set of all updates that have been seen at the warehouse by we_k. Let $U_k \in F$ be the last update received before or by we_k. Let se_k be the S_up source event that processed U_k, and let ss_k be the source state immediately after se_k. We first show that $V[ws_k] = V[ss_k]$ using ECA^K.

Lemma B.0.6 $V[ws_k] = V[ss_k]$. □

Proof: We can show that $V[ss_k] = V[ws_k]$ by contradiction and considering the various cases. That is, assume that $V[ss_k] \neq V[ws_k]$. Then there must be a tuple t that is either missing from $V[ws_k]$ or is an extra tuple in $V[ws_k]$. (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.)

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

Subcase I(a): There is (at least one) insert at the source (before se_k) that generates t. Let U_i be the last of these events. Insert U_i adds a tuple at the source that contains one of the keys involved in t. (Recall that t has a key value for each of the relations involved in the view definition.)

Note there can be no deletes involving a t key after U_i is processed at the source (at least not before or at se_k). If there were, they would remove t from $V[ss_k]$, a contradiction.

Under ECA^K, U_i gets propagated to the warehouse, a query Q is generated, answered at the source, 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 we_k. (If U_i was received after we_k, then $U_i \notin F$ and U_i must have been processed after se_k. If the Q answer arrived after we_k, then $UQS(we_k)$ would not have been empty.) 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 deletes affecting t at the source between the processing of U_i and se_k, there can be no such deletes at the warehouse, and t remains until we_k, a contradiction.

Subcase I(b): There are no inserts at the source that generate t. This means that t must have been in $V[ss_0]$, where ss_0 is the initial source state. Since the initial materialized view is correct, t must also be in $V[ws_0]$.

We can now proceed similarly to Case I(a), arguing that there are no deletes affecting t at the source (before se_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 $t \in V[ws_k]$.

Subcase II(a): There is at least one delete at the source that involves a t key value. The proof is analogous to Case I(a), with one extra case to consider. We consider the last such source delete U_d, where se_d is the processing of U_d at the source and we_d is the processing of U_d at the warehouse. Clearly, we_d deletes t from the materialized view. Since $t \in V[ws_k]$, it must have been re-inserted. Notice that there are no inserts that could generate t at the source after se_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* we_d, adding t. The query generated for U_j must have been processed at the source after se_d (else the answer would have arrived before we_d). Since the query is executed at the source after U_d is processed, it does not see one of the key values of t (i.e., the one deleted by U_d). Thus, there is no way the answer could contain t, and t is not added to the materialized view, 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 $V[ws_k]$, it must have been inserted by some answer received at the warehouse. Say se_j is the source event that generates the answer. At that point $V[state(se_j)]$ must contain t. Since there are no deletes affecting t at the source, then t remains, a contradiction. ∎

Theorem B.0.2 *Algorithm ECA^K is strongly consistent.* □

Proof: Given Lemma B.0.6, it is straightforward to see that ECA^K is strongly consistent. Since the materialized view at the warehouse only changes when $UQS = \phi$, the following three properties holds for any warehouse state ws_x: (1) From Lemma B.0.6, there exists a source state ss_x, such that $V[ss_x] = V[ws_x]$. (2) If we have two warehouse states, ws_y and ws_q, where $y < q$, then the source states they map to occur in this same order (because ss_i is the state after the corresponding update S_up_i occurred. (3) For a finite sequence of updates $U_1, U_2, \ldots, U_n$, since $V[ws_n] = V[ss_n]$, the algorithm converges. From (1), (2) and (3), we obtain that ECA^K is strongly consistent. ∎

Appendix C

Proof of Correctness for LCA and CCA

In this section we provide correctness proof for the Lazy Compensating Algorithm (LCA) and Complete Compensating Algorithm (CCA). The reason that we present these two proofs in the same section is because in both algorithms the same queries were sent to the source, the correctness proofs for both algorithms are based on the same set of lemmas and in nature the two algorithms are very similar.

The following Lemma C.0.7 is a generalization of Lemma A.0.2. It shows how a sequence of updates affects the evaluation of a query. Its intuitive meaning is that if a query Q is intended to be evaluated at the source state before a set of updates S occur, the correct answer can be obtained from the difference between the query answer evaluated after the occurrence of updates in S and the answer of a compensating query, $Q\langle S\rangle$. This property holds for all queries generated from an incrementally maintainable view.

Lemma C.0.7 *Let $S = \{U_{j,1}, U_{j,2}, \ldots, U_{j,n}\}$ be a set of updates that occurs in the order specified. Let $ss_{j,k}$ be the source state after update $U_{j,k}$ occurred and $ss_{j,0}$ be the source state before $U_{j,1}$ occurred. Then, $Q[ss_{j,0}] = Q[ss_{j,n}] - Q\langle S\rangle[ss_{j,n}]$ for any query Q.* □

Proof: We prove this lemma by induction on n.

1. Base case: When $n = 1$, $S = \{U_{j,1}\}$. From Lemma A.0.2, we have $Q[ss_{j,0}] = Q[ss_{j,1}] - Q\langle U_{j,0}\rangle[ss_{j,1}]$ for any query Q.

2. Induction step: Assume for n, we have

$$Q[ss_{j,0}] = Q[ss_{j,n}] - Q\langle S\rangle[ss_{j,n}] \tag{C.1}$$

Now we prove the lemma for n+1. Let $S' = \{U_{j,1}, U_{j,2}, \ldots, U_{j,n}, U_{j,n+1}\}$. We want to prove

$$Q[ss_{j,0}] = Q[ss_{j,n+1}] - Q\langle S'\rangle[ss_{j,n+1}] \tag{C.2}$$

By definition, we have

$$Q\langle S'\rangle = Q\langle S\rangle + Q\langle U_{j,n+1}\rangle - Q\langle S\rangle\langle U_{j,n+1}\rangle \tag{C.3}$$

From Lemma A.0.2, the following equation holds:

$$Q[ss_{j,n}] = Q[ss_{j,n+1}] - Q\langle U_{n+1}\rangle[ss_{j,n+1}] \tag{C.4}$$

Now we prove equation C.2:

$$\begin{aligned}
RHS &= Q[ss_{j,n+1}] - Q\langle S'\rangle[ss_{j,n+1}] \\
&= Q[ss_{j,n}] + Q\langle U_{j,n+1}\rangle[ss_{j,n+1}] - Q\langle S'\rangle[ss_{j,n+1}] && (Equation\ C.4) \\
&= Q[ss_{j,n}] + Q\langle U_{j,n+1}\rangle[ss_{j,n+1}] - Q\langle S\rangle[ss_{j,n+1}] - Q\langle U_{j,n+1}\rangle[ss_{j,n+1}] \\
&\quad + Q\langle S\rangle\langle U_{j,n+1}\rangle[ss_{j,n+1}] && (Equation\ C.3) \\
&= Q[ss_{j,n}] - Q\langle S\rangle[ss_{j,n+1}] + Q\langle S\rangle\langle U_{j,n+1}\rangle[ss_{j,n+1}] \\
&= Q[ss_{j,n}] - Q\langle S\rangle[ss_{j,n}] && (Equation\ C.4) \\
&= Q[ss_{j,0}] = LHS && (Equation\ C.1)
\end{aligned}$$

Therefore, we proved that Equation C.2 holds.

By induction, we have , $Q[ss_{j,0}] = Q[ss_{j,n}] - Q\langle S\rangle[ss_{j,n}]$ for any n. ∎

The second lemma shows that the total number of queries triggered by one update can not exceed n, which is the number of base relations in the view definition.

Lemma C.0.8 *Let the sequence of queries sent by LCA or CCA corresponding to update U_i be $Q_{i,0}, Q_{i,1}, \ldots, Q_{i,p}$, then $p \leq n-1$.* $\square$

Proof: From Section 3.2.2, each query is a sum of terms. Let $rel(t)$ be the set of relations used in a term t. Consider an update U. If $rel(U) \in rel(t)$, then $|rel(t\langle U\rangle)| = |rel(t)| - 1$; if $rel(U) \notin rel(t)$, then $t\langle U\rangle = \phi$ and $|rel(t\langle U\rangle)| = 0$. Therefore, we have $|rel(t\langle U\rangle)| \leq |rel(t)| - 1$.

Let the maximum of all $|rel(T_k)|$ for all terms T_k in query $Q_{i,j}$ $(0 \leq j \leq p)$ be M_i. By definition, for all $j > 0$, $Q_{i,j} = Q_{i,j-1}\langle S\rangle$ where S contains at least one update. Therefore, $M_i \leq M_{i-1} - 1$. Since the first query LCA or CCA sends is $Q_{i,0} = V\langle U_i\rangle$, $M_0 = n-1$. Therefore, $M_p \leq M_{p-1} - 1 \leq M_{p-2} - 2 \ldots \leq M_0 - p$. Since M_p is greater or equal to 0, we have $M_0 - p \geq 0$. That is, $p \leq M_0 = n-1$. Thus the sequence of compensating queries is finite and has length less than or equal to n. ∎

The next lemma shows that using LCA or CCA, all answers to queries sent corresponding to one update U_i together will add up to $Q[ss_i]$, which is the correct view delta corresponding to U_i.

Lemma C.0.9 *Using LCA or CCA, for any update U_i, $\sum_p A_{i,p} = Q_{i,0}[ss_i]$.* $\square$

Proof: Using LCA or CCA, query $Q_{i,0}$ is triggered by the receipt of update U_i, query $Q_{i,j+1} (j \geq 0)$ is triggered by the receipt of answer $A_{i,j}$. $Q_{i,j+1}$ is only sent when there is at least one new update between the sending of $Q_{i,j}$ and the receiving of $A_{i,j}$.

Let the list of queries sent corresponding to U_i be $Q_{i,0}, Q_{i,1}, \ldots, Q_{i,p}$. Let the corresponding source states when each query is sent be $ss_{i,0}, ss_{i,1}, \ldots, ss_{i,p}$ and the query answers be $A_{i,0}, A_{i,1}, \ldots, A_{i,p}$. The relationship of the queries, answers and states are shown in the following Table C.1. Notice that $ss_{i,j}$ is the source state that query $Q_{i,j}$ intends to be evaluated, $ss_{i,j+1}$ is the state that the query is actually evaluated. That is,

$$A_{i,j} = Q_{i,j}[ss_{i,j+1}] \tag{C.5}$$

Notice that since there is no updates between $ss_{i,p}$ and $ss_{i,p+1}$, we have $ss_{i,p} = ss_{i,p+1}$. That is, the last two queries are evaluated at the same source state (otherwise, the algorithms will send more compensating queries).

States	Events	Queries
$ss_i = ss_{i,0}$	U_i	$Q_{i,0}$
$ss_{i,1}$	$A_{i,0} = Q_{i,0}[ss_{i,1}]$	$Q_{i,1}$
...	...	...
$ss_{i,p}$	$A_{i,p-1} = Q_{i,p-1}[ss_{i,p}]$	$Q_{i,p}$
$ss_{i,p}$	$A_{i,p} = Q_{i,p}[ss_{i,p}]$	

Table C.1: Source states, events, queries and answers generated by LCA and CCA

Let S be the collection of updates occurred after $Q_{i,k}$ and before $A_{i,k}$. Then we have $Q_{i,k+1} = -Q_{i,k}\langle S\rangle$ following the algorithm. From Lemma C.0.7, we also have that for any Q,

$$Q[ss_{i,k}] = Q[ss_{i,k+1}] - Q\langle S\rangle[ss_{i,k+1}] \tag{C.6}$$

This gives us

$$Q_{i,k}[ss_{i,k}] = Q_{i,k}[ss_{i,k+1}] + Q_{i,k+1}[ss_{i,k+1}] \tag{C.7}$$

Now we prove Lemma C.0.9 by induction on p.

1. Base case: When $p = 0$, there is no compensating query sent corresponding to U_i. That means there is no other updates between $Q_{i,0}$ and $A_{i,0}$, and thus $Q_{i,0}$ is evaluated at state $ss_{i,1} = ss_{i,0}$. Therefore, $A_{i,0} = Q_{i,0}[ss_{i,1}] = Q_{i,0}[ss_{i,0}] = Q_{i,0}[ss_i]$

2. Induction step: Assume that for $p \geq 0$, $\sum_{k=0}^{k=p} A_{i,k} = Q_{i,0}[ss_i]$. That is, with one update query ($Q_{i,0}$) and $p - 1$ compensating queries, all the query answers together comprise the correct answer to the update query.

   Now we show that the same equation holds for one update query and p compensating queries. That is, we prove equation $\sum_{k=0}^{k=p+1} A_{i,k} = Q_{i,0}[ss_i]$. Since there are $p + 1$ queries, we have $ss_{i,p+1} = ss_{i,p+2}$. This gives us $A_{i,p+1} = Q_{i,p+1}[ss_{i,p+2}] = Q_{i,p+1}[ss_{i,p+1}]$. Therefore, we have

$$
\begin{aligned}
& \sum_{k=1}^{k=p+1} A_{i,k} \\
= & \sum_{k=0}^{k=p+1} Q_{i,k}[ss_{i,k+1}] && \text{(Equation C.5)} \\
= & \sum_{k=0}^{k=p-1} Q_{i,k}[ss_{i,k+1}] + Q_{i,p}[ss_{i,p+1}] + Q_{i,p+1}[ss_{i,p+2}] \\
= & \sum_{k=0}^{k=p-1} Q_{i,k}[ss_{i,k+1}] + Q_{i,p}[ss_{i,p+1}] + Q_{i,p+1}[ss_{i,p+1}] \\
= & \sum_{k=0}^{k=p-1} Q_{i,k}[ss_{i,k+1}] + Q_{i,p}[ss_{i,p}] - Q_{i,p}[ss_{i,p}] + Q_{i,p}[ss_{i,p+1}] + Q_{i,p+1}[ss_{i,p+1}] \\
= & \sum_{k=0}^{k=p} Q_{i,k}[ss_{i,k+1}] - Q_{i,p}[ss_{i,p}] + Q_{i,p}[ss_{i,p+1}] + Q_{i,p+1}[ss_{i,p+1}] \\
= & \sum_{k=0}^{k=p} Q_{i,k}[ss_{i,k+1}] && \text{(Equation C.7)} \\
= & \sum_{k=1}^{k=p} A_{i,k} && \text{(Equation C.5)} \\
= & Q_{i,0}[ss_i] && \text{(Inductive assumption)}
\end{aligned}
$$

By induction, we proved that $\sum_{k=0}^{k=p} A_{i,k} = Q[ss_i]$ for all p. ∎

Theorem C.0.3 *Algorithm CCA is complete.* □

Proof: We prove by showing that after `COLLECT[i]` is applied to MV, the content of MV, $V[ws_i]$ is the same as $V[ss_i]$. The proof is done by simple induction.

1. Base case: when $i = 0$, by assumption we have $V[ws_0] = V[ss_0]$.

2. Induction step:

   Using CCA, we have `COLLECT[i]` $= \sum_p A_{i,p}$. From Lemma C.0.9, we have $\sum_p A_{i,p} = Q_{i,0}[ss_i]$. Therefore, we have `COLLECT[i]` $= Q_{i,0}[ss_i]$ where $Q_{i,0} = V\langle U_i \rangle$.

   Assume for all $i \leq c$, we have $V[ws_i] = V[ss_i]$. When $i = c + 1$,

$$
\begin{aligned}
V[ws_{c+1}] &= V[ws_c] + \texttt{COLLECT[c + 1]} && (\textit{Algorithm CCA}) \\
&= V[ss_c] + \texttt{COLLECT[c + 1]} && (\textit{Indictive Assumption}) \\
&= V[ss_c] + \sum_p A_{c+1,p} && \text{(Algorithm CCA)} \\
&= V[ss_c] + Q_{c+1,0}[ss_{c+1}] && \text{(Lemma C.0.9)} \\
&= V[ss_c] + V\langle U_{c+1}\rangle[ss_{c+1}] && \text{(Algorithm CCA)} \\
&= V[ss_{c+1}] && \text{(Lemma A.0.2)}
\end{aligned}
$$

By induction, we have proved that $V[ws_i] = V[ss_i]$ for all i. Thus, CCA is complete. ∎

Theorem C.0.4 *Algorithm LCA is strongly consistent.* □

Proof: The prove is similar to that of CCA. For LCA, each time `COLLECT` is applied to the view, `COLLECT` contains the answers to all queries corresponding to a sequence of source updates. More specifically, if the algorithm applies `COLLECT` to MV after updates $l_1, l_2, \ldots, l_n$ are processed, then we show by induction that after processing update l_i, the view is updated to $V[ws_{l_i}] = V[ss_{l_i}]$. Let $l_0 = 0$.

1. Base case: when $i = 0$, by assumption we have $V[ws_{l_0}] = V[ws_0] = V[ss_0] = V[ss_{l_0}]$.

2. Induction step:

   Using LCA, after processing update l_i, `COLLECT` contains all answers to all queries corresponding to updates from $l_{i-1}+1$ to l_i. Let the content of `COLLECT` at this time be `COLLECT`$[l_i]$. That is, `COLLECT`$[l_i] = \sum_{k=l_{i-1}+1}^{k=l_i} \sum_p A_{k,p}$. From Lemma C.0.9, we have $\sum_p A_{i,p} = Q_{i,0}[ss_i]$ where $Q_{i,0} = V\langle U_i\rangle$.

   Assume for $i \leq c$, we have $V[ws_{l_i}] = V[ss_{l_i}]$. When $i = c + 1$,

$$\begin{aligned}
V[ws_{l_{c+1}}] &= V[ws_{l_c}] + \texttt{COLLECT}[\mathtt{l_{c+1}}] && (Algorithm LCA) \\
&= V[ss_{l_c}] + \texttt{COLLECT}[\mathtt{l_{c+1}}] && (Indictive Assumption) \\
&= V[ss_{l_c}] + \sum_{k=l_c+1}^{k=l_{c+1}} \sum_{p} A_{k,p} && \text{(Algorithm LCA)} \\
&= V[ss_{l_c}] + \sum_{k=l_c+1}^{k=l_{c+1}} Q_{k,0}[ss_k] && \text{(Lemma C.0.9)} \\
&= V[ss_{l_c}] + \sum_{k=l_c+1}^{k=l_{c+1}} V\langle U_k\rangle[ss_k] && \text{(Algorithm LCA)} \\
&= V[ss_{l_{c+1}}] && \text{(Lemma A.0.2)}
\end{aligned}$$

By induction, we have proved that $V[ws_{l_i}] = V[ss_{l_i}]$ for all l_i. Thus, LCA is strongly consistent. ∎

Appendix D

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 which is similar to the proof of ECA^K. 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.

Theorem D.0.5 *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 $UQS = \phi$. 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 $V[ss_k] = V[ws_x]$. Similar to the proof of Lemma B.0.6, 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 re-inserted. 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 $UQS = \phi$, 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. ∎

Appendix E

Proof of Correctness for C-Strobe Algorithm

We prove the completeness of C-Strobe by enumerating update scenarios and studying possible view contents.

Theorem E.0.6 *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 D, 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:

$$V[ss_{i-1}] \cup V\langle U_i \rangle[ss_{i-1}] = MV[ws_{i-1}] \cup Delta \tag{E.1}$$

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 (E.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 4.2.3, 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:

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

   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

$$V\langle U_i\rangle[ss_{i-1}] \subset Delta \Longrightarrow V[ss_{i-1}] \cup V\langle U_i\rangle[ss_{i-1}] \subset V[ss_{i-1}] \cup Delta$$

Also we have proved in Case II(b) that

$$\begin{aligned} &Delta \subset MV[ws_{i-1}] \cup V\langle U_i\rangle[ss_{i-1}] \\ &\qquad\qquad \Longrightarrow MV[ws_{i-1}] \cup Delta \subset MV[ws_{i-1}] \cup V\langle U_i\rangle[ss_{i-1}] \end{aligned}$$

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 (E.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. ∎

Appendix F

Proof of Correctness for Simple Painting Algorithm

The Simple Painting Algorithm (SPA) is used by the merge process to maintain MVC at the warehouse when all view managers are complete. SPA guarantees complete warehouse states. SPA receives action lists from view managers and REL_i sets from the integrator, and generates transactions for the warehouse database system. Now we prove the completeness of SPA.

Theorem F.0.7 *The Simple Painting Algorithm is complete under MVC.* □

Proof: An algorithm is complete if, when applied to an initially consistent warehouse state, it generates a complete warehouse state sequence.

Consider a set of f source updates that are processed by the view managers and coordinated by a merge process running SPA. The source updates are numbered by the integrator as $U_1, U_2 \ldots U_f = SS$. SS is a serial source schedule. The merge process changes the warehouse state by sending warehouse transactions. Using SPA, a warehouse transaction WT_i corresponds to one row in the VUT, and thus corresponds to one source update. So the total number of WTs is f. Suppose the sequence of warehouse transactions executed is $W = WT_{i_1}, \ldots, WT_{i_f}$, the corresponding warehouse state sequence is $W_{seq} = ws_{i_0}, ws_{i_1}, \ldots, ws_{i_f}$, where ws_{i_0} is the initial warehouse state. (Note that the sequence $i_1, i_2, \ldots$ may be different from the sequence $1, 2, \ldots$ because the merge process can reorder actions.) Corresponding to W, we construct a source execution schedule

$S = U_{i_1}, U_{i_2}, \ldots, U_{i_f}$ and source state sequence $S_{seq} = ss_{i_0}, ss_{i_1}, \ldots, ss_{i_f}$, where ss_{i_0} is the initial source state.

We prove the completeness of SPA in two steps. First, we show that S_{seq} is a *consistent* source state sequence. Then we prove that there exists a mapping m from W_{seq} to S_{seq}, where m satisfies all of the requirements of the mapping in the definition of completeness in Section 5.2.

We prove that S_{seq} is a consistent source state sequence by showing that schedules S and SS are equivalent. First, the two schedules involve the same set of updates. Second, let U_i and U_j be any two conflicting updates. U_i and U_j must be on the same base relation and hence affect the same view(s). If U_i precedes U_j in SS, then U_i must precede U_j in S, because WT_j will not be applied before WT_i (line 2 in $ProcessRow(i)$). Since the orders of all conflicting updates are the same in S and SS, the two schedules are equivalent[BHG87]. Therefore, S_{seq} is a consistent source state sequence.

We now define a mapping m from W_{seq} to S_{seq} as $m(ws_{i_k}) = ss_{i_k}$ for all $0 \leq k \leq f$. We show that m satisfies the mapping properties defined in Section 5.2. Initially, the warehouse state $ws_{i_0} \equiv ss_{i_0}$ by assumption. Because each view manager is complete and its actions applied in order, for each view V, the content of V at warehouse state ws_{i_k} is equivalent to $V(ss_{i_k})$. Therefore, we have $ws_{i_k} \equiv ss_{i_k}$ for all $0 \leq k \leq f$. It is easy to see that if $ws_i < ws_j$, then $m(ws_i) = ss_i < ss_j = m(ws_j)$. Also, for each source state ss_{i_k}, there exists ws_{i_k} such that $m(ws_{i_k}) = ss_{i_k}$. That is, the warehouse state sequence W_{seq} is complete. Since W_{seq} is any state sequence generated by a merge process running SPA, algorithm SPA is complete.

■

Appendix G

Properties of Complete Views

G.1 Proof of Correctness for Theorem 8.6.1

First we give two more SM rules for views with no `WHERE` clause in the definition. When there is no `WHERE` clause, $cond\_path = \lambda$ and $Y = X$.

Rule (b7) For a view with no `WHERE` clause, if the update is a modify, then Condition (b) is false and the update is maintenance irrelevant. $\diamond$

Rule (b8) For a view with no `WHERE` clause, if the update is an insert or a delete and $\exists tail\_path \in T$ such that $N_2.tail\_path \neq \phi$, then Condition (b) is true. $\diamond$

For Conditions (c) and (d), because $cond\_path = \lambda$, all $Y \in \mathbf{Y}$ are in Scenario 1 of Figure 8.2, where Y is a descendent object of N_2.

Theorem 8.5.1 describes those objects that are needed by the maintainer to maintain a complete view C.

Theorem 8.5.1 For a complete view $C = CV(view\_path)$ and an update U,

- If U is a modify, then U is maintenance irrelevant to C.
- If U is a delete, then U is self-maintainable to C given $C[D_{-U}]$, the view content before the update.
- If U is an insert, then U is maintainable with respect to C given $C[D_{-U}]$ and $CV(N_2, tail\_path)[D_{+U}]$ for all $tail\_path \in T$.

Proof: A complete view C contains all objects that are reachable from $ROOT$ via a prefix of *view_path*.

- When U is a modify: According to SM Rule (b12), all modifications are maintenance irrelevant to the view C because there is no `WHERE` clause in the view definition.
- When U is a delete: $U = delete(N_1, N_2)$
  - If $N_1 \notin C[D_{-U}]$, then either $path(ROOT, N_1) = \phi$, or $path(ROOT, N_1)$ is not a prefix of *view_path*. In both cases, Condition (a) is false and U is maintenance irrelevant to C.
  - If $N_1 \in C[D_{-U}]$ but $N_2 \notin C[D_{-U}]$, then for any path $p \in path(ROOT, N_1)$, $p.label(N_2)$ is not a prefix of *view_path* (otherwise, $N_2 \in C[D_{-U}]$). Therefore, Condition (a) is false and the update is maintenance irrelevant.
  - When both N_1 and N_2 are in $C[D_{-U}]$, if there does not exist $p \in path(ROOT, N_1)$ such that $p.label(N_2)$ is a prefix of *view_path*, then Condition (a) is false and U is irrelevant to view C. Otherwise, Condition (a) is true and we consider Condition (b). Consider all *tail_path*s in T. If $N_2.tail\_path = \phi$, then Condition (b) is false and the update is maintenance irrelevant. Otherwise, all objects in $N_2.tail\_path$ were in $C[D_{-U}]$, because before removing the link from N_1 to N_2, they can all be reached from $ROOT$ by a prefix of *view_path*. Therefore Condition (b) is true.

    Let $S = CV(N_2, tail\_path)$ for all $tail\_path \in T$. From SM rule (b13), we have $\mathbf{Y} = S$, so all objects in $\mathbf{Y}$ can be obtained from $C[D_{-U}]$ and Condition (c) is true. Some objects in S may need to be removed from C due to this deletion U. If after removing N_2 from N_1, an object $O \in S$ can still be reached from $ROOT$ using a prefix of *view_path*, then O stays in the view. Otherwise, we claim that O cannot be reached from $ROOT$ via a prefix of *view_path* even considering objects outside of $C[D_{-U}]$, because if there is such a path, all objects along that path should be in $C[D_{-U}]$. Therefore in the second case, O should be removed from the view. In both cases, whether O should stay can be decided from $C[D_{-U}]$. So Condition (d) is true and the view is self-maintainable.

  Therefore, for delete, it is possible for the maintainer to obtain the new view given $C[D_{-U}]$.

- When U is an insert: $U = insert(N_1, N_2)$

  We show that $C[D_{-U}] \cup CV(N_2, tail\_path)[D_{+U}]$ (for all $tail\_path \in T$) is $C[D_{+U}]$, which is the new view content after the update.

  Let LHS $= C[D_{-U}] \cup CV(N_2, tail\_path)[D_{+U}]$ (for all $tail\_path \in T$).

  - We first prove that if $O \in C[D_{+U}]$, then $O \in LHS$. If O was in the view C before the update, then $O \in C[D_{-U}]$ and thus $O \in LHS$. If $O \notin C[D_{-U}]$ but $O \in C[D_{+U}]$, then O much be reachable from $ROOT$ through a path p passing through object N_1 and N_2 (otherwise $O \in C[D_{-U}]$), and p is a prefix of $view\_path$. That is, there exists a prefix of $view\_path$, path p, such that $p \in path(ROOT, N_1).label(N_2).path(N_2, O)$. Let $p = p_1.label(N_2).p_3$ where $p_1 \in path(ROOT, N_1)$ and $p_3 \in path(N_2, O)$. Because p is a prefix of $view\_path$, p_3 must be a prefix of $tail\_path$. Since $p_3 \in path(N_2, O)$, object O can be reached from N_2 following a prefix of path $tail\_path$, thus, $O \in CV(N_2, tail\_path)[D_{+U}]$. This proofs that $O \in LHS$.
  - Now we prove that if $O \in LHS$, $O \in C[D_{+U}]$. For insert, all object in $C[D_{-U}]$ are in $C[D_{+U}]$. If $O \in CV(N_2, tail\_path)[D_{+U}]$, by the definition of $tail\_path$, we have $view\_path \in path(ROOT, N_1).label(N_2).tail\_path$. Therefore, O can be reached from $ROOT$ via a prefix of $view\_path$. So $O \in C[D_{+U}]$.

Thus for insert, it is possible for the maintainer to obtain the new view content from LHS. That is, the view is maintainable given LHS.

∎

G.2 Proof of Correctness for Theorem 8.6.2

Theorem 8.5.2 For a view V defined by expression (7.4.7) and an update U,

- If U is a modify or a delete, $CV(view\_path)[D_{-U}]$ contains all objects sufficient and necessary to maintain view V with respect to U.
- If U is an insert, $CV(view\_path)[D_{+U}]$ contains all objects sufficient and necessary to maintain view V with respect to U.

Proof:
Sufficient: Theorem 8.5.1 shows that $C = CV(view\_path)$ is maintainable given $C[D_{-U}]$ for a delete or a modify, and given $C[D_{+U}]$ for an insert. For a view V defined with the same *view_path* as C, objects need to be examined in evaluating SMT Conditions (a) and (b) for view V are the same as those needed for view C. Objects need to be examined in evaluating SMT Conditions (c) and (d) for V are a subset of those needed for view C. Thus, given $C[D_{-U}]$ is enough to maintain V for delete or modify, and $C[D_{+U}]$ is enough to maintain V for insert.
Necessary: We show that if one of the objects specified by Theorem 8.5.2 is unknown to the maintainer, then for some updates, V is not maintainable. Since source updates can change the base database in an arbitrary way, if a view were to be maintainable with respect to all source updates, it has to be maintainable given any base database.

First we consider delete and modify. Say one object W in $C[D_{-U}]$ is unknown to the maintainer. W is reachable from $ROOT$ following a prefix of *view_path*. Let this path be p_w, that is, $p_w \in path(ROOT, W)$ and $view\_path = p_w.p_t$. We can construct a base database D', such that in D', there exists $X \in W.p_t$, such that $sel\_cond(X) = true$. Because of this X, there exists at least one Y on the path from $ROOT$ to X (passing W) such that Y appears in the view.

Now let there be an update $U = delete(W, Z)$. Remember that no additional information other than the OIDs of the updated objects are available to the maintainer. Now the maintainer is unable to evaluate Condition (b), because the maintainer does not know whether Z is on $path(W, X)$ and if it is, whether there are still links from W to Z after removing one link from W. (There can be multiple links between two objects in a graph-structured database.) Since the result of Condition (b) is unavailable, the view is not maintainable with respect to update U.

Similarly, if the value of X is modified and $sel\_cond(X)$ is now false, the maintainer can not decide whether there is another X' in $W.p_t$ such that $sel\_cond(X') = true$, because the maintainer does not know the outgoing edges from W. Therefore, Condition (b) is unavailable and the view is not maintainable with respect to this modify.

From the proof of Theorem 8.5.1, we have $C[D_{+U}] = C[D_{-U}] \cup CV(N_2, tail\_path)[D_{+U}]$ (for all $tail\_path \in T$). For $insert(N_1, N_2)$, if an object W in $C[D_{-U}]$ is unknown to the maintainer, then let D' be the database in which there is no path from $ROOT$ to N_1 other than possibly passing through W. For D', it is impossible to decide whether there is a path from $ROOT$ to N_1 passing W, so the result of Condition (a) is unavailable. If an object W in $CV(N_2, tail\_path)[D_{+U}]$ is missing, then let D' be the database where excluding all paths to and from W, there is no such object $X \in N_2.tail\_path$ and $sel\_cond(X) = true$. For this database, the maintainer is impossible to test Condition (b), and the view is not self-maintainable.

∎

Bibliography

[AASY97] D. Agrawal, A. E. Abbadi, A. K. Singh, and T. Yurek. Efficient view maintenance at data warehouses. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 417–427, May 1997.

[AB91] S. Abiteboul and A. Bonner. Objects and views. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 238–247, May 1991.

[ABGM90] 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, September 1990.

[AGM+97] S. Abiteboul, R. Goldman, J. McHugh, V. Vassalos, and Y. Zhuge. Views for semistructured data. In *Proceedings of the Workshop on Management of Semistructured Data*, pages 83–90, May 1997.

[AM+98] S. Abiteboul, J. McHugh, , M. Rys, V. Vassalos, and J. Wiener. Incremental maintenance for materialized views over semistructured data. In *Proceedings of the 24th International Conference on Very Large Data Bases (VLDB)*, pages 38–49, August 1998.

[AQM+97] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, April 1997.

[BCL89] J.A. Blakeley, N. Coburn, and P.-A. Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Transactions on Database Systems*, 14(3):369–400, September 1989.

[BCP96] E. Baralis, S. Ceri, and S Paraboschi. Conservative timestamp revised for materialized view maintenance in a data warehouse. In *Proceedings of the Workshop on Materialized Views, Techniques and Applications*, pages 1–9, June 1996.

[BDHS96] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 505–516, June 1996.

[BDK92] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Building an Object-Oriented Database System: The Story of O2*. Morgan Kaufmann, 1992.

[BGMS92] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of multidatabase transaction management. *VLDB Journal*, 1(2):181–239, October 1992.

[BHG87] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[BK97] Y. Breitbart and H.F. Korth. Replication and consistency: Being lazy helps sometimes. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 173–184, May 1997.

[BLT86] J.A. Blakeley, P.-A. Larson, and F.W. Tompa. Efficiently updating materialized views. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 61–71, June 1986.

[Cat94] R.G.G. Cattell, editor. *The Object Database Standard: ODMG–93.* Morgan Kaufmann, 1994.

[CB94] M. Cochinwala and J. Bradley. A multidatabase system for tracking and retrieval of financial data. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*, pages 714–721, 1994.

[CD92] S. Cluet and C. Delobel. A general framework for the optimization of object-oriented queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 383–392, June 1992.

[CGL+96] L.S. Colby, T. Griffin, L. Libkin, I.S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 469–480, June 1996.

[CJ98] Felipe Carino and Mark Jahnke. Bank of america: Case study - the information currency advantage. In *Proceedings of the 24th International Conference on Very Large Data Bases (VLDB)*, pages 641–644, August 1998.

[CJS+94] 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.

[CM97] R. Chen and W. Meng. Efficient view maintenance in a multidatabase environment. In *Database Systems for Advanced Applications (DASFAA)*, pages 391–400, April 1997.

[CW91] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the 17th International Conference on Very Large Data Bases (VLDB)*, pages 577–589, September 1991.

[DDJ+98] L. Do, P. Drew, W. Jin, V. Jumani, and D. V. Rossum. Issues in developing very large data warehouses. In *Proceedings of the 24th International Conference on Very Large Data Bases (VLDB)*, pages 633–636, August 1998.

[GB95] A. Gupta and J.A. Blakeley. Using partial information to update materialized views. *Information Systems*, 20(8):641–662, November 1995.

[GBLP96] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In *Proceedings of Twelfth IEEE International Conference on Data Engineering*, pages 152–159, 1996.

[GHOS96] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The danger of replication and a solution. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 173–182, June 1996.

[GHRU97] H. Gupta, V. Harinarayan, A. Rajaraman, and J. Ullman. Index selection for OLAP. In *Proceedings of the IEEE International Conference on Data Engineering*, 1997.

[GJM96] A. Gupta, H. Jagadish, and I. Mumick. Data integration using self-maintainable views. In *International Conference on Extending Database Technology (EDBT)*, pages 140–144, March 1996.

[GL95] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 328–339, May 1995.

[GM95] A. Gupta and I.S. 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.

[GM96] A. Gupta and I. Mumick, editors. *Proceedings of the workshop on materialized views: techniques and applications*, June 1996.

[GM98] A. Gupta and I. Mumick, editors. *Materialized Views*. MIT Press, 1998.

[GMS93] A. Gupta, I. Mumick, and V. Subrahmanian. Maintaining views incrementally. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 157–166, May 1993.

[GMW82] H. Garcia-Molina and G. Wiederhold. Read-only transactions in a distributed database. *ACM Transaction on Database Systems*, 7(2):209–234, June 1982.

[GN95] R. Gallersdorfer and M. Nicola. Improving performance in replicated databases through relaxed coherency. In *Proceedings of the 21st International Conference on Very Large Data Bases (VLDB)*, pages 445–456, September 1995.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[GW93] A. Gupta and J. Widom. Local verification of global integrity constraints in distributed databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 49–58, May 1993.

[Han87] E.N. Hanson. A performance analysis of view materialization strategies. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 440–453, 1987.

[HD92] J.V. Harrison and S.W. 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.

[HGMW+95] J. Hammer, H. Garcia-Molina, J. Widom, W. Labio, and Y. Zhuge. The Stanford Data Warehousing Project. *IEEE Data Engineering Bulletin, Special Issue on Materialized Views and Data Warehousing*, 18(2):41–48, June 1995.

[HRU96] V. Harinarayan, A. Rajaraman, and J.D. Ullman. Implementing data cubes efficiently. In *Proceedings of ACM SIGMOD 1996 International Conference on Management of Data*, pages 205–216, 1996.

[Huy96] N. Huyn. Efficient view self-maintenance. In *Proceedings of the Workshop on Materialized Views, Techniques and Applications*, pages 17–25, June 1996.

[HY90] R. Hull and M. Yoshikawa. Ilog: Declarative creation and manipulation of object identifiers. In *Proceedings of the 16th International Conference on Very Large Data Bases (VLDB)*, pages 455–468, August 1990.

[HZ96a] R. Hull and G. Zhou. A framework for supporting data integration using the materialized and virtual approaches. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 481–492, June 1996.

[HZ96b] R. Hull and G. Zhou. Towards the study of performance trade-offs between materialized and virtual integrated views. In *Proceedings of the Workshop on Materialized Views, Techniques and Applications*, pages 91–102, June 1996.

[IK93] W.H. Inmon and C. Kelley. *Rdb/VMS: Developing the Data Warehouse.* QED Publishing Group, 1993.

[JBL86] N. Coburn J.A. Blakeley and P.-A. Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. In *Proceedings of the 12th International Conference on Very Large Data Bases (VLDB)*, pages 457–466, August 1986.

[KKS92] M. Kifer, W. Kim, and Y. Sagiv. Querying object oriented databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 393–402, June 1992.

[KLM+97] A. Kawaguchi, D. Lieuwen, I. Mumick, D. Quass, and K. Ross. Concurrency control theory for deferred materialized views. In *Proceedings of the International Conference on Database Theory (ICDT)*, 1997.

[KS95] D. Konopnicki and O. Shmueli. W3qs: A query system for the world wide web. In *Proceedings of the 21st International Conference on Very Large Data Bases (VLDB)*, pages 54–65, September 1995.

[LGM96] W. Labio and H. Garcia-Molina. Efficient snapshot differential algorithms in data warehousing. In *International Conference on Very Large Data Bases (VLDB)*, pages 63–74, September 1996.

[LHM+86] B. Lindsay, L.M. Haas, C. Mohan, H. Pirahesh, and P. Wilms. A snapshot differential refresh algorithm. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, May 1986.

[LMR90] W. Litwin, L. Mark, and N. Roussopoulos. Interoperability of multiple autonomous databases. *ACM Computing Surveys*, 22(3):267–293, September 1990.

[LQA97] W. Labio, D. Quass, and B. Adelberg. Physical database design for data warehousing. In *Proceedings of the 13th International Conference on Data Engineering (ICDE)*, pages 277–288, April 1997.

[LW95] D. Lomet and J. Widom, editors. *Special Issue on Materialized Views and Data Warehousing,* IEEE Data Engineering Bulletin 18(2), June 1995.

[Mic98] Microsoft Corporation, Personal investment management using XML. Available from http://www.microsoft.com/xml, 1998.

[OV91] M.T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, 1991.

[PAGM96] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object fusion in mediator systems. In *Proceedings of the 22nd International Conference on Very Large Data Bases (VLDB)*, pages 414–424, September 1996.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proceedings of the 11th International Conference on Data Engineering (ICDE)*, pages 251–260, March 1995.

[QGMW96] D. Quass, A. Gupta, I. Mumick, and J. Widom. Making views self-maintainable for data warehousing. In *Proceedings of the 4th International Conference on Parallel and Distributed Information Systems (PDIS)*, pages 158–169, December 1996.

[QRUW95] D. Quass, A. Rajaraman, J. Ullman, and J. Widom. Querying semistructured heterogeneous information. In *Proceedings of the Fourth International Conference on Deductive and Object-Oriented Databases*, pages 319–344, December 1995.

[Qua96] D. Quass. Maintenance expressions for views with aggregation. In *Proceedings of the ACM Workshop on Materialized Views: Techniques and Applications*, pages 110–118, 1996.

[QW91] X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *IEEE Transactions on Knowledge and Data Engineering*, 3(3):337–341, September 1991.

[QW97] D. Quass and J. Widom. On-line warehouse view maintenance. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 393–404, 1997.

[RK86] N. Roussopoulos and H. Kang. Preliminary design of ADMS+-: A workstation-mainframe integrated architecture for database management systems. In *Proceedings of the 12th International Conference on Very Large Data Bases (VLDB)*, pages 355–364, August 1986.

[RSS96] K. A. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 447–458, June 1996.

[Run92] E. Rundensteiner. Multiview: A methodology for supporting multiple views in object-oriented databases. In *Proceedings of the 18th International Conference on Very Large Data Bases (VLDB)*, pages 187–198, August 1992.

[SC97] U. Dayal S. Chaudhuri. An overview of data warehousing and OLAP technology. *SIGMOD Record*, 26(1):65–74, March 1997.

[SF90] A. Segev and W. Fang. Currency-based updates to distributed materialized views. In *Proceedings of the 6th International Conference on Data Engineering (ICDE)*, pages 512–520, February 1990.

[SF91] A. Segev and W. Fang. Optimal update policies for distributed materialized views. *Management Science*, 37(7):851–70, July 1991.

[SI84] O. Shmueli and A. Itai. Maintenance of views. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 240–255, May 1984.

[SJ96] M. Staudt and M. Jarke. Incremental maintenance of externally materialized views. In *Proceedings of the 22nd International Conference on Very Large Data Bases (VLDB)*, pages 75–86, September 1996.

[SP89a] A. Segev and J. Park. Maintaining materialized views in distributed databases. In *Proceedings of the 5th International Conference on Data Engineering (ICDE)*, pages 262–70, February 1989.

[SP89b] A. Segev and J. Park. Updating distributed materialized views. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):173–184, June 1989.

[Suc96] D. Suciu. Query decomposition and view maintenance for query language for unstructured data. In *Proceedings of the 22nd International Conference on Very Large Data Bases (VLDB)*, pages 227–238, September 1996.

[Suc97] D. Suciu, editor. *Proceedings of the Workshop on Management of Semistructured Data*, May 1997.

[Syb92] Sybase, Inc. *Command Reference Manual*, release 4.9 edition, 1992.

[TB88] F.W. Tompa and J.A. Blakeley. Maintaining materialized views without accessing base data. *Information Systems*, 13(4):393–406, November 1988.

[WGL+96] J.L. Wiener, H. Gupta, W.J. Labio, Y. Zhuge, H. Garcia-Molina, and J. Widom. A system prototype for warehouse view maintenance. In *Proceedings of the Workshop on Materialized Views, Techniques and Applications*, pages 26–33, June 1996.

[WQ90] 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, November 1990.

[ZGM98a] Y. Zhuge and H. Garcia-Molina. Graph structured views and their incremental maintenance. In *Proceedings of the 14th International Conference on Data Engineering (ICDE)*, pages 116–125, February 1998.

[ZGM98b] Y. Zhuge and H. Garcia-Molina. Performance analysis of WHIPS. Technical report, Stanford University, May 1998. http://www-db.stanford.edu/pub/papers/zg-whipsperf.ps.

[ZGM99] Y. Zhuge and H. Garcia-Molina. Self-maintainability of graph-structured views. Technical report, Stanford University, February 1999. http://www-db.stanford.edu/pub/papers/zgself-full.ps.

[ZGMHW94] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. Technical report, Stanford University, October 1994. http://www-db.stanford.edu/pub/papers/anomaly-full.ps.

[ZGMHW95] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 316–327, May 1995.

[ZGMW96] Y. Zhuge, H. Garcia-Molina, and J.L. Wiener. The Strobe algorithms for multi-source warehouse consistency. In *Proceedings of the 4th International Conference on Parallel and Distributed Information Systems (PDIS)*, pages 146–157, December 1996.

[ZGMW98] Y. Zhuge, H. Garcia-Molina, and J.L. Wiener. Consistency algorithms for multi-source warehouse view maintenance. *Distributed and Parallel Databases*, 6(1):7–40, January 1998.

[ZWGM97] Y. Zhuge, J. L. Wiener, and H. Garcia-Molina. Multiple view consistency for data warehousing. In *Proceedings of the 13th International Conference on Data Engineering (ICDE)*, pages 289–300, April 1997.