EFFICIENT MAINTENANCE AND RECOVERY OF 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

Wilburt Juan Labio

August 1999

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

Hector Garcia-Molina (Principal Advisor)

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

Dallan Quass

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

Jennifer Widom

Approved for the University Committee on Graduate Studies:

Abstract

Data warehouses collect data from multiple remote sources and integrate the information as *materialized views* in a local database. The materialized views are used to answer queries that analyze the collected data for patterns, anomalies, and trends. This type of query processing is often called *on-line analytical processing* (OLAP). So that OLAP queries can be posed and answered easily, the data from the remote sources is "cleansed" and translated to a common schema.

The warehouse views must be updated when changes are made to the remote information sources. Otherwise, the answers to OLAP queries are based on stale data. Answering OLAP queries based on stale data is clearly a problem especially if (answers to) OLAP queries are used to support critical decisions made by the organization that owns the data warehouse. Because the primary purpose of the data warehouse is to answer OLAP queries, only a limited amount of time and/or resources can be devoted to the warehouse update. Hence, we have developed new techniques to ensure that the warehouse update can be done efficiently.

Also, the warehouse update is not devoid of failures. Since only a limited amount of time and/or resources are devoted to the warehouse update, it is most likely infeasible to restart the warehouse update from scratch. Thus, we have developed new techniques for resuming failed warehouse updates.

Finally, warehouse updates typically transfer gigabytes of data into the warehouse. Although the price of disk storage is decreasing, there will be a point in the "lifetime" of a data warehouse when keeping and administering all of the collected is unreasonable. Thus, we have investigated techniques for reducing the storage cost of a data warehouse by selectively "expiring" information that is not needed.

Acknowledgments

This thesis is dedicated to my beloved wife Lena. Without her patience, love, caring and encouragement, it would have not been possible for me to finish my degree. This thesis took five years to finish. Lena has sacrificed so much and has been with me every step of the way. She was my inspiration in each page of this thesis.

Five years really does not seem that long because my advisor, Hector Garcia-Molina, has made each day worthwhile. Hector taught me how to do good research and how to have fun at the same time. I thank Hector for his tutelage, support, and for just being the "dream" advisor I had hoped for coming into Stanford.

I thank my parents, Gregorio and Purisima, and my brother, Nathaniel, for their eternal support. Whenever I was down, they were there for encouragement. Whenever I was up, they were there to enjoy the fruits of my labor with me.

I thank my co-authors especially Brad Adelberg, Dallan Quass, Ramana Yerneni and Jun Yang. I have learned much from each of them.

I thank my co-implementors of WHIPS: Claire Cui, Himanshu Gupta, Jun Yang, and Yue Zhuge. It has been fun cramming for those demos.

I thank Jeff Ullman and Jennifer Widom. What I learned from their classes was a solid foundation of my research.

I thank Arturo Crespo – for opening my eyes to the beauty of nature; Narayanan Shivakumar – for making me realize that things are easier than what they seem to be; and Tom Schirmer – for the enjoyable lunches, dinners and afternoon visits.

Finally, I thank God. I believe nothing good comes from man without the blessing of God . . . and I believe this thesis is good!

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Many organizations collect vast amounts of information about their activities. For instance, a large retail store (*e.g.*, WalMart) typically collects gigabytes of point-of-sales data per month [Car97]. The same retail store probably collects other types of information as well, such as customer data, inventory data, advertisement data, employee data, etc. An increasing number of organizations are realizing that the vast amounts of collected data can and must be used to guide their business decisions [Inm96]. Typically, the management of the organization wants to answer complex *analytical queries* (*e.g.*, "What is the average revenue for each product category?") based on the collected data. However, answering these queries by accessing the organization's various data sources poses the following problems.

- The data sources are distributed across the organization. Hence, answering the analytical queries can be expensive since distributed data sources need to be accessed.
- The data sources are not optimized to handle complex analytical queries. For instance, inventory data sources are on-line systems that process fairly simple queries (*e.g.*, "Insert a new order," "Find the last order of product `X`").
- The data sources are not centrally administered and may have inconsistencies. For instance, addresses may have different formats in the various sources. In general, the data from the various sources must be "cleansed" and made consistent to answer the analytical queries.

To alleviate these problems, the *data warehousing* architecture has been proposed (*e.g.*, [Inm92]). In this architecture, the information from the various data sources is integrated

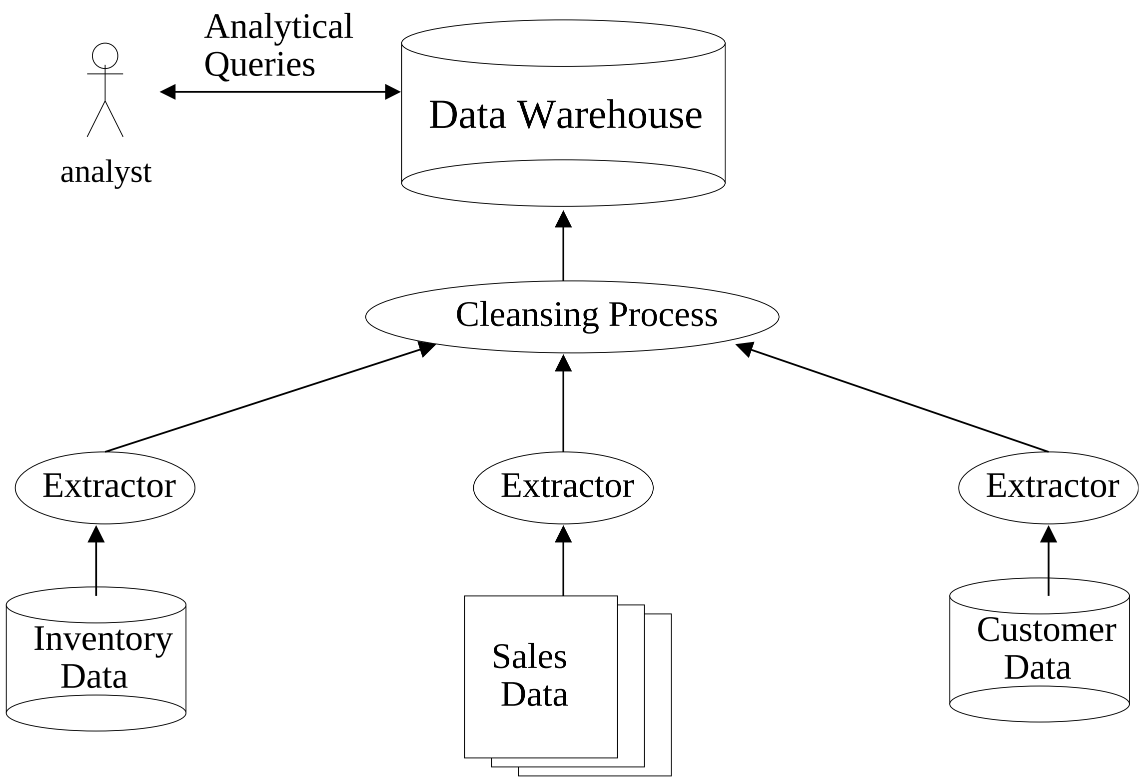

Figure 1.1: The Data Warehousing Architecture

into a central database (Figure 1.1). Custom *extractors* are created to obtain the data from the various sources. The extracted data is then *cleansed* to resolve any inconsistencies. The cleansed data from the various sources is then filtered and integrated. We call the process that performs the cleansing and the integration of the source data the *cleansing process*. The cleansed and integrated data is then entered into the data warehouse. The extraction and cleansing of the source data is done in advance of the queries to minimize the processing required at query time.

Building a data warehouse provides a number of benefits.

- The processing of analytical queries is simplified because only the data warehouse needs to be accessed. As mentioned, the data extraction and the data cleansing are done a priori. Also, additional structures can be built in the data warehouse to further improve the efficiency of query processing. For instance, indices as well as summaries of the warehouse data can be created. Many analytical queries can be processed more efficiently by accessing summary data instead of the unsummarized or "detail" data [LMSS95, SDJL96].

- The warehouse data can keep a historical record of the various source data. By retaining all of this data, the current activity of an organization can be compared against history. Historical data can also be used for forecasting the future activities of an organization.

Numerous corporations have built data warehouses because of these benefits. However, once a data warehouse is successfully created, managing and maintaining the data warehouse is very challenging. We outline these challenges in Section 1.1. In this thesis, we provide new techniques that will make the design, the deployment and the maintenance of a data warehouse more efficient and manageable. We also describe a research prototype that integrates and implements many of the solutions proposed.

1.1 Research Issues

In order to successfully maintain a data warehouse, a number of interesting research issues need to be resolved. For a survey, see [CD97, Wid95]. We outline a few of the interesting research areas, and then describe the specific problems addressed in this thesis.

- *Efficient Maintenance of a Data Warehouse.* When the underlying data sources change, the warehouse data needs to be updated to reflect the changes. (Otherwise, the answers to analytical queries are based on stale warehouse data.) Since the primary purpose of a data warehouse is to answer analytical queries, only a limited amount of time and computing resources can be allocated to the update. (Typically, the warehouse update is performed offline, *i.e.*, during the night, or during the weekend when the query load is low.) Since updating the warehouse involves complex processing of large amounts of data, it is a challenge to finish the update during the allocated time. To answer this challenge, developing and integrating numerous techniques is required. More specifically, efficient change detection algorithms are required at the sources [LGM96], and these algorithms need to be combined with algorithms that efficiently compute and install the changes to the warehouse data [MQM97].

- *Lowering the Storage Requirements of a Data Warehouse.* Every time a data warehouse is created or updated, gigabytes or even terabytes of data are loaded into the data warehouse [JMS95]. Even though disk space keeps getting cheaper, the high cost of administering numerous disks makes it impractical to keep all of the warehouse data. Since some of the warehouse data may be accessed more often than other data, selecting the appropriate data to compress, archive, or simply remove is an important issue [GMLY98].

- *Answering Analytical Queries Efficiently.* Answering analytical queries may still take hours even though it only requires accessing the data warehouse. In order to improve the query processing, selecting the appropriate summary data and indices to create is crucial [Gup97]. Once the appropriate summary data is selected, algorithms that translate analytical queries to use the summary data are required [SDJL96]. It may also be acceptable and more efficient to compute approximate answers to analytical queries as opposed to computing exact answers [AGPR99].

- *Recovery of the Warehouse Load.* When data is loaded into the warehouse, a significant amount of time is spent on cleansing the extracted data. Because the cleansing process involves complex processing of large amounts of data, it is not devoid of failures. Unfortunately, the cleansing process is outside the control of the back-end database of the data warehouse [Sag98]. Thus, in practice, we cannot rely on the back-end database to recover failed warehouse loads. In order to avoid "redoing" the entire cleansing process in case of failures, it is crucial to develop recovery algorithms that can resume a warehouse load that failed during the cleansing process.

Before we describe the specific problems we address in this thesis, we introduce the warehouse model that we work with. We also introduce some notation that will be used throughout the thesis.

1.2 Overview of Warehouse Creation and Update

Warehouse Creation

Conceptually, a data warehouse is created using the following steps.

1. Extraction of source data.

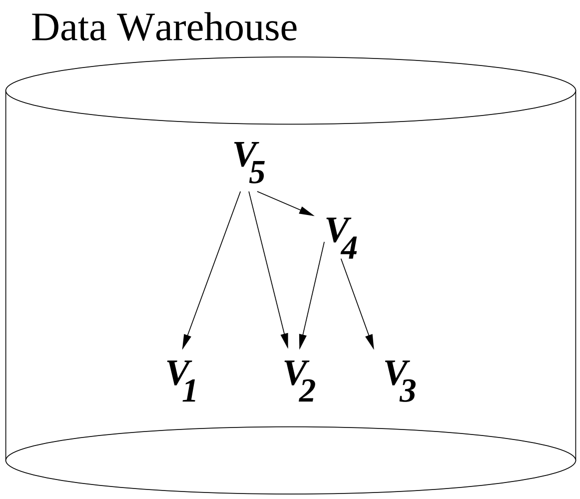

Figure 1.2: A Simple VDAG Example

2. Cleansing of extracted data.

3. Materialized view initialization.

We now discuss each step in turn.

When a warehouse is created, data is extracted from the various *remote data sources* and is used to initialize the warehouse. We assume the warehouse data is stored in a relational database, referred to as the *warehouse database* for conciseness. We do allow duplicate tuples in any relation, supporting *bag* semantics [Ull88]. Since the remote data sources may not be relational, the extractors are responsible for transforming the data into the relational model. (See [UW97] for algorithms for transforming data of a different data model into the relational model.) Hence, whether the remote data source is a relational database, an IMS database or a flat file, the extractor outputs the source data in the relational model. That is, for each remote data source, the extractor obtains a set of tables $\{ T \}$, and each table T is a bag of tuples.

The extracted data is then cleansed by a *cleansing process* (see Figure 1.1). In Chapter 6, we will show examples of typical operations involved in the cleansing process.

The cleansed data is used to initialize the warehouse data. The warehouse data is modeled using a *view directed acyclic graph* (VDAG) as shown in Figure 1.2. Each node V in the VDAG represents a *materialized view* containing warehouse data. A materialized view V is a query over a set of tables from the sources and/or over other materialized views, whose result is computed and stored in the data warehouse. The query is called the *definition* of the materialized view V. The definition of V involves projection, selection, join, and aggregation operations, and corresponds to `SELECT-FROM-WHERE-GROUPBY` SQL queries. Although this thesis deals with a restricted form of view definition (*e.g.*, `HAVING` clauses are not considered), we believe the view definition considered is sufficiently general.

An edge ($V_j \rightarrow V_i$) in the VDAG indicates that the definition of view V_j refers to view V_i (*i.e.*, V_i is in the `FROM` clause of V_j's definition). If a view V has no outgoing edges, this indicates that V is defined over remote data sources. For simplicity, we assume that a view V is defined only over remote data sources, or only over views at the warehouse. We call views defined over remote data sources *base views*, and views defined over other views (at the warehouse) *derived views*. Base views are defined using `SELECT-FROM-WHERE` SQL queries over source tables. Although we do consider a powerful base view definition language, base views are often defined to simply copy source tables. Derived views are defined using `SELECT-FROM-WHERE-GROUPBY` SQL queries over other warehouse views.

The cleansed data is first inserted into the base views. The derived views are then initialized (in topological order) by submitting their definition queries to the warehouse database.

In today's warehouses, the base view data is often obtained by extracting and cleansing on-line transaction processing (OLTP) source data. The resulting base views are often called "dimension tables" or "fact tables." The fact tables are also often called "detail data." Derived views, often called "summary tables," are defined over the base views to summarize the detail data. Note however that derived views, as we have defined above, are more general than summary tables which typically join dimension and fact tables, and aggregate the result of the join. Analytical or On-Line Analytical Processing (OLAP) queries are posed in terms of the warehouse views, and are answered by simply submitting the query to the warehouse database.

Warehouse Update

When the data at the remote sources changes (*e.g.*, there are new sales), the materialized views in the warehouse need to be updated to reflect the source changes. One strategy in updating the data warehouse is to rebuild the warehouse from scratch. However, usually incremental maintenance of the warehouse is much more efficient. In this thesis, we focus on developing techniques for incremental warehouse updates.

Conceptually, a data warehouse is (incrementally) updated using the following steps.

1. Extraction of source data changes.

2. Cleansing of extracted changes.

3. Materialized view maintenance.

The warehouse update can occur immediately after the source data changes. We call this the *immediate* warehouse update. In today's warehouses, the changes to the remote sources are often not detected and propagated to the warehouse immediately for two reasons. First, the remote sources are often heavily-loaded OLTP sources that allot the extractor a short time window to detect changes. Thus, when a source change does happen, it may not occur within the allotted time window for detecting changes. Second, even after the changes have been detected, data warehouse resources are only available to compute and install the deltas to the warehouse views when the query load at the warehouse is low. Hence, it is important to support a *deferred* warehouse update wherein changes at the sources are extracted when the administrators of the data sources indicate it is appropriate to do so. Furthermore, the detected changes are then propagated periodically when the query load at the warehouse is low. The contributions we make in this thesis can support deferred warehouse update as well as immediate warehouse update.

We now discuss each step of the warehouse update in turn. During the warehouse update, the extractors are used to detect the changes of the remote sources. For each table T obtained from each remote source by the extractors during warehouse creation, the extractor detects the *insertions* to T (denoted $\triangle T$), the *deletions* to T (denoted $\bigtriangledown T$), and the *updates* to T (denoted $\mathbin{\rlap{\triangle}\blacktriangle} T$).

The detected deletions and insertions are then cleansed by the cleansing process. The cleansing process for the warehouse update is probably different from the cleansing process used for warehouse creation. In particular, the cleansing process may use advanced algorithms ([ZGMHW95]) that ensure that the changes to the base views are computed consistently given the changes detected from the remote data sources. Assuming such algorithms are in place, the output of the cleansing process is a set of correct base view changes.

The changes of the base views are not applied directly to the base views. Instead they are computed and stored in *delta tables*. We assume that $\triangle V$ contains the insertions to V, and $\bigtriangledown V$ contains the deletions to V, and $\triangle\!\!\!\!\triangle V$ contains the updates to V. Employing delta tables allows standard *view maintenance expressions* to be used to *compute* the changes of the the derived views. That is, if V is a derived view, a view maintenance expression based on V's definition is used to compute $\triangle V$, $\bigtriangledown V$, and $\triangle\!\!\!\!\triangle V$. The warehouse update must also *install* the changes computed for the warehouse views. That is, the tuples in $\triangle V$ are inserted into V, and the tuples in $\bigtriangledown V$ are deleted from V, and the tuples in $\triangle\!\!\!\!\triangle V$ are updated.

Warehouse Load

Finally, we use the term *warehouse load* to refer to the process of warehouse creation or warehouse update. That is, whenever source data is loaded into the warehouse, the process is called a warehouse load.

1.3 Contributions and Thesis Outline

In this thesis, we focus on: (1) reducing the warehouse update cost; (2) lowering the warehouse storage requirement; and (3) developing algorithms for recovering the warehouse load in case of failures. We now give an overview of our contributions in these three areas.

Efficient Change Detection (Chapter 2)

The first step in the warehouse update is change detection. Since many data sources allot a short time for the extractor to detect changes, change detection must be done as efficiently as possible. Unfortunately, as we will discuss in Chapter 2, the extractor is often tasked to detect changes by comparing large "snapshots" of the source data. We reduce the problem of change detection given the snapshots to performing an outerjoin between the old and new snapshots. Although the outerjoin operation can detect all and only the changes, performing an outerjoin can be too expensive. We develop approximate change detection algorithms that can be six times more efficient than performing an outerjoin. These algorithms may miss detecting updated source tuples or may report unchanged source tuples as both deleted and inserted. However, we show that in practice, the approximate change detection algorithms will most likely detect all the changed tuples.

Efficient View Maintenance (Chapter 3)

Once the remote source changes are detected and cleansed, the changes to the warehouse views need to be efficiently computed and installed (*i.e.*, the third step of the warehouse update). In Chapter 3, we show that there are numerous "strategies" for performing the view maintenance step of the warehouse update. We then develop efficient algorithms that pick good strategies. We prove that under a reasonable cost model, the strategies picked are optimal for each individual view, and are optimal for a large class of VDAGs. We show experimentally that the resulting maintenance time using the strategies picked can be five times shorter than the maintenance time of the conventional strategies.

Choosing Additional Structures to Improve Warehouse Update (Chapter 4)

In Chapter 4, we show that the warehouse update can be improved further by creating additional indices and views. The problem then is to choose the appropriate combination of indices and views to add. We show in Chapter 4 that there are too many choices to consider and an exhaustive algorithm is infeasible. We develop an algorithm based on A* search that picks the optimal combination of indices and views, but prunes as many as 99% of the choices. Because of the enormity of the search space, this algorithm may still be too slow. Hence, we develop greedy algorithms that may pick sub-optimal combinations but are much more efficient than the A*-based algorithm.

Selective Archiving of Warehouse Data (Chapter 5)

Because of recent developments in query answering using views, it is reasonable to assume that most of the OLAP queries can be answered using only the derived views. That is, the base views are used mostly for computing changes to derived views, but rarely for answering OLAP queries. In Chapter 5, we develop a constraint language that can be used over the base views (as well as derived views). We show that the language can describe many types of constraints such as append-only, referential integrity, and key constraints. Using the declared constraints, we develop an algorithm that identifies base view tuples that will never be used in computing the changes of the derived views. Under the assumption that most OLAP queries can be answered using derived views, these base view tuples can be archived, resulting in a significant reduction of storage cost. For instance, we show that with appropriate append-only constraints, close to 100% of typical detail data can be archived. Although summary data can be archived also using the same method, we predict that archiving detail data will suffice in reducing storage requirements since most data in the warehouse is detail data.

Recovery Algorithms for the Cleansing Process (Chapter 6)

The cleansing process of the warehouse load (*i.e.*, the second step of warehouse creation and warehouse update) is often done outside the control of the warehouse database. Since the cleansing process is not devoid of failures, recovery algorithms are required to avoid redoing the entire cleansing process in case of failures. As we will discuss in Chapter 6, developing a recovery algorithm is difficult because, for efficiency reasons, intermediate results of the cleansing process are often not saved in stable storage. Even if some intermediate results are saved, it is still very difficult to deduce what processing does not need to be redone. The reason for this difficulty is that the cleansing process is quite complex and may involve user-defined transformations of the extracted data. In Chapter 6, we develop a recovery algorithm called *DR* that avoids redoing the entire cleansing process in case of failures. *DR* does not rely on any low-level details of the cleansing process, but instead relies on high-level properties of the transformations used in the cleansing process. We show that *DR* can be much better than redoing the cleansing process, or dividing the cleansing process into stages and redoing incomplete stages. We also develop a hybrid algorithm that combines *DR* with staging.

The WHIPS Prototype (Chapter 7)

We present the WHIPS (WareHouse Information Processing at Stanford) prototype in Chapter 7. WHIPS is not a new database system. It is a distributed application that runs on top of the database, and that manages view maintenance. In Chapter 7, we discuss the design decisions that were made in developing WHIPS. We show through experiments why the design decision made were appropriate. These decisions provide guidelines for anyone developing data warehouse management software that runs on top of a database.

The algorithms and techniques developed in this thesis build on previous work in view maintenance, rule condition maintenance, database recovery, and query processing. We discuss related work in the relevant chapters.

Chapter 2

Detecting Source Changes

2.1 Introduction

In this chapter, we focus on the detection and extraction of changes to the remote data sources. The detection and extraction of changes depends on the facilities at the source. If the source is sophisticated, say a relational database system with triggers, then this process is relatively easy. In many cases, however, the source does not have advanced facilities available for detecting changes (*e.g.*, legacy sources), and there are essentially three ways to detect and extract changes [IC94]:

1. The application running on top of the source is altered to send the changes to the warehouse.

2. A system log file is parsed to obtain the relevant changes (as done in the IBM Data Propagator [Gol95]). Since log files are used for recovery, this approach may not require any modification to the application.

3. The changes are inferred by comparing a current source snapshot with an earlier one. Typically, the snapshots used are the same ones generated for backup, so this approach may not require modification to the application either. We call the problem of detecting differences between two source snapshots the *snapshot differential* problem; it is the problem we address in this chapter.

Although the first two methods are usually preferred, they do have limitations and disadvantages. The first method requires that existing code be altered. In most cases the code is so "shopworn" that additional modifications are problematic. Since the changes are recorded as they happen, this method also entails extra processing on top of normal operations. The second method also has its difficulties. For instance, it is often the case that DBA (database administrator) privileges are required to access the log, so site administrators are reluctant to provide access. Moreover, log files often have a format that is hard to decipher and DBMS vendors are usually not willing to disclose the format. It may also be the case that the source does not even have (or need) a log. The third method is used in practice when the other methods do not apply. Some commercial products, such as the Prism Warehouse Manager [IC94], provide support for all three methods. However, as far as we know, there are no published papers detailing the algorithms used by commercial systems.

We stress that we are *not* arguing in favor of snapshot differentials as the best solution for reporting changes to a warehouse. It clearly does not scale well: as the volume of source data grows, we have to perform larger and larger comparisons. We are saying, however, that it is a solution we are stuck with for the foreseeable future (until sophisticated database systems become universal), and because differentials are such inherently expensive operations it is absolutely critical that we perform them as efficiently as possible. In this chapter we will present very efficient differential algorithms; they perform well because they exploit the fact that the semantics of the problem permits certain flexibility as discussed below.

2.1.1 Problem Formulation

The snapshot differential problem can be formulated as the *set differential* problem or the *bag differential* problem. Henceforth, snapshot differential refers to both the set and bag differential problems.

Set Differential

For the set differential problem, we view a source snapshot as a file containing a *set* of distinct records. The file is of the form $\{R_1, R_2, \ldots, R_n\}$ where R_i denote a record. Each R_i is of the form $\langle K, B\rangle$, where K is the key and B is the rest of the record representing one or more fields. Each record has a unique key value. Without loss of generality, we refer to B as a single field in the rest of the chapter.

For the set differential problem we have two snapshots, F_1 and F_2 (the later snapshot). Our goal is to produce a file F_{OUT} that also has the form $\{R_1, R_2, \ldots, R_n\}$ where each record R_i has one of the following three forms.

1. $\langle Update, K_i, B_j \rangle$
2. $\langle Delete, K_i \rangle$
3. $\langle Insert, K_i, B_i \rangle$

The first form is produced when a record $\langle K_i, B_i \rangle$ in file F_1 is updated to $\langle K_i, B_j \rangle$ in file F_2. The second form is produced when a record $\langle K_i, B_i \rangle$ in F_1 does not appear in F_2. Lastly, the third form is produced when a record $\langle K_i, B_i \rangle$ in F_2 was not present in F_1. We refer to the first form as *updates*, the second as *deletes* and the third as *inserts*. The first field is only necessary in distinguishing between updates and inserts. It is included for clarity in the case of deletes.[1]

Note that the key attribute K is used for finding "matching" records to produce update reports. Let us suppose that record $\langle K_i, B_j \rangle$ is in F_2, and record $\langle K_i, B_i \rangle$ is in F_1. Because the key values of the two records are the same, we assume that the record $\langle K_i, B_j \rangle$ in F_2 was the record $\langle K_i, B_i \rangle$ in F_1 except that its B attribute may have been updated. In the unlikely scenario that the key attribute is updated, the update is reported using a delete and an insert.

It is important to realize that there is no unique report of changes that captures the difference between two snapshots. At one extreme, a deletion can be reported for each record in F_1 and an insertion can be reported for each record in F_2. Obviously, this type of reporting can be wasteful. A record $\langle K_i, B_i \rangle$ in F_1 that is also in F_2 is reported as both deleted and inserted when no report is necessary. Also, if $\langle K_i, B_i \rangle$ is in F_1, and $\langle K_i, B_j \rangle$ is in F_2, a more concise update report is sufficient. (In Chapter 4, we show experimentally the importance of reporting updates.) In either case, we call the delete and insert report a *useless pair* since either no report is necessary, or a more concise update report suffices. More formally, a useless pair is a message sequence in one of the following two forms.

1. $\langle Delete, K_i, B_i \rangle, \ldots, \langle Insert, K_i, B_j \rangle$
2. $\langle Insert, K_i, B_j \rangle, \ldots, \langle Delete, K_i, B_i \rangle$

[1] In some applications, we may also want to filter out some changes that we know in advance not to be of interest to the warehouse (*e.g.*, only cancer patient data is collected at the warehouse). However, for simplicity, we assume that all of the changes are relevant to the warehouse.

Note that B_i and B_j may be the same. The first form is called a *useless delete-insert* pair, while the second form is called a *useless insert-delete* pair.

A useless insert-delete pair may introduce a correctness problem for set differentials. As an example, suppose the warehouse maintains a copy of the source relation represented by the snapshots. Upon receiving an insert report, the view maintenance algorithm will attempt to insert the record into the copy of the source relation at the warehouse. It will most likely be ignored since a record with the same key already exists. Thus, when the delete is processed, the record with the key K_i will be deleted from the warehouse. On the other hand, a useless delete-insert pair (which is composed of the opposite sequence) does not compromise the correctness of the warehouse. However, it introduces overhead in processing messages.

Since having useless pairs is not an effective way of reporting changes, one may be tempted to require set differential algorithms to generate *no* useless pairs. However, strictly forbidding useless delete-insert pairs turns out to be counterproductive! Allowing the generation of "some" useless delete-insert pairs gives the differential algorithm significant flexibility and leads to solutions that can be very efficient in some cases. We return to these issues later when we quantify the savings of "flexible" differential algorithms over algorithms that do not allow useless delete-insert pairs. Thus, in this chapter we do allow useless delete-insert pairs, with the ultimate goal of keeping their numbers relatively small.

For set differentials, we do want to avoid useless insert-delete pairs since they may compromise correctness. Useless insert-delete pairs can be eliminated by recording the changes detected in a file. A second pass can then be performed over the file to eliminate the useless pairs altogether. Since the size of the file is probably much smaller than the snapshots, the second pass will not be too expensive. We assume for the rest of the chapter that all useless insert-delete pairs are eliminated by the method just outlined.

Finally, it is important to observe that using

$$F_2 \ - \ F_1,$$

to find insertions, and

$$F_1 \ - \ F_2,$$

to find deletions can produce useless pairs. That is, all of the updated records are reported using useless pairs. Furthermore, the above strategy requires two minus ($-$) operations. The algorithms we propose in this chapter are much more efficient. Also, most of the algorithms are guaranteed to report updated records using update reports instead of useless pairs.

Bag Differential

For the bag differential problem, we view a source snapshot as a file containing a *bag* of records. The file is of the form $\{R_1, R_2, \ldots, R_n\}$ where R_i denotes a record. Since the file is a bag, there may be duplicate records and the snapshots do not have key attributes. Recall that key attributes were used in set differentials to detect "matching" records in the two snapshots. That is, it was assumed that the record $\langle K_i, B_j\rangle$ in F_2 was the record $\langle K_i, B_i\rangle$ in F_1 except that its B attribute may have been updated. By using the key attributes, updated records can be more easily found.

Since bags do not have keys, one may be tempted to formulate the bag differential as reporting all the records in F_2 that are not in F_1 as inserts, and all the records in F_1 that are not in F_2 as deletes. We can then use

$$F_2 \dot{-} F_1,$$

to find the inserted records, and

$$F_1 \dot{-} F_2,$$

to find the deleted records. The operation $F_1 \dot{-} F_2$ removes m copies of a record R from F_1 if there are m copies of R in F_2, and there are at least m copies of R in F_1. If F_1 has n copies of R where $n < m$, all of the n copies of R are removed from F_1.

The problem with the above strategy is that all of the updated records are reported using useless pairs. (Useless pairs are defined for bag differentials shortly.) Furthermore, the above strategy requires two expensive operations ($\dot{-}$).

Although bags do not have keys, often there are attributes we call the *identification* attributes that have the following properties.

1. The number of records that have a specific identification attribute I value, say I_1, is small. Since I is not a key attribute, it is not guaranteed that the number of records with an I_j value is either one or zero.

2. The I attributes are not updated often.

Identification attributes can be found given statistics on the domain sizes of the various attributes, and statistics on how often certain attributes are updated.

To illustrate, let us suppose we have a bank database. In the database, there is a table recording the withdrawals. The table has the attributes *accountID* and *time* to record the account from which money is withdrawed from, and the time of the withdrawal. Strictly speaking, these two attributes do not constitute a key of the table. For instance, a particular account may belong to more than one person, and two or more of the owners may withdraw money at the same time. Clearly, each withdrawal will most likely have a unique combination of *accountID* and *time* values.

Just like key attributes, identification attributes are useful in detecting updated records. That is, given a record in $\langle I_i, B_i \rangle$ in F_1, the update report can be produced by examining the records in F_2 with an I_i identification attribute value.

Hence, we view each R_i to be of the form $\langle I, B \rangle$, where I is the identification attribute and B is the rest of the record representing one or more fields. As in the set differential problem, we refer to B as a single field without loss of generality.

For the bag differential problem we have two snapshots, F_1 and F_2 (the later snapshot). Our goal is to produce a file F_{OUT} that also has the form $\{R_1, R_2, \ldots, R_n\}$ and each record R_i has one of the following three forms.

1. $\langle Update, I_i, B_i, B_j \rangle$
2. $\langle Delete, I_i, B_i \rangle$
3. $\langle Insert, I_i, B_i \rangle$

Note that the report forms for updates and deletes in the bag differential are different from the report forms for updates and deletes in the set differential. The report form for inserts is unchanged. Record $\langle Insert, I_i, B_i \rangle$ still means that a record $\langle I_i, B_i \rangle$ was inserted into F_2. We provide the intuition behind the report forms for updates and deletes in the next example.

EXAMPLE 2.1.1 Let us suppose that F_1 and F_2 are as shown in Figure 2.1. Note that snapshot F_1 has duplicate records, and both snapshots have records with the same identification attribute values. Since F_1 has two $\langle I_1, B_1 \rangle$ records, while F_2 has only one $\langle I_1, B_1 \rangle$ record, a $\langle I_1, B_1 \rangle$ record was deleted. However, reporting this delete as $\langle Delete, I_1 \rangle$ is ambiguous because it could mean that one of the $\langle I_1, B_1 \rangle$ records in F_1 was deleted, or that the $\langle I_1, B_{10} \rangle$ record in F_1 was deleted. Hence, the delete must be reported as $\langle Delete, I_1, B_1 \rangle$,

F1	**F2**
< I1, B1 >	**< I1, B1 >**
< I1, B1 >	
< I1, B10 >	**< I1, B10 >**
< I2, B2 >	**< I2, B3 >**
< I2, B20 >	**< I2, B20 >**

Figure 2.1: Example F_1 and F_2 for Bag Differentials

specifying that one of the $\langle I_1, B_1 \rangle$ records was deleted. If both $\langle I_1, B_1 \rangle$ records were deleted, then there would be two $\langle Delete, I_1, B_1 \rangle$ records in the bag differential.

The record $\langle I_2, B_2 \rangle$ in F_1 was also updated to $\langle I_2, B_3 \rangle$. Reporting this update as $\langle Update, I_2, B_3 \rangle$ is ambiguous because it could mean that $\langle I_2, B_2 \rangle$ or $\langle I_2, B_{20} \rangle$ was updated to $\langle I_2, B_3 \rangle$. Thus, the update must be reported as $\langle Update, I_2, B_2, B_3 \rangle$ specifying that a $\langle I_2, B_2 \rangle$ record was updated to $\langle I_2, B_3 \rangle$. □

As the example illustrated, a delete report $\langle Delete, I_i, B_i \rangle$ specifies that one of the $\langle I_i, B_i \rangle$ records in F_1 was deleted. An update report $\langle Update, I_i, B_i, B_j \rangle$ specifies that one of the $\langle I_i, B_i \rangle$ records in F_1 was updated to $\langle I_i, B_j \rangle$ in F_2. Finally, an insert report $\langle Insert, I_i, B_i \rangle$ specifies that a record $\langle I_i, B_i \rangle$ was inserted into F_2.

A useless insert-delete pair is defined similarly for the bag differential problem. That is, it is a message sequence composed of $\langle Insert, I_i, B_i \rangle$ followed (not necessarily immediately) by $\langle Delete, I_i, B_j \rangle$. If B_i is the same as B_j, no changes were needed to be reported for the record $\langle I_i, B_i \rangle$. On the other hand, if B_j is different from B_i, then the change could have been reported more succinctly by $\langle Update, I_i, B_i, B_j \rangle$. Note that for bag differentials, a useless insert-delete pair does not introduce any correctness problem. A useless delete-insert pair is similar to a useless insert-delete pair except that the delete report comes before the insert report. In this chapter, most of the bag differential algorithms we develop will not report useless pairs.

For bag differentials, unnecessary updates can also be reported. For instance, let us suppose F_1 has the single record $\langle I_1, B_1 \rangle$, while F_2 has two records $\langle I_1, B_2 \rangle$ and $\langle I_1, B_1 \rangle$. Clearly, the record $\langle I_1, B_1 \rangle$ is unchanged while the record $\langle I_1, B_2 \rangle$ was inserted. However, the difference between F_1 and F_2 can also be reported as $\langle I_1, B_1 \rangle$ being updated to $\langle I_1, B_2 \rangle$ (*i.e.*, $\langle Update, I_1, B_1, B_2 \rangle$), and $\langle I_1, B_1 \rangle$ being inserted (*i.e.*, $\langle Insert, I_1, B_1 \rangle$). Hence, an insertion plus an update is reported when a single insert report suffices. More formally, an unnecessary update is a message sequence in one of the following four forms.

1. $\langle Update, I_i, B_i, B_j \rangle, \ldots, \langle Insert, I_i, B_i \rangle$
2. $\langle Insert, I_i, B_i \rangle, \ldots, \langle Update, I_i, B_i, B_j \rangle$
3. $\langle Update, I_i, B_j, B_i \rangle, \ldots, \langle Delete, I_i, B_i \rangle$
4. $\langle Delete, I_i, B_i \rangle, \ldots, \langle Update, I_i, B_j, B_i \rangle$

The change reported by the first two sequences is more concisely reported as $\langle Insert, I_i, B_j \rangle$. The change reported by the last two sequences is more concisely reported as $\langle Delete, I_i, B_j \rangle$. In this chapter, we develop algorithms that avoid reporting unnecessary updates.

Why Ignore The Record Location?

Conceptually, we have represented snapshots as sets or bags because the physical location of a record within a snapshot file may change from one snapshot to another. That is, records with matching keys are not expected to be in the same physical position in F_1 and F_2 because the source is free to reorganize its storage between snapshots. Also, insertions and deletions may also change physical record positions in the snapshot.

2.1.2 Differences with Joins

The set differential problem is closely related to the problem of performing a join between two relations. In particular, if we join F_1 and F_2 on the key attribute K and on the condition that their B attributes differ, we can obtain the update records required for the set differential problem. However, the join does not capture the unmatched deleted and inserted records. An outerjoin, however, can generate the inserts and deletes, although the resulting records will not be in the desired format (they will have all fields of both relations, some with null values).

Using an outerjoin to perform bag differentials is further complicated by the presence of duplicates. For instance, if F_1 has two $\langle I_1, B_1 \rangle$ records that are both updated to $\langle I_1, B_2 \rangle$, the outerjoin will produce four update records of the form $\langle Update, I_1, B_1, B_2 \rangle$. Clearly, only two update records are required.

Still, join and outerjoin are so closely related to the snapshot differential problem that the traditional join algorithms ([ME92],[HC94]) can be adapted to our needs. Indeed, in Section 2.2 we show these modifications. However, given the particular semantics and intended application of the differential algorithms, we can go beyond the join-based solutions and obtain new and more efficient algorithms. The three main ideas we exploit are as follows:

- As discussed earlier, some useless delete-insert pairs are acceptable. In the context of outerjoins, a useless delete-insert pair is equivalent to "reporting" two records as "dangling" when they actually have matching keys. Traditional outerjoin algorithms do not have useless delete-insert pairs. The extra flexibility we have allows algorithms that are "sloppy" (but very efficient) in matching records.

- For some data warehousing applications, it may be acceptable to miss a few of the changes, especially if these "errors" are very infrequent. For example, if the warehouse is used for statistical analysis or data mining, missing one sales record out of billions may be acceptable. Thus, for differentials we can use probabilistic algorithms that may miss some differences (with arbitrarily low probability), but that can be much more efficient. Again, traditional algorithms are not allowed any "errors," must be very conservative, and must pay the price.

- Snapshot differentials are an on-going process running at a source (or intermediate source). Since snapshot differentials are an on-going process, it possible to save some of the information used in one differential to improve the next iteration. Traditional join algorithms typically do not take advantage of data structures created during other joins (other than existing general purpose indexes).

2.1.3 Outline

The rest of the chapter is organized as follows. We first present how the join algorithms can be extended to perform snapshot differentials in Sections 2.2.1 and 2.2.2. We present the record compression techniques to reduce snapshot size in Section 2.2.3 and show how these techniques can be used with the outerjoin algorithms in Sections 2.2.4 and 2.2.5. In Section 2.3, we introduce our *window* algorithm, representing a second class of efficient snapshot differential algorithms. The algorithms are analytically compared in Section 2.4.1; we report on the implementation and evaluation of some of the algorithms in Section 2.4.2. Section 2.5 briefly reviews related research in the literature. We summarize the chapter in Section 2.6.

2.2 Using Compression

In this section we first describe existing join algorithms but we do not cover all the known variations and optimizations of these algorithms. We believe that many of these further optimizations can also be applied to the snapshot differential algorithms we present.

We first extend the join algorithms to handle set differentials (Section 2.2.1) and to handle bag differentials (Section 2.2.2). After extending the join algorithms to handle the snapshot differential problem, we study compression techniques to optimize them in Sections 2.2.3 to 2.2.5. In the sections below, we denote the size of a file F as $|F|$ blocks and the size of main memory as $|M|$ blocks. We also exclude the cost of writing the output file in our cost analysis since it is the same for all of the algorithms.

2.2.1 Set Differentials using Outerjoins

The basic sort merge join first sorts the two input files. It then scans the files once and any pair of records that satisfy the join condition are produced as output. The algorithm can be adapted to perform an outerjoin by identifying the records that do not join with any records in the other file during the scan. The algorithm can be adapted with no extra cost when two records are being matched: the record with the smaller key is guaranteed to have no matching records.

Since set differentials are an on-going process running at a source, it is possible to save the sorted file of the previous snapshot. Thus, the algorithm only needs to sort the second file, F_2. This sorting can be done using the multiway merge-sort algorithm. This algorithm constructs runs which are sequences of blocks with sorted records. After a series of passes, the file is partitioned into progressively longer runs. The algorithm terminates when there is only one run left. In general, it takes $2 \cdot |F| \cdot log_{|M|}|F|$ IO operations to sort a file with size $|F|$ ([Ull89a]). However, if there is enough main memory ($|M| > \sqrt{|F|}$), the sorting can be done in $4 \cdot |F|$ IO operations (sorting is done in two passes). The second phase of the algorithm, which involves scanning and merging the two sorted files, entails $|F_1| + |F_2|$ IO operations for a total of $|F_1| + 5 \cdot |F_2|$ IO operations.

The IO cost can be reduced further by just producing the sorted runs (denoted as $F_{2\ runs}$) in the first phase. This improved algorithm, called *SM*, is shown in Figure 2.2. Line 1 produces the sorted F_2 runs, at a cost of only $2 \cdot |F_2|$ IOs. (File F_1 has already been sorted at this point.) The sorted F_2 file, needed for the next run of the algorithm, can then

Algorithm 2.2.1 *SM*
Input $F_{1\ sorted}$, F_2
Output F_{out} (the set differential), $F_{2\ sorted}$
Method

1. $F_{2\ runs} \leftarrow SortIntoRuns(F_2)$
2. $r_1 \leftarrow$ read the next record from $F_{1\ sorted}$
3. $r_2 \leftarrow$ read the next record from $F_{2\ runs}$; $F_{2\ sorted} \leftarrow Output(\langle r_2.K,\ r_2.B\rangle)$
4. **while** $((r_1 \neq NULL) \wedge (r_2 \neq NULL))$
5.   **if** $((r_1 = NULL) \vee (r_1.K > r_2.K))$ **then**
6. $F_{out} \leftarrow Output(\langle Insert,\ r_2.K,\ r_2.B\rangle)$
7. $r_2 \leftarrow$ read the next record from $F_{2\ runs}$; $F_{2\ sorted} \leftarrow Output(\langle r_2.K,\ r_2.B\rangle)$
8.   **else if** $((r_2 = NULL) \vee (r_1.K < r_2.K)$ **then**
9. $F_{out} \leftarrow Output(\langle Delete,\ r_1.K\rangle)$
10. $r_1 \leftarrow$ read the next record from $F_{1\ sorted}$
11.   **else if** $(r_1.K = r_2.K)$ **then**
12.   **if** $(r_1.B \neq r_2.B)$ **then**
13. $F_{out} \leftarrow Output(\langle Update,\ r_2.K,\ r_2.B\rangle)$
14. $r_1 \leftarrow$ read the next record from $F_{1\ sorted}$
15. $r_2 \leftarrow$ read the next record from $F_{2\ runs}$; $F_{2\ sorted} \leftarrow Output(\langle r_2.K,\ r_2.B\rangle)$

$\diamond$

Figure 2.2: Sort Merge Outerjoin as a Set Differential Algorithm

be produced while matching $F_{2\ runs}$ with F_1. In producing the sorted F_2 file (Lines 3, 7, 15), we read into memory one block from each run in $F_{2\ runs}$ (if the block is not already in memory), and select the record with the smallest K value. The merge process (Lines 4 through 15) now costs $2 \cdot |F_2| + |F_1|$ IOs. Thus, when sort merge outerjoin is used as a snapshot differential algorithm, the total cost incurred is $|F_1| + 4 \cdot |F_2|$ IOs.

Another join method that we discuss here is the partitioned hash outerjoin algorithm. In the partitioned hash outerjoin algorithm, the input files are partitioned into buckets by computing a hash function on the join attribute. Records are matched by considering each pair of corresponding buckets. First, one of the buckets is read into memory (the smaller one) and an in-memory hash table is built (assuming the bucket fits in memory). The second bucket is then read and a probe into the in-memory hash table is made for each record in an attempt to find a matching record in the first bucket. Matching records are merged and produced as output.

We now obtain the IO cost formula for the partitioned hash algorithm. Creating the buckets incurs $2 \cdot |F_1| + 2 \cdot |F_2|$ IOs and the matching phase and merging phase incur $|F_1| + |F_2|$ IOs, assuming the buckets fit in memory. This assumption has a main memory requirement of $|M| > \sqrt{min(|F_1|, |F_2|)}$. If the buckets do not fit in memory, additional repartitioning needs to be done. In general the IO cost is $2 \cdot log_N(|F_1|/|M|) \cdot (|F_1| + |F_2|)$ with repartitioning (where N is the number of buckets) [Gra93]. For the rest of the analysis, we assume that the both buckets do fit in memory. In a similar manner to the sort merge outerjoin algorithm, the buckets of the later snapshot can be saved for the next snapshot differential process. Thus the total IO cost incurred is $|F_1| + 3 \cdot |F_2|$ since only the second snapshot needs to be partitioned into buckets.

The partitioned hash join algorithm can be modified easily to perform set differentials. The first phase is unchanged and is still used to partition both snapshots into buckets by computing a hash function on the key attribute. In the second phase, each pair of corresponding buckets (denoted B_{F_1} and B_{F_2}) is processed. Assuming both buckets fit in memory, both are read into memory for processing. For each record R_1 in B_{F_1}, the record R_2 (if any) in B_{F_2} with a key of $R_1.K$ is found. If $R_1.B$ and $R_2.B$ are different, the appropriate update report is produced. Otherwise, no report is necessary. In either case, both R_1 and R_2 are removed from the two buckets (in memory) once they are matched. After all of the records in B_{F_1} are processed, the remaining records in B_{F_1} are reported as deletes. The remaining records in B_{F_2} are reported as inserts. It is easy to see that the IO cost of the partitioned hash outerjoin algorithm is not altered with this modification (given that the two buckets fit in memory). To reduce the processing cost, an index on the key attribute(s) can be constructed. This way, given a record R_1 from B_{F_1}, the record R_2 in B_{F_2} with a key of $R_1.K$ can be found in $O(log\ n)$ time, assuming there are n records in B_{F_2}.

2.2.2 Bag Differentials using Outerjoins

If the sort merge outerjoin algorithm *SM* (Figure 2.2) is used to compute bag differentials, it can fail to match records that have not changed and report unnecessary updates. For instance, let us suppose that F_1 has the single record $\langle I_1, B_1 \rangle$, and that F_2 has two records $\langle I_1, B_2 \rangle$ and $\langle I_1, B_1 \rangle$ appearing in that sequence. That is, record $\langle I_1, B_1 \rangle$ is unchanged and record $\langle I_1, B_2 \rangle$ was inserted. The sort merge outerjoin algorithm would match $\langle I_1, B_1 \rangle$ of F_1 with the first record $\langle I_1, B_2 \rangle$ of F_2. An update report $\langle Update, I_1, B_1, B_2 \rangle$ is then be produced. (Although *SM* actually produces $\langle Update, I_1, B_2 \rangle$, it can be modified easily to produce the appropriate update report for the bag differential.) The second record $\langle I_1, B_1\rangle$ is then reported as an insert. Clearly, we can avoid unnecessary updates by identifying that $\langle I_1, B_1\rangle$ is unchanged and reporting $\langle I_1, B_2\rangle$ as an insert.

To avoid unnecessary update reports, the bag of records with the same I values for both F_1 and F_2 need to be processed together. Let us suppose that BAG_1 contains the records from F_1 with a identification value of I_i. Let BAG_2 contain the records from F_2 with a identification value of I_i. To illustrate, let us suppose we are given the following two bags containing the records with a identification value of I_1.

$$\begin{aligned} BAG_1 &: \{\langle I_1, B_1\rangle, \langle I_1, B_1\rangle\} \\ BAG_2 &: \{\langle I_1, B_2\rangle, \langle I_1, B_1\rangle, \langle I_1, B_3\rangle\} \end{aligned}$$

After the unchanged records (*i.e.*, $\langle I_1, B_1\rangle$) are identified, they are removed from BAG_1 and BAG_2 to produce the bags BAG'_1 and BAG'_2.

$$\begin{aligned} BAG'_1 &: \{\langle I_1, B_1\rangle\} \\ BAG'_2 &: \{\langle I_1, B_2\rangle, \langle I_1, B_3\rangle\} \end{aligned}$$

At this point, there is no record in BAG'_1 that is also in BAG'_2, and vice versa. If there are the same number of records in BAG'_2 as in BAG'_1, we assume that the records in BAG'_1 were all updated since this generates the minimum number of change reports. If there are more records in BAG'_2 (as in the case above), some records must have also been inserted. Otherwise, there are more records in BAG'_1, and some records must have been deleted.

To produce the appropriate update reports, we first match the records in BAG'_1 and BAG'_2 and (arbitrarily) designate which records have been updated as opposed to inserted or deleted. In the example, the record $\langle I_1, B_1\rangle$ in BAG'_1 can be matched with either $\langle I_1, B_2\rangle$ or $\langle I_1, B_3\rangle$. That is, either $\langle I_1, B_1\rangle$ was updated to $\langle I_1, B_2\rangle$ or $\langle I_1, B_3\rangle$. Let us suppose $\langle I_1, B_1\rangle$ is matched with the latter record and the update report $\langle \mathit{Update}, I_1, B_1, B_3\rangle$ is produced. Any remaining "unmatched" records like $\langle I_1, B_2\rangle$ in BAG'_2 are reported as inserts. Unmatched records in BAG'_1 are reported as deletes.

The matching algorithm *Match* is shown in Figure 2.3. The matching algorithm is actually quite general since the input bags BAG_1 and BAG_2 do not need to have records with the same identification values. Lines 1–2 remove the records in BAG_1 and BAG_2 that have not changed using the $\dot{-}$ operator. Lines 4–6 match records in BAG'_1 and BAG'_2 based on the identification attribute and produce the appropriate update report. A delete is reported in Line 8 if there are no records in BAG'_2 with the same identification value as the current record R_1 from BAG'_1. Any remaining records in BAG'_2 are reported as inserts in Line 10.

The matching algorithm is general and can be used to compute the bag differential of two snapshots. However, it is too inefficient for use on large snapshots. Even when an in-memory index on the identification attribute is created, in the worst case, the complexity of the matching algorithm is $O(n_1 \cdot n_2)$, where n_1 is the number of records in BAG_1, and n_2 is the number of records in BAG_2. Thus, the matching algorithm should only be used when the two input bags are small. The strategy then is to use sort merge outerjoin or partitioned hash outerjoin to ensure that the inputs to the matching algorithm are relatively small. For instance, the partitioned hash outerjoin can first partition the snapshots into buckets, and then use the matching algorithm to process each pair of buckets. This way, the inputs to the matching algorithm are not that large.

Algorithm 2.2.2 *Match*
Input BAG_1, BAG_2
Output F_{OUT} (bag differential of BAG_1 and BAG_2
Method
1. $BAG'_1 \leftarrow BAG_1 \dot{-} BAG_2$
2. $BAG'_2 \leftarrow BAG_2 \dot{-} BAG_1$
3. **for** each record R_1 in BAG'_1
4.   **if** there is a record R_2 in BAG'_2 where $R_2.I = R_1.I$ **then**
5. $F_{OUT} \leftarrow Output(\langle Update, R_1.I, R_1.B, R_2.B \rangle)$
6.   Remove R_2 from BAG'_2
7.   **else**
8. $F_{OUT} \leftarrow Output(\langle Delete, R_1.I, R_1.B \rangle)$
9. **for** each record R_2 remaining in BAG'_2
10. $F_{OUT} \leftarrow Output(\langle Insert, R_2.I, R_2.B \rangle)$

◇

Figure 2.3: Matching Algorithm

The *SM* outerjoin (Figure 2.2) can be modified easily to use the matching algorithm to perform bag differentials. The only portion that needs to be changed is when the records being read have the same identification value. In this case, the modified sort merge outerjoin (Figure 2.4) reads the next records in the two snapshots with the same identification value to produce the two bags BAG_1 and BAG_2 that are input to the matching algorithm (Lines

13–14). The matching algorithm is then used to process BAG_1 and BAG_2 and produce the appropriate reports. As long as the identification attribute(s) is selected carefully so that the number of records in BAG_1 and BAG_2 with the same identification value is small, both BAG_1 and BAG_2 should fit in memory. Assuming BAG_1 and BAG_2 do fit in memory, the number of IOs incurred by the sort merge outerjoin is still $|F_1| + 4 \cdot |F_2|$ IOs.

Algorithm 2.2.3 *SM-Bag*
Input $F_{1\ sorted}$, F_2
Output F_{out} (the bag differential), $F_{2\ sorted}$
Method
1. Lines 1–10 in Figure 2.2
11. **else if** $(r_1.I = r_2.I)$ **then**
12.   **if** $(r_1.B \neq r_2.B)$ **then**
13. $BAG_1 \leftarrow r_1$ plus all the records following r_1 with same identification value
14. $BAG_2 \leftarrow r_2$ plus all the records following r_2 with same identification value
15.   Match(BAG_1, BAG_2)
16. $r_1 \leftarrow$ read the next record from $F_{1\ sorted}$
17. $r_2 \leftarrow$ read the next record from $F_{2\ runs}$; $F_{2\ sorted} \leftarrow Output(\langle r_2.I,\ r_2.B\rangle)$

$\diamond$

Figure 2.4: Sort Merge Outerjoin as a Bag Differential Algorithm

The partitioned hash outerjoin algorithm can be modified easily to compute bag differentials given the matching algorithm (Figure 2.3). Recall that the first phase of the algorithm partitions both snapshots into buckets by computing a hash function on the identification attribute. In the second phase, each pair of corresponding buckets (denoted B_{F_1} and B_{F_2}) is processed using the matching algorithm. (B_{F_1} is the BAG_1 input, B_{F_2} is the BAG_2 input.) Although the records in B_{F_1} and B_{F_2} do not have the same identification values, the matching algorithm *Match* in Figure 2.3 is general enough to handle the situation. Assuming the two buckets fit in memory, the IO cost of the modified partitioned hash outerjoin algorithm is still $|F_1| + 3 \cdot |F_2|$ IOs. The processing cost can be reduced by creating in-memory indices on the identification attribute.

2.2.3 Compression Techniques

Our compression algorithms reduce the sizes of records and the required IO. Compression can be performed in varying degrees. For instance, compression may be performed on the records of a file by compressing the whole record (possibly excluding the key field) into n bits. A block or a group of blocks can also be compressed into n bits. There are also numerous ways to perform compression such as computing the check sum of the data, hashing the data to obtain an integer or simply omitting fields in a record that are not important in the comparison process. Compression can also be *lossy* or *lossless*. In the latter case, the compression function guarantees that two different uncompressed values are mapped into different compressed values. Lossy compression functions do not have this guarantee but have the potential of achieving higher compression factors. Henceforth, we assume that we are using a lossy compression function. We ignore the details of the compression function and simply refer to it as $Compress(x)$.

There are a number of benefits from processing compressed data. First of all, the compressed intermediate files, such as the buckets for the partitioned hash outerjoin, are smaller. Thus, there will be fewer IO when reading the intermediate files. Moreover, the compressed file may be small enough to fit in memory. Even if the compressed file does not fit entirely in memory, some of the join algorithms may still benefit. For example, the compressed file may result in buckets that fit in memory which improves the matching phase of the partitioned hash outerjoin algorithm.

Compression is not without its disadvantages. As mentioned earlier, a lossy compression function may map two different records into the same compressed value. Thus, the snapshot differential algorithm is probabilistic and may not be able to detect all the changes to a snapshot. We now show that the algorithm may fail to detect changes with a probability of 2^{-n}, where n is the number of bits for the compressed value. Assume that we are compressing an object (which may be the B field, or the entire record, or an entire block, etc.) of b bits ($b > n$). There are then 2^b possible values for this object. Since there are only 2^n values that the compressed object can attain, there are $2^b/2^n$ original values mapped to each compressed value. Thus for each given original value, the probability that another value maps to the same compressed value is $((2^b/2^n) - 1)/2^b$, which is approximately 2^{-n} for large values of b. For sufficiently large values of n, this probability can be made very small. The expression 2^{-n}, henceforth denoted as E, gives the probability that a *single* comparison is erroneous. For example, if the B field of the record $\langle K, B\rangle$ is compressed into a 32-bit integer, the probability that a single comparison (of two B fields) is erroneous is 2^{-32} or approximately $2.3 \cdot 10^{-10}$. However, as we compare more records, the likelihood that a change is missed increases. To put this probability of error into perspective, let

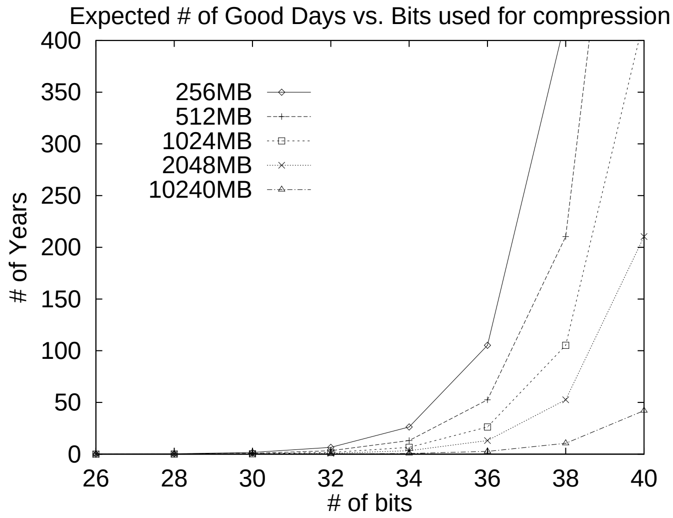

Figure 2.5: $N_{good\ days}$ for Different File Sizes

us suppose we perform a differential on two 256 MB snapshots daily. We now proceed to compute how many days we expect to pass before a record change is missed. We first compute the probability (denoted as p_{day}) that there is no error in comparing two given snapshots (that is, there is no error in one day). Let us suppose that the record size is 150 bytes which means that there are approximately 1,789,570 records for each file.

$$p_{day} = (1 - E)^{records(F)} = (1 - 2.3 \cdot 10^{-10})^{1,789,570} = 0.99979169 \tag{2.1}$$

Using this probability, we can compute the expected number of days (denoted as $N_{good\ days}$) before an error occurs.

$$N_{good\ days} = (1 - p_{day}) \cdot \sum_{1 \le i} i \cdot p_{day}^{i-1} = \frac{1}{1 - p_{day}} = 2,430\ days \tag{2.2}$$

The expected number of days comes out to be 2,430 days, or more than 6.7 years! We believe that for some types of warehousing applications, such as data mining, this error rate will be acceptable.

It is evident from the equations above that as the number of records increases, the expected number of days before an error occurs goes down. This is shown more clearly in Figure 2.5. The graph shows that a 10 GB file will encounter more errors than a 256 MB file. However, as the number of bits used for compressing the B field is increased, the the expected number of years before an error occurs can be made comfortably large even for large files.

For the algorithms we will present here, we consider two ways of compressing the records. For both compression formats, we do not compress the key, and we denote the compressed B field as b. The first format is simply compress a record $\langle K, B\rangle$ into $\langle K, b\rangle$. (For bag differentials, a record $\langle I, B\rangle$ is compressed into $\langle I, b\rangle$.) For the second form, the only difference is that a pointer is appended forming the record $\langle K, b, p\rangle$. (For bag differentials, a record $\langle I, B\rangle$ is compressed into $\langle I, b, p\rangle$.) The pointer p points to the corresponding disk resident uncompressed record. The use of the pointer will be explained when we describe the algorithms. We use u to represent the ratio of the size of the original record to that of the compressed record (including the key and pointer, if any). So, if an uncompressed file is size $|F|$, the compressed size will be $|F|/u$ blocks long.

Algorithm 2.2.4 *SMC1*
Input $f_{1\ sorted}$, F_2
Output F_{out} (the set differential), $f_{2sorted}$
Method
1. $F_{2\ runs} \leftarrow SortIntoRuns(F_2)$
2. $r_1 \leftarrow$ read the next record from $f_{1\ sorted}$ (other r_1 reads later on are also from $f_{1\ sorted}$)
3. $r_2 \leftarrow$ read the next record from $F_{2\ runs}$;
   $f_{2\ sorted} \leftarrow Output(\langle r_2.K,\ Compress(r_2.B)\rangle)$
4.-6. Lines 4–6 of Figure 2.2
7. $r_2 \leftarrow$ read the next record from $F_{2\ sorted}$;
   $f_{2\ sorted} \leftarrow Output(\langle r_2.K,\ Compress(r_2.B)\rangle)$
8.-11. Lines 8–11 of Figure 2.2
12. **if** $(r_1.b \neq Compress(r_2.B))$ **then**
13.-14. Lines 8–11 of Figure 2.2
15. $r_2 \leftarrow$ read the next record from $F_{2\ sorted}$;
   $f_{2\ sorted} \leftarrow Output(\langle r_2.K,\ Compress(r_2.B)\rangle)$

$\diamond$

Figure 2.6: Sort Merge Outerjoin Enhanced with the $\langle K,\ b\rangle$ Compression Format

2.2.4 Set Differentials and Compression

We now augment the sort merge outerjoin with compression. We assume that the compressed sorted F_1 file was produced in the previous differential (denoted as $f_{1\ sorted}$, with

Algorithm 2.2.5 *SMC2*
Input $f_{1\ sorted}$, F_2
Output F_{out} (the set differential), $f_{2\ sorted}$
Method

1. $f_{2\ runs} \leftarrow SortIntoRuns \circ Compress(F_2)$
2. $r_1 \leftarrow$ read the next record from $f_{1\ sorted}$
3. $r_2 \leftarrow$ read the next record from $f_{2\ runs}$; $f_{2\ sorted} \leftarrow Output(\langle r_2.K,\ r_2.b,\ r_2.p\rangle)$
4. **while** $((r_1 \neq NULL) \wedge (r_2 \neq NULL))$
5.   **if** $((r_1 = NULL) \vee (r_1.K > r_2.K))$ **then**

5a. $r_{full} \leftarrow$ read record in F_2 with address $r_2.p$

6. $F_{out} \leftarrow Output(\langle Insert,\ r_2.K,\ r_{full}.B\rangle)$
7. $r_2 \leftarrow$ read the next record from $f_{2\ runs}$; $f_{2\ sorted} \leftarrow Output(\langle r_2.K,\ r_2.b,\ r_2.p\rangle)$
8.   **else if** $((r_2 = NULL) \vee (r_1.K < r_2.K)$ **then**
9. $F_{out} \leftarrow Output(\langle Delete,\ r_1.K\rangle)$
10. $r_1 \leftarrow$ read the next record from $f_{1\ sorted}$
11.   **else if** $(r_1.K = r_2.K)$ **then**
12.   **if** $(r_1.b \neq r_2.b)$ **then**

12a. $r_{full} \leftarrow$ read record in F_2 with address $r_2.p$

13. $F_{out} \leftarrow Output(\langle Update,\ r_2.K,\ r_{full}.B\rangle)$
14. $r_1 \leftarrow$ read the next record from $f_{1\ sorted}$
15. $r_2 \leftarrow$ read the next record from $f_{2\ runs}$; $f_{2\ sorted} \leftarrow Output(\langle r_2.K,\ r_2.b,\ r_2.p\rangle)$

$\diamond$

Figure 2.7: Sort Merge Outerjoin Enhanced with the $\langle K,\ b,\ p\rangle$ Compression Format

a size of $|F_1|/u$). For this algorithm, we use the $\langle K,\ b\rangle$ compression format. The modified sort merge algorithm called *SMC1* is shown in Figure 2.6. Note that only the steps that differ from the *SM* outerjoin algorithm (Figure 2.2) are shown explicitly. Lines 3, 7 and 15 now first compress the B field before producing an output into $f_{2\ sorted}$ (which is needed in the next differential). Also, when detecting the updates in Line 12, the compressed versions of the B field are compared.

The sorting phase of the algorithm incurs $2 \cdot |F_2|$ IOs (since it generates only the sorted runs as in Algorithm 2.2.1). The matching phase (Line 4 onwards) incurs $|F_2| + |f_1|$ IOs since the two files are scanned once. Lastly, the sorted $f_{2\ sorted}$ must be produced for the next differential, which costs $|f_2|$ IOs. The total cost is then $|f_1| + 3 \cdot |F_2| + |f_2|$ IOs.

Greater improvements may be achieved by compressing not only the first snapshot but also the second snapshot before the files are matched. When the second snapshot arrives, it is read into memory and *compressed* sorted runs are written out. In essence, the uncompressed F_2 file is read only once. The problem introduced by compressing the second snapshot is that when insertions and updates are detected, the original uncompressed record must be obtained from F_2. In order to find the original (uncompressed) record, a pointer to the record must be saved in the compressed record. Thus, for this algorithm, the $\langle K, b, p\rangle$ compression format must be used. The full algorithm called *SMC2* is shown in Figure 2.7. Line 5a (Line 12a) shows that when an insertion or update is detected, the pointer p of the current record is used to obtain the original record in order to produce the correct output.

Line 1 of Algorithm 2.2.5 only incurs $|F_2|+|f_2|$ IOs instead of $2\cdot|F_2|$ IOs. Lines 4 through 15 incur $|f_1| + |f_2| + U + I$ IOs, where U and I are the number of updates and insertions found. An additional $|f_2|$ IOs are needed to write out the sorted f_2 file. As a result, the overall cost is $|f_1| + |F_2| + 3 \cdot |f_2| + U + I$. The savings in IO cost is significant especially if there are few updates and inserts. Moreover, we are also assuming that each access using the pointer p requires a random IO. The disk access can be optimized by recording all the pointers that need to be accessed. After the differential is performed, these recorded pointers are used to produce the inserts and the updates. By sorting the pointers, the cost of probing the original snapshot is lessened since the IO operations are no longer random.

The partitioned hash outerjoin is augmented with compression in a very similar manner to the sort merge outerjoin. We assume that the compressed bucket files for the first snapshot (denoted collectively as f_1) was produced in the previous snapshot differential.

When the second snapshot arrives, the buckets are created as explained in Section 2.2.1, incurring $2 \cdot |F_2|$ IOs. The corresponding buckets are matched by reading the smaller bucket (which is most likely a bucket in f_1) into main memory. An in-memory hash table is constructed and the algorithm proceeds in a similar fashion to the partitioned hash outerjoin. The only difference is that the compressed B fields are compared when searching for an update. In addition, the records in the buckets of F_2 are compressed and written into a bucket file f_2. After processing all of the F_2 buckets, the set of compressed buckets that comprise f_2 is also complete and ready for the next snapshot differential. The matching phase incurs $|f_1| + |F_2|$ IOs to read in the buckets and $|f_2|$ to write out the buckets for the next snapshot differential. Therefore, the overall cost is $|f_1| + 3 \cdot |F_2| + |f_2|$ IOs.

Like the sort merge outerjoin, greater performance gains can be made by compressing the buckets of F_2 before the matching phase. Similarly, the $\langle K, b, p\rangle$ compression format is used. In this case, only $|F_2|+|f_2|$ IO operations are needed to bucketize F_2 into a set of compressed buckets denoted as f_2. The matching phase is similar except that pointers must be followed to report inserts and updates. As a result, the overall IO cost is $|f_1| + |F_2| + 2 \cdot |f_2| + I + U$. As in the sort merge outerjoin, we can also argue that the probes on F_2 through p can be recorded and can be done more efficiently after processing f_2.

The performance gains can even be greater if the compression factor u is high enough such that all of the buckets of F_1 fit in memory. In this case, all the buckets for F_1 are simply read into memory ($|f_1|$ IOs). The file F_2 is then scanned, and for each record in F_2 read, the in-memory buckets are probed. The compressed buckets for F_2 can also be constructed for the next differential during this probe. The overall cost of this algorithm is only $|f_1| + |F_2| + |f_2|$ IOs. Note that the cost is independent of the number of updates and inserts unlike the algorithm discussed previously. Unfortunately, this optimization cannot be used for the sort merge outerjoin because constructing the compressed sorted file for F_2 cannot be done by just scanning through F_2 once.

2.2.5 Bag Differentials and Compression

The bag differential algorithms developed in Section 2.2.2 can also be augmented with compression. However, the compression format $\langle I, b\rangle$ cannot be used. Intuitively, this is because for bag differentials, the old values for the B attribute are required in reporting deletes and updates. That is, to unambiguously report that a record $\langle I_i, B_i\rangle$ was deleted, it must be reported as $\langle Delete, I_i, B_i\rangle$. Similarly, to unambiguously report that a record $\langle I_i, B_i\rangle$ was updated to $\langle I_i, B_j\rangle$, it must be reported as $\langle I_i, B_i, B_j\rangle$. In both cases, the old value of the B attribute (*i.e.*, B_1) is required. Unfortunately, assuming a lossy compression function, the B attribute value cannot be recovered from the compression format $\langle I, b\rangle$.

On the other hand, the uncompressed B attribute value can be obtained from the compression format $\langle I, b, p\rangle$ by following the pointer p that points to the uncompressed record on disk. Thus, in this section, we augment the sort merge outerjoin as well as the partitioned hash outerjoin with the compression format $\langle I, b, p\rangle$.

The sort merge outerjoin *SMC2* that uses the compression format $\langle I, b, p\rangle$ for set differential is shown in Figure 2.7. The algorithm requires only slight modifications for it to apply to bag differentials as shown in Figure 2.8. First, when a deletion is detected (Line 8a), the uncompressed record must be read to obtain the old value of the B attribute. Second, when the identification values of the two records match, the algorithm is modified to read the next records in the two snapshots with the same identification value. Again, these records must be processed together to avoid reporting unnecessary updates. The bag of records from F_1 denoted BAG_1, and the bag of records from F_2 denoted BAG_2, are input to the matching algorithm discussed previously (see Figure 2.3). The matching algorithm also needs to be modified slightly since it is taking as input compressed records as opposed to uncompressed ones. Hence, whenever an update, or a delete or an insert is detected, the matching algorithm must follow the appropriate pointers to obtain the uncompressed B values.

Recall that the IO cost of the sort merge outerjoin using the compression format $\langle I, b, p\rangle$ to perform set differentials is $|f_1| + |F_2| + 3 \cdot |f_2| + I + U$. This equation assumes that I insertions and U updates are reported. IO operations are incurred whenever insertions and updates are reported because an uncompressed record must be read from the disk to create the report. In the case of bag differentials, the uncompressed record must also be located to report a deletion. Hence, the IO cost of the sort merge outerjoin using the compression format $\langle I, b, p\rangle$ is $|f_1| + |F_2| + 3 \cdot |f_2| + I + U + D$, where D is the number of deletions reported. The IO cost of the sort merge outerjoin can be high if there are a lot of changes detected. If there are only few changes detected, the benefit of compressing the snapshots can be substantial as we will show in Section 2.4.

Algorithm 2.2.6 *SMC2-Bag*
Input $f_{1\ sorted}$, F_2
Output F_{out} (the bag differential), $f_{2sorted}$
Method
1.-8. Lines 1–8 of Figure 2.7
8a. $r_{full} \leftarrow$ read record in F_1 with address $r_1.p$
9.-12. Lines 9–12 of Figure 2.7
13. $BAG_1 \leftarrow r_1$ plus all the records following r_1 with same identification value
14. $BAG_2 \leftarrow r_2$ plus all the records following r_2 with same identification value
15. Match(BAG_1, BAG_2) // Algorithm 2.2.2
16. $r_1 \leftarrow$ read the next record from $f_{1\ sorted}$
17. $r_2 \leftarrow$ read the next record from $f_{2\ runs}$; $f_{2\ sorted} \leftarrow Output(\langle r_2.I,\ r_2.b,\ r_2.p\rangle)$
$\diamond$

Figure 2.8: Sort Merge Outerjoin Enhanced with the $\langle I,\ b,\ p\rangle$ Compression Format for Bag Differential

In Section 2.2.4, we also modified the partitioned hash outerjoin algorithm to use the $\langle I, b, p\rangle$ compression format to perform set differentials. For the algorithm to perform bag differentials, it must use the matching algorithm *Match* (Figure 2.3) to process each pair of buckets as explained in Section 2.2.2. The matching algorithm follows the appropriate pointers whenever an update, or a delete or an insert is detected. Because the original B attribute value is required in producing a deletion report, the IO cost of the algorithm increases by D where D is the number of deletions detected. (The IO cost increases from $(|f_1| + |F_2| + 2 \cdot |f_2| + I + U)$ to $(|f_1| + |F_2| + 2 \cdot |f_2| + I + U + D)$.)

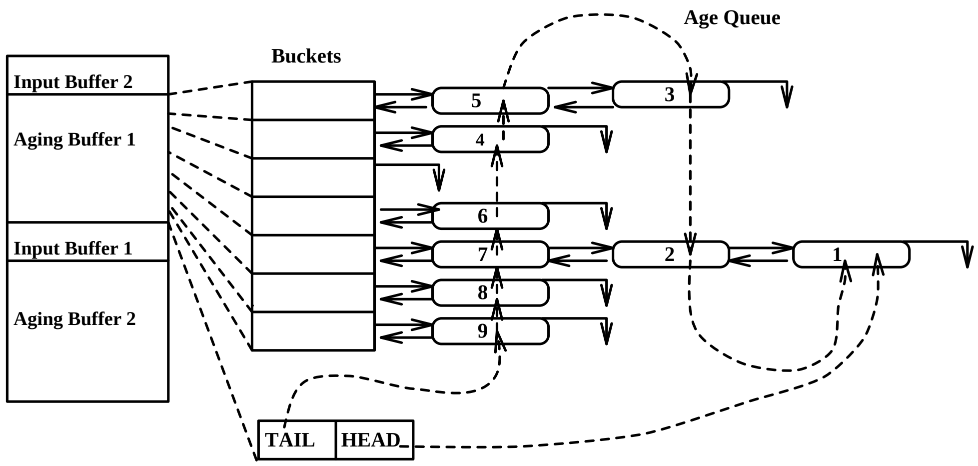

Figure 2.9: The *window* Algorithm Data Structures

2.3 The Window Algorithm

2.3.1 Window for Set Differentials

In the previous section, we described algorithms that compute the differential of two snapshots based on outerjoin algorithms. We saw that the snapshots are read multiple times. Since the files are large, reading the snapshots multiple times can be costly. We now present an algorithm that reads the snapshots exactly *once*. This new algorithm works well when matching records are physically "nearby" in the files. As mentioned in Section 2.1.1, matching records cannot be expected to be in the same position in the two snapshots, due to possible reorganizations at the source. However, we may still expect a record to remain

Algorithm 2.3.1 *Window*
Input F_1, F_2, n (number of blocks in the input buffer)
Output F_{out} (the set differential)
Method
1. $input\ buffer\ 1 \leftarrow$ Read n blocks from F_1
2. $input\ buffer\ 2 \leftarrow$ Read n blocks from F_2
3. **while** $((input\ buffer\ 1 \neq EMPTY) \wedge (input\ buffer\ 2 \neq EMPTY))$
4. Match $input\ buffer\ 1$ against $input\ buffer\ 2$
5. Match $input\ buffer\ 1$ against $aging\ buffer\ 2$
6. Match $input\ buffer\ 2$ against $aging\ buffer\ 1$
7. Insert contents of $input\ buffer\ 1$ into $aging\ buffer\ 1$
8. Insert contents of $input\ buffer\ 2$ into $aging\ buffer\ 2$
9. $input\ buffer\ 1 \leftarrow$ Read n blocks from F_1
10. $input\ buffer\ 2 \leftarrow$ Read n blocks from F_2
11. Report records in $input\ buffer\ 1$ as deletes
12. Report records in $input\ buffer\ 2$ as inserts

$\diamond$

Figure 2.10: Window Algorithm

in a relatively small area, such as a block, cylinder, or track. This is because file reorganization algorithms typically rearrange records within a physical sub-unit. The *window* algorithm takes advantage of this, and of ever increasing main memory capacity, by maintaining a moving window of records in memory for each snapshot. Only the records within the window are compared in the hope that the matching records occur within the window. Unmatched records are reported as either an insert or a delete, which can lead to useless delete-insert pairs. As discussed in Section 2.1.1, a small number of these may be tolerable.

For the window algorithm, we divide available memory into four distinct parts as shown in Figure 2.9. Each snapshot has its own *input buffer* (*input buffer 1* is for F_1) and *aging buffer*. The input buffer is simply the buffer used in transferring blocks from disk. The aging buffer is essentially the moving window mentioned above.

The algorithm is shown in Figure 2.10 and we now proceed to explain each step. Lines 1 and 2 simply read a constant number of input block of records from file F_1 and file F_2 to fill *input buffer 1* and *input buffer 2*, respectively. This process will be done repeatedly by Lines 9 and 10. Before the input buffers are refilled, the algorithm guarantees that they are empty. Lines 4 through 6 are concerned with matching the records of the two snapshots.

In Line 4, the matching is performed in a nested loop fashion. This is not expensive since the input buffers are relatively small. The matched records can produce updates if the B fields differ. The slots that these matching records occupy in the buffer are also marked as free. In Line 5, the remaining records in *input buffer 1* are matched against *aging buffer 2*. Since the aging buffers are much larger, the aging buffers are actually hash tables to make the matching more efficient (Figure 2.9). For each remaining record in *input buffer 1*, the hash table that is *aging buffer 2* is probed for a match. As in Line 4, an update may be produced by this matching. The slots of the matching records are also marked as free. Line 6 is analogous to Line 5 but this time matching *input buffer 2* and *aging buffer 1*. Lines 7 and 8 clear both input buffers by forcing the unmatched records in the input buffers into their respective aging buffers. The same hash function used in Lines 4 and 5 is used to determine which bucket the record is placed into. Since new records are forced into the aging buffer, some of the old records in the aging buffer may be displaced. These displaced records constitute the deletes (inserts) if the records are displaced from *input buffer 1* (*input buffer 2*). The displacement of old records is explained further below. The steps are then repeated until both snapshots are processed. At that point, any remaining records in the aging buffers are output as inserts or deletes.

In the hash table that constitutes the aging buffer there is an embedded "aging" queue, with the head of the queue being the oldest record in the buffer, and the tail being the youngest. Figure 2.9 illustrates the aging buffer. Each entry in the hash table has a timestamp associated with it for illustration purposes only. The figure shows that the oldest record (with the smallest timestamp) is at the head of the queue. Whenever new records are forced into the aging buffer, the new records are placed at the tail of the queue. If the aging buffer is full, the record at the head of the queue is displaced as a new record is enqueued at the tail. This action produces a delete (insert) if the buffer in question is *aging buffer 1* (*aging buffer 2*).

Since files are read once, the IO cost for the *window* algorithm is only $|F_1| + |F_2|$ *regardless* of memory size, snapshot size and number of updates and inserts. Thus the window algorithm achieves the *optimal IO* performance if compression is not considered. However, the *window* algorithm can produce useless delete-insert pairs in Steps 6 and 7 of the algorithm. Intuitively, the number of useless delete-insert pairs produced depends on how physically different the two snapshots are.

To quantify this difference, we define the concept of the *distance* of two snapshots. We want the distance measure to be symmetric and independent of the size of the file. While the reason for the first property is obvious, the reason for the second is more subtle. If the measure is not independent of the size of the file, we may end up with a measure that is unbounded. For instance, if the distance of two snapshots is defined to be the sum of the absolute value of the differences in positions of matching records, this sum may become arbitrarily large for large snapshots. Moreover, such a measure can be misleading since two small snapshots that are in opposite order will have a small distance measure when intuitively they should have a large distance.

The equation below exhibits the two desired properties.

$$distance = \frac{\sum_{R_1 \epsilon F_1, R_2 \epsilon F_2, match(R_1, R_2)} |pos(R_1) - pos(R_2)|}{max(records(F_1), records(F_2))^2/2} \tag{2.3}$$

The function *pos* returns the physical position of a record in a snapshot. The boolean function *match* is true when records R_1 and R_2 have matching keys. The function *records* returns the number of records of a snapshot file. F represents the larger of the two files. Thus, this equation sums up the absolute value of the difference in position of the matching records and normalizes it by the maximum distance for the given snapshot file sizes. The maximum distance between two snapshots is attained when the records in the second snapshot are in the opposite order (the first record is exchanged with the last record, the second record with the second to the last, and so on) relative to the first snapshot. If $records(F_1) = records(F_2)$, it is easy to see that in the worst case the average displacement of each record is $records(F)/2$, and hence the maximum distance is $records(F) \cdot records(F)/2$. If the files are of different sizes, using the larger of the two files gives an upper bound on the maximum distance. Our *distance* metric will be used in the following section to evaluate the window algorithm.

2.3.2 Window for Bag Differentials

The window algorithm can be modified easily to perform bag differentials. Recall that for set differentials, the window algorithm first fills the two input buffers. Given a record $\langle I_1, B_1 \rangle$ in *input buffer 1* (or *aging buffer 1*), the window algorithm looks for the record in *input buffer 2* or *aging buffer 2* that has a identification value of I_1. If found, the window algorithm checks if the B attributes of the two records are the same. If the B attributes do not match, the appropriate update report is produced. However, in computing bag differentials, there may be more than one record in *input buffer 2* and *aging buffer 2* with

Algorithm 2.3.2 *Window-Bag*
Input F_1, F_2, n (number of blocks in the input buffer)
Output F_{out} (the bag differential)
Method
1. $input\ buffer\ 1 \leftarrow$ Read n blocks from F_1
2. $input\ buffer\ 2 \leftarrow$ Read n blocks from F_2
3. **while** $((input\ buffer\ 1 \neq EMPTY) \wedge (input\ buffer\ 2 \neq EMPTY))$
4. Match $input\ buffer$ 1 against $input\ buffer$ 2 and $aging\ buffer$ 2 looking for unchanged records
5. Match $input\ buffer$ 2 against $input\ buffer$ 1 and $aging\ buffer$ 1 looking for unchanged records
6. Match $input\ buffer$ 1 against $input\ buffer$ 2 and $aging\ buffer$ 2 looking for updated records
7. Match $input\ buffer$ 2 against $input\ buffer$ 1 and $aging\ buffer$ 1 looking for updated records
8. Insert contents of $input\ buffer$ 1 into $aging\ buffer$ 1
9. Insert contents of $input\ buffer$ 2 into $aging\ buffer$ 2
10. $input\ buffer\ 1 \leftarrow$ Read n blocks from F_1
11. $input\ buffer\ 2 \leftarrow$ Read n blocks from F_2
12. Report records in $input\ buffer$ 1 as deletes
13. Report records in $input\ buffer$ 2 as inserts

$\diamond$

Figure 2.11: Window Algorithm for Bag Differentials

a identification value of I_1. If the window algorithm is not modified and processes the first record it finds in *input buffer 2* or *aging buffer 2* with a identification of I_1, it will report an update when the B attributes do not match. However, reporting this update is unnecessary if the record $\langle I_1, B_1 \rangle$ is also in *input buffer 2* or *aging buffer 2*. Hence, the window algorithm must be modified to find the record (if any) with a B attribute of B_1 as well. This way, the algorithm detects unchanged records first and avoids reporting unnecessary updates. The modified algorithm is shown in Figure 2.11. It is clear that the modified window algorithm still has an IO cost of $|F_1| + |F_2|$.

	Variable Description	Default Values
M	Memory Size	32 MB
B	Block Size	16K
F	File Size	256 MB or 1024 MB
R	Record Size	150 bytes
$records(F)$	Number of Rows	1,789,569 or 7,158,279
r	Compressed Record Size	10 or 14 bytes
u	Compression Factor	15 or 10
$U+I$	Number of Inserts and Updates	1% of $records(F)$
IO	Number of IOs	N/A
X	Intermediate File Size	N/A
E	Probability of Error	N/A

Table 2.1: List of Variables

2.4 Performance Evaluation

2.4.1 Analytical IO Comparison

We have outlined in the previous sections algorithms for computing set differentials: performing sort merge outerjoin (SM), performing a partitioned hash outerjoin (PH), performing a sort merge outerjoin with two kinds of record compression ($SMC1$, $SMC2$), performing partitioned hash outerjoin with two kinds of record compression ($PHC1$, $PHC2$) and using the *window* algorithm (W). $SMC1$ denotes sort merge outerjoin with a record compression format of $\langle K, b\rangle$ (similarly for $PHC1$); $SMC2$ uses the record compression format $\langle K, b, p\rangle$ (similarly for $PHC2$).

Recall that we also modified the above algorithms to compute bag differentials. Furthermore, we showed that the IO cost of the bag differential algorithms are almost the same as those of the set differential algorithms. Thus, we focus only one the set differential algorithms in this section (including Section 2.4.2).

We compare the set differential algorithms in terms of IO cost, size of intermediate files, and the probability of error. This study serves as an illustration of potential differences between the algorithms in a few realistic scenarios.

Table 2.1 shows the variables that will be used in comparing the algorithms. We assume that the snapshots have the same number of records. The number of records ($records(F)$)

Algorithm	IO_{256} (%savings)	IO_{1024} (%savings)	X_{256} (MB)	X_{1024} (MB)	Probability of Error (E)
SM	81,920	327,680	16384	65,536	0
$SMC1$	51,336 (37%)	205,346 (37%)	16,384	65,536	$2.3 \cdot 10^{-10}$
$SMC2$	40,833 (50%)	163,333 (50%)	1,639	6,554	$2.3 \cdot 10^{-10}$
PH	65,536 (20%)	262,144 (20%)	16,384	65,536	0
$PHC1$	18,568 (77%)	205,346 (37%)	16,384	65,536	$2.3 \cdot 10^{-10}$
$PHC2$	19,660 (76%)	156,779 (52%)	1,639	6,554	$2.3 \cdot 10^{-10}$
W	32,768 (60%)	131,072 (60%)	0	0	0

Table 2.2: Comparison of Algorithms

are calculated using F/R, where R is the record size (150 bytes). The compressed record size is 10 bytes for the $\langle K, b\rangle$ format and 14 bytes for the $\langle K, b, p\rangle$ format. This leads to compression factors of 15 and 10 respectively.

Table 2.2 shows a summary of the results computed for the various algorithms. The two columns labeled IO_{256} and IO_{1024} show the IO cost incurred in processing 256 MB and 1024 MB snapshots for the different algorithms. Using the sort merge outerjoin as a baseline, we can see that the partitioned hash outerjoin (PH) reduces the IO cost by 20%. Compression using the $\langle K, b\rangle$ record format achieves a 37% reduction in IO cost over sort merge using $SMC1$, and a 50% reduction using $SMC2$. For the 256 MB file, the compressed file fits in memory which enables the $PHC1$ and $PHC2$ algorithms to build a complete in-memory hash table, as explained in Section 2.2.4. The reduction in IO cost for these two algorithms, in this case, surpasses even that of the window algorithm.

However, when the larger file is considered, the compressed file no longer fits in the 32 MB memory. Thus the $PHC1$ and $PHC2$ algorithms achieve more modest reductions in this case (37% and 52% respectively). Other than these two algorithms, the reductions achieved by the other algorithms are unchanged even with the larger file.

Figure 2.12 shows how the algorithms compare when the size of the snapshots is varied over a range. The values of other parameters are unchanged. Note that we have not plotted $SMC1$ and $SMC2$ since their plots are almost indistinguishable from $PHC1$ and $PHC2$ respectively beyond a file size of 500 MB. Also note the discontinuity in the graph for $PHC1$ and $PHC2$. $PHC1$ is able to build an in-memory hash table if the file is smaller than 500 MB (and files smaller than 320 MB for $PHC2$). If the partitioned hash outerjoin algorithms

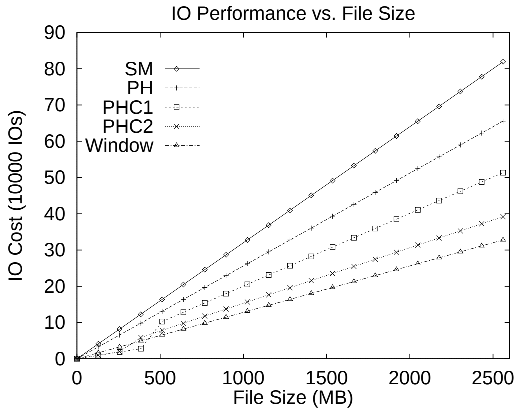

Figure 2.12: IO Cost Comparison of Algorithms

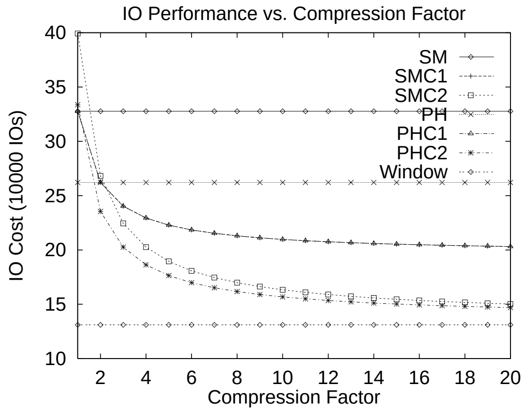

Figure 2.13: IO Cost and Compression Factor

are able to build an in-memory hash table, they can even outperform the window algorithm.

Clearly, the IO savings for compression algorithms depend on the compression factor. Figure 2.13 illustrates that when the compression factor is low, the algorithms with compression perform worse than PH (even worse than SM in case of $SMC1$ and $SMC2$). The other point that this graph illustrates is that the benefits of compression are bounded (which is to be expected from the IO cost equations). Thus, going beyond a factor of 10 in this case does not buy us much.

The performance of the compression algorithms that use the pointer format (algorithms $PHC2$ and $SMC2$) depend on the number of updates and inserts. If $U + I$ is higher than what we have assumed, $PHC1$ and $SMC1$ outperform $PHC2$ and $SMC2$. Figure 2.14 shows the performance of the algorithms with different $U + I$. This shows that $PHC2$ and $SMC2$ are only useful for scenarios with relatively few changes between snapshots (less than say 2 percent of the records). By manipulating the IO cost equations, it is not hard to show that if $U + I$ is greater than 1.7%, $PHC1$ and $SMC1$ incur less IO than $PHC2$ and $SMC2$.

The next two columns in Table 2.2 (X_{256} and X_{1024}) examine the size of the intermediate files. In the case of the SM, PH, $SMC1$ and $PHC1$ algorithms, uncompressed intermediate files need to be saved. In the case of the $SMC2$ and $PHC2$ algorithms, the compressed versions of these files are constructed, which leads to a more economic disk usage. The *window* algorithm, on the other hand, does not construct any intermediate files.

The last column (labeled E) illustrates the probability of a missed matching record

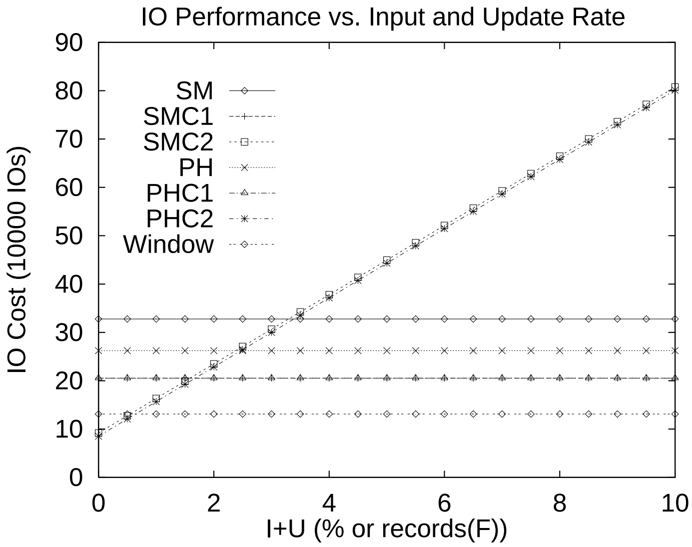

Figure 2.14: IO Cost and Varying Update and Insertion Rates

pair. Note that both record compression formats result in the same probability of error although the two formats have different compression factors. This is because the B field is compressed into a 32 bit integer for both formats.

In closing this section, we stress that the numbers we have shown are only illustrative. The gains of the various algorithms can vary widely. For example, if we assume very large records, then even modest compression can yield huge improvements. On the other hand, if we assume very large memories (relative to the file sizes), then the gains become negligible.

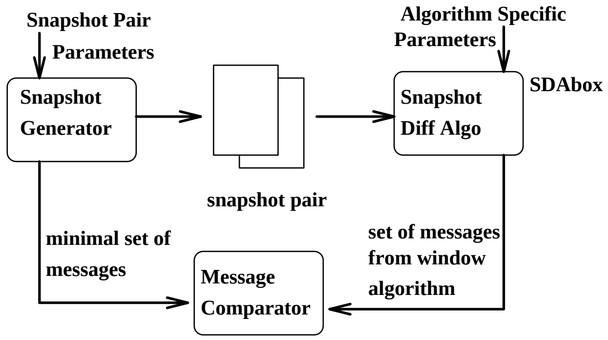

Figure 2.15: The Evaluation System

	Snapshot Parameters	Default Values
R	Size of B field Size of Record	150 bytes 156 bytes
F	Number of Records File Size	650,000 100 MB
	$disp_{avg}$	50,000 records
U	Number of Updates	20% of $records(F)$
	Window Parameters	Default Values
AB	Aging Buffer Size	8 MB
IB	Input Block Size	16K

Table 2.3: List of Parameters

2.4.2 Evaluation of Implemented Algorithms

In *WHIPS*, we have implemented the sort merge outerjoin and the *window* algorithm to compute the snapshot differentials. We have also built a snapshot differential algorithm evaluation system, which we used to study the effects of the snapshot pair distance on the number of useless delete-insert pairs that is produced by the *window* algorithm. We will also use the evaluation system to compare the actual running times of the *window* algorithm and the sort merge outerjoin algorithm. The evaluation system is depicted in Figure 2.15.

The snapshot generator produces a pair of synthetic snapshots with records of the form $\langle K,\ B\rangle$. The snapshot generator produces the two snapshots based on the following parameters: size of the B field, number of records, average record displacement ($disp_{avg}$) and percentage of updates. The first snapshot is constructed to have ordered K fields with the specified number of records and with the specified B field size. Table 2.3 shows the default snapshot pair parameters.

Conceptually, the second snapshot is produced by first copying the first snapshot. Each record R_j in the second snapshot is then swapped with a record that is, on average (uniformly distributed from 0 to $2 \cdot disp_{avg}$), $disp_{avg}$ records away from R_j. Based on the specified percentage of updates, some of the records in the second snapshot are modified to simulate updates. Insertions and deletions are not generated since they do not affect the number of useless delete-insert pairs produced. Notice that $disp_{avg}$ is not the distance measure between snapshots. It is a generator parameter that indirectly affects the resulting

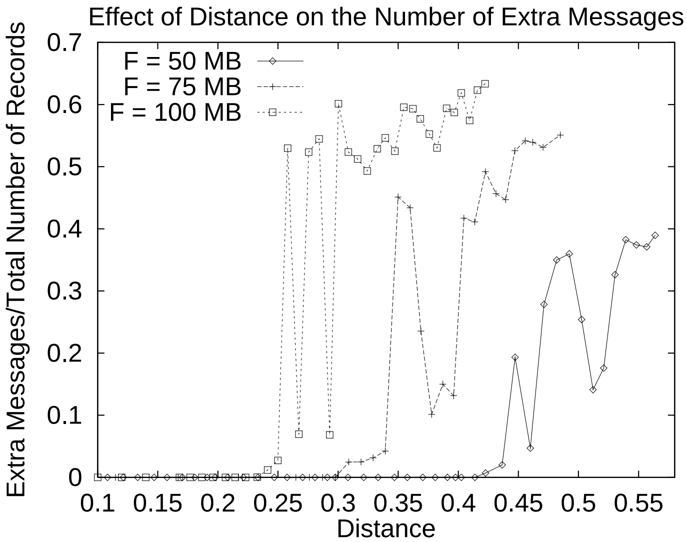

Figure 2.16: Effect of Distance on the Number of Extra Messages

distance. Thus, after generating the two snapshots, the actual distance of the two snapshots is then measured.

The two snapshots are then passed to the differential algorithm (in the SDA_{BOX}) being tested. Note that any of the previous algorithms discussed can be plugged into the SDA_{BOX}. In the experiments that we present here we focus on the *window* and the sort merge outerjoin algorithms. Algorithm specific parameters are also passed into the SDA_{BOX}. By varying the aging buffer size and the input buffer size parameters passed into the SDA_{BOX}, we can study how these parameters affect the *window* algorithm. Table 2.3 also shows the default *window* parameters. These were used unless the parameter was varied in an experiment.

After the snapshot differential algorithm is run, the output of the algorithm is compared to what was "produced" by the snapshot generator. Since the snapshot generator synthesized the two snapshots, it also knows the minimal set of differences of the two snapshots (which is the set of records of the first snapshot that it modified to produce the second). The message comparator can then check for the correctness of the output and count the number of extra messages.

The experiments we conducted enable us to evaluate, given the size of the aging buffer, and the size and the distance of the snapshots, how well the *window* algorithm will perform

File Size	$records(F)$	$dist_{crit}$	$disp_{crit\ MB}$
50 MB	162,500	0.44	5.11
75 MB	325,000	0.34	7.91
100 MB	650,000	0.24	11.2

Table 2.4: $dist_{crit}$ and $disp_{crit\ MB}$

in terms of the number of extra messages produced. In the first experiment, we varied the $disp_{avg}$ (and indirectly the distance) and measured the number of extra messages produced. This experiment was performed on three pairs of snapshots whose sizes ranged from 50 MB to 100 MB. Figure 2.16 shows that, as expected, as the distance of the snapshots increases beyond the capacity of the aging buffer, the number of extra messages increases. As the number of extra messages sharply rises, the graphs exhibit strong fluctuations. This is because the synthetic snapshots were produced randomly and only one experiment was done for each distance. (Only one experiment was done for each distance since it is hard to create two or more synthetic snapshot pairs with exactly the same distance.) For each snapshot size, there is a critical distance ($dist_{crit}$) which causes the *window* algorithm to start producing extra messages with the given aging buffer size.

For a system designer, it is helpful to translate $dist_{crit}$ into a critical average *physical* displacement. For instance, if the designer knows that records can only be displaced within a cylinder and the designer can only allocate 8 MB to each aging buffer, it is useful to know if the *window* algorithm produces few useless delete-insert messages in this scenario. We now capture this notion by first manipulating the definition of distance (Equation (2.3) in Section 2.3.1) to show that $dist_{crit}$ of the different snapshot pairs can be translated into a critical average physical displacement (in terms of MB). Since there are no insertions nor deletions in the synthetic snapshot pair, we can define a critical average *record* displacement (denoted as $disp_{crit}$) which is related to $dist_{crit}$ as shown in Equation (2.5).

$$dist_{crit} = \frac{\sum_{R_1 \epsilon F_1, R_2 \epsilon F_2, match(R_1,R_2)} |pos(R_1)-pos(R_2)|}{records(F)^2/2} \tag{2.4}$$

$$= \frac{records(F)\cdot disp_{crit}}{records(F)\cdot records(F)/2} \tag{2.5}$$

$$disp_{crit\ MB} = disp_{crit} \cdot R = dist_{crit} \cdot (records(F)/2) \cdot R \tag{2.6}$$

Using the size of the record (R), we can translate the $dist_{crit}$ into a critical average physical displacement (denoted as $disp_{crit\ MB}$ which is in terms of MB) using Equation

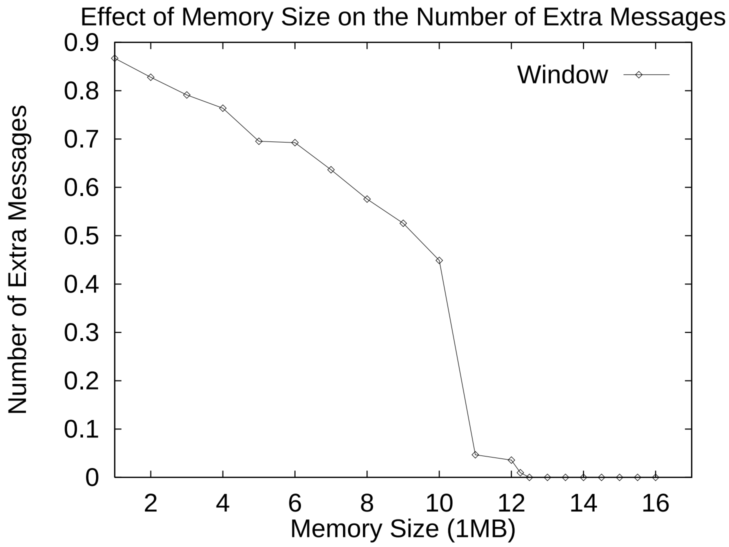

Figure 2.17: Effect of the Memory Size on the Number of Extra Messages

(2.6). Table 2.4 shows the result of the calculations for the different snapshot pairs. The $dist_{crit}$ of the snapshot pairs are estimated from Figure 2.16. This table shows, for example, that the *window* algorithm can tolerate an average physical displacement of about 11.2 MB given an aging buffer size of only 8 MB to compare 100 MB snapshots. Thus, if a system designer knows that the records can only be displaced within, say a page (which is normally smaller than 11.2 MB), then the designer can be assured that the *window* algorithm will not produce excessive amounts of extra messages.

In the next experiment, we focus on the 100 MB snapshots. Using the parameters listed in Table 2.3, we varied the size of the aging buffer from 1.0 MB to 16 MB. The $disp_{avg}$ was set at 50,000 with a resulting distance of 0.34, which is well above the $dist_{crit}$. Figure 2.17 shows that once the size of the aging buffer is at least 12.8 MB, no extra messages are produced. This is to be expected since we showed previously (Table 2.4) that the tolerable $disp_{crit\ MB}$ for the 100 MB file is 11.2 MB. Using the same snapshot pair, we also varied the input block size from 8 K to 80 K. The variation had no effect on the number of extra messages and we do not show the graph here. Again, this is to be expected, since the size of the aging buffer is much larger than the size of the input block. Thus, even if the input block size is varied, the window size stays the same. We also varied the record size (keeping the size of the snapshot constant) and this showed no effect on the number of extra messages

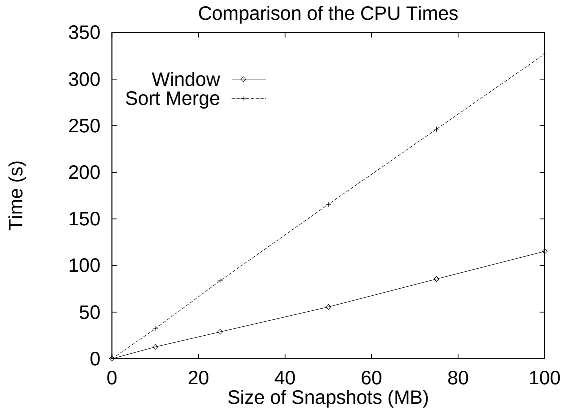

Figure 2.18: Comparison of the CPU Times

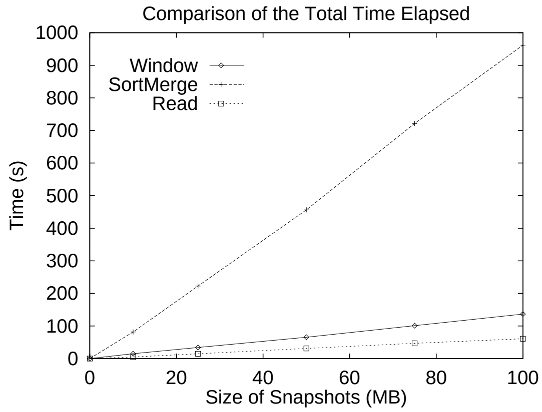

Figure 2.19: Comparison of the Total Times

produced.

Lastly we compared the CPU time and the clock time (which includes the IO time) that the *window* algorithm consumes to that of the sort merge outerjoin based algorithm. We ran the simulations on a DEC Alpha 3000/400 workstation running UNIX. We used the UNIX *sort* utility in the implementation of the sort merge outerjoin. (UNIX *sort* may not be the most efficient, but we believe it is adequate for the comparisons we wish to perform here.) We used the same input block size for both the *window* and the sort merge outerjoin algorithms (16 K). The $disp_{avg}$ of the two snapshots was set so that the resulting distance was 0.05 (within the $dist_{crit}$ for all file sizes). The analysis in the previous section illustrated that the *window* algorithm incurs fewer IO operations than the sort merge outerjoin algorithm. Figure 2.18 shows that the *window* algorithm is also significantly less CPU intensive than the sort merge based algorithm. As expected then, Figure 2.19 shows that the *window* algorithm outperforms the sort merge outerjoin in terms of clock time. Moreover, Figure 2.19 also shows that the CPU time is a small fraction of the clock time in the *window* algorithm. Thus, the IO comparisons of Section 2.4.1 are indeed useful.

2.5 Related Work

Snapshots were first introduced in [AL80]. Snapshots were then used in the system R* project at IBM Research in San Jose [Loh85]. The data warehouse snapshot can be updated by maintaining a log of the changes to the database. This approach was defined to be a *differential refresh* strategy in [KR87]. Note that one way of implementing the differential refresh strategy is to use triggers. Every time there is a change to the source, the trigger is set off and the appropriate change is recorded in the log. If snapshots were sent periodically, this was called the *full refresh* strategy. Again, we focus on the case where the source strategy is full refresh. [LHM+86] also presented a method for refreshing a snapshot that minimizes the number of messages sent when refreshing a snapshot. The method requires annotating the base tables with two columns for a record address and a timestamp. We cannot adopt this method in data warehousing since the sources are autonomous.

Reference [CRGMW96] investigates algorithms to find differences in hierarchical structures (*e.g.*, documents, CAD designs). Our focus here is on simpler, record structured differences, and on dealing with very large snapshots that may not fit in memory.

There has also been recent complementary work on copy detection of files and documents. Tools have been created to find similar files in a file system [MW94]. Copy detection mechanisms for documents have been proposed in an attempt to safeguard intellectual property on the Internet ([BDGM95], [SGM95]). These mechanisms ultimately provide as output the extent of the similarity of two files. The snapshot differential problem is concerned with detecting the specific differences of two files as opposed to measuring how different (or similar) two files are. Also related are [BGMF88] and [FWA86], which propose methods for finding differing pages in files. However, these methods can only detect a few changes and assume that no insertions or deletions have taken place.

The snapshot differential problem is also related to text comparison, for example, as implemented by UNIX *diff* and DOS *comp*. However, the text comparison problem is concerned with the order of the records. That is, it considers a *sequence* of records, while the snapshot differential problem is concerned with a *set* of records. Reference [HT77] outlines an algorithm that finds the longest common subsequence of the lines of the text, which is used in the UNIX *diff*. Report [LGM95] takes a closer look at how this algorithm can be adopted to solve the snapshot differential problem, although the solution is not as efficient as the ones presented here.

The methods for solving the snapshot differential problem proposed here are based on joins which have been well studied; [ME92] and [Sha86] are good surveys on join processing. The snapshot differential algorithms proposed here are used in the data warehousing system *WHIPS*. An overview of the system is presented in [HGMW+95]. After the changes of multiple sources are detected, the changes are integrated using methods discussed in [ZGMHW95].

Note that there are also cases wherein knowledge of the semantics of the information maintained at the warehouse helps make change detection simpler. For instance, if the warehouse keeps a history of all the data contained at the source, then it makes sense to simply pass complete snapshots to the warehouse. We have an outline of these special cases in report [LGM95].

2.6 Chapter Summary

We have defined the snapshot differential problem and discussed its importance in data warehousing. The algorithms we have proposed are "extensions" of traditional join algorithms, but take advantage of the semantics of the problem, *i.e.*, the flexibility allowed for snapshot differentials. All of our proposed algorithms are relatively simple, but we view this simplicity as essential for dealing efficiently with large files. In summary, we have the following results:

- By augmenting outerjoin algorithms with record compression, we have shown that very significant savings in IO cost can be attained. We have also illustrated that the probability that an error will occur if compression is used can be made negligible while still being efficient enough.
- We have introduced the *window* algorithm which works extremely well if the snapshots are not too different. Under this scenario, this algorithm outperforms the join-based algorithms and its running time is comparable to simply reading the snapshots once. We have defined the concept of *snapshot pair distance* to characterize quantitatively the scenarios where the algorithm is applicable.

Chapter 3

Maintenance of the Data Warehouse

3.1 Introduction

Once the changes to the remote data sources are detected (Chapter 2) and go through data cleaning (Chapter 6), the changes to the warehouse views need to be efficiently computed and installed. During the time that warehouse views are being updated, a process we call the "view-update," either OLAP queries are not processed or OLAP queries compete with the view-update for resources. To reduce OLAP down time or interference, it is critical to minimize the work involved in a view-update and shrink the view-update window.

As mentioned in Chapter 1, the derived data at the warehouse is often stored in materialized views. Previous work ([GL95], [Qua96]) has developed standard expressions for maintaining a large class of materialized views incrementally. However, there are still numerous alternative "strategies" for implementing these expressions, and these strategies incur different amounts of work and lead to different length update windows.

EXAMPLE 3.1.1 Let us consider the warehouse depicted by the view directed acyclic graph (VDAG) shown in Figure 3.1. There are four materialized views: *CUSTOMER*, *ORDER*, *LINEITEM*, and *V*. The edge from *V* to *CUSTOMER* indicates that view *V* is defined on view *CUSTOMER* (and similarly for the other edges). Unlike *V*, the *CUSTOMER*, *ORDER* and *LINEITEM* views are defined on remote data sources.

Periodically, the changes (*i.e.*, inserted, deleted and updated tuples) of *CUSTOMER*,

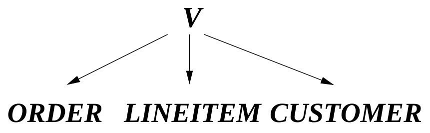

Figure 3.1: Example VDAG of Materialized Views

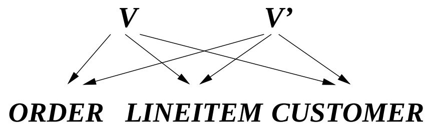

Figure 3.2: More Complex VDAG

ORDER and *LINEITEM* are computed from the changes of remote data sources. The changes of the remote data sources can be detected using algorithms discussed in Chapter 2, or other methods. View maintenance algorithms that handle remote and autonomous sources, like the algorithms developed in [ZGMHW95], may then be used to compute the changes to *CUSTOMER*, *ORDER* and *LINEITEM*. Once the changes of these views are obtained, the changes of V need to be computed, and the changes of all the views need to be installed. There are many ways to perform these update tasks using standard view maintenance expressions.

One strategy for updating V, denoted *Strategy 1*, is (as in [CGL+96]):

1. Compute the changes of V considering at once all the changes of *CUSTOMER*, *ORDER*, *LINEITEM*, and using the prior-to-update states of these views.

2. Install the changes of all four views. Installation of changes involves removing deleted tuples, adding inserted tuples, and changing updated tuples.

In *Strategy 2*, the changes of V are computed piecemeal, considering the changes of each of its base views one at a time:

1. Compute the changes of V only considering the changes of *CUSTOMER* (and the original state of the views).

2. Install the changes of *CUSTOMER*. (The following steps will see this new state.)

3. Compute the changes of V only considering the changes of *ORDER*.

4. Install the changes of *ORDER*. (This new state will be seen by the next step.)

5. Compute the changes of V only considering the changes of *LINEITEM*.

6. Install the changes of *LINEITEM*.

7. Install the changes of V.

In [GMS93], the correctness of both these strategies was discussed. Specifically, it was shown that both strategies compute the same final "database state" (*i.e.*, extension of all warehouse views). However, it was not shown how to choose among the strategies. In particular, the strategies can result in significantly different length update windows. For instance, we show later in the chapter that if *CUSTOMER*, *ORDER* and *LINEITEM* are TPC-D relations [Com], and V is defined using the TPC-D "Shipping Priority" Query, the update window can be two to three times longer if Strategy 1 is used instead of Strategy 2! We show experimentally that for views with more complex definitions than V, even larger disparities in update windows exist across different update strategies.

For the simple VDAG of Figure 3.1, there are 11 strategies in addition to Strategies 1 and 2. For instance, a slight variant of Strategy 2 computes the changes of V based on the changes of *LINEITEM* first, then *ORDER*, and then *CUSTOMER*. In some cases, this variant may have a shorter update window than Strategy 2, but in other cases Strategy 2 may be better. □

The previous example illustrated that even for a *single view*, there are many update strategies. Finding optimal strategies for a single view is one challenge we address in this chapter. In the next example, we illustrate that the update strategies for a *VDAG of views* cannot be constructed by simply picking the strategies for each view independently. In this chapter, we also address the problem of finding optimal strategies for a VDAG of views.

EXAMPLE 3.1.2 Let us consider the VDAG shown in Figure 3.2. This VDAG now includes a second view V' defined over *CUSTOMER*, *ORDER* and *LINEITEM*. Say we update V using Strategy 2 (Example 3.1.1), and V' is updated using the following *Strategy 3*:

1. Compute the changes of V' only considering the changes of *LINEITEM*.
2. Install the changes of *LINEITEM*. (These changes are visible to the following steps.)
3. Compute the V' changes considering the changes of *CUSTOMER* and *ORDER*.
4. Install the changes of *CUSTOMER* and *ORDER*.
5. Install the changes of V'.

Note that in Strategy 2, the fifth step occurs after the changes of *CUSTOMER* and *ORDER*, but not *LINEITEM*, have been installed. On the other hand, in Strategy 3 the third step occurs after the changes of *LINEITEM* have been installed, but not the changes of *CUSTOMER* and *ORDER*. Since only one of these states can be achieved,[1] we cannot combine Strategy 2 and Strategy 3. On the other hand, it is possible to combine Strategy 1 and Strategy 3 in a consistent manner. □

The previous example showed that we may not be able to construct a *correct* strategy for a VDAG of views by combining independently chosen single view strategies. Even if we can, the combined strategy may not be the best among all correct strategies. In this chapter, we define formally the notion of a correct update strategy for a VDAG of views, and we develop techniques to obtain correct and efficient update strategies for a VDAG of views.

One could argue that standard database query optimizers may be able to generate efficient view-update strategies by leveraging their proficiency in finding good plans for a query or even a set of queries. However, today's query optimizers assume that during the execution of the queries the database state does not change. As illustrated by our examples, view-update strategies employ sequences of computation and installation steps. More importantly, each step may change the database state, which in turn affects the rest of the steps. Hence, picking the best strategy involves:

- Choosing the set of queries (for update computations) and data manipulation expressions;
- Sequencing these queries and data manipulation expressions; and
- Ensuring that the chosen sequence results in the correct final database state.

To our knowledge, query optimizers do not handle these tasks. As a result, the warehouse administrator (WHA) is often saddled with the task of creating "update scripts" for the warehouse views. Since there are many alternative update strategies, the WHA can easily pick an inefficient update strategy, or even worse an update strategy that incorrectly updates the warehouse. Furthermore, the WHA may have to change the script frequently, since what strategy is best depends on the current size of the warehouse views and the current set of changes.

[1] We do not assume that multiple versions of the warehouse data are maintained.

In this chapter, we develop a framework for studying the space of update strategies. We make the following specific contributions:

- We characterize the correctness and optimality of update strategies for a VDAG of views.
- We develop a very efficient algorithm called *MinWorkSingle* that finds an update strategy that minimizes the work incurred in updating a single materialized view.
- Based on *MinWorkSingle*, we develop an efficient heuristic algorithm called *MinWork* that produces a good update strategy for a general VDAG of materialized views. We show that for a large class of VDAGs, the *MinWork* update strategy is actually the least expensive.
- We also develop a search algorithm called *Prune* that produces the least expensive update strategy for an even larger class of VDAGs.
- Based on performance experiments with a TPC-D scenario, we demonstrate that the *MinWorkSingle* and *MinWork* update strategies result in update windows that are significantly shorter than the update windows of conventional update strategies.

The rest of the chapter is organized as follows. In Section 3.2, we briefly review our warehouse model (discussed in Chapter 1). Alternative update strategies for a VDAG of views are discussed in Section 3.3. There we also define formally the problem of minimizing the work incurred. In Sections 3.4, 3.5 and 3.6 we present our algorithms and discuss practical issues surrounding their implementation. In Section 3.7, we show through experiments that our algorithms can significantly reduce update windows. Related work is discussed in Section 3.8.

3.2 Preliminaries

Warehouse Model

As a reminder of our warehouse model discussed in Chapter 1, Figure 3.3 shows a simple example of a VDAG with three base views (*i.e.*, V_1,V_2,V_3) and two derived views (*i.e.*, V_4, V_5). As a more concrete example, Figure 3.4 shows the VDAG representation of a warehouse that contains six TPC-D relations as base views. In this example, *ORDER* and

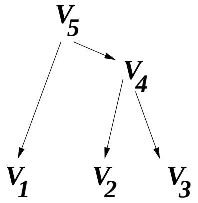

Figure 3.3: Example VDAG

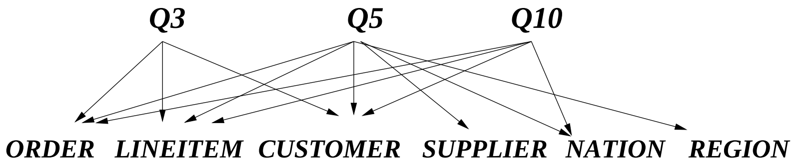

Figure 3.4: VDAG of a TPC-D Warehouse

LINEITEM represent fact tables, and the other base views represent dimension tables. The derived views $Q3$, $Q5$ and $Q10$ represent summary tables defined over the TPC-D base views. Often, derived views that further summarize $Q3$, $Q5$ and $Q10$ can also be defined.

We define $Level(V)$ to be the maximum distance of V to a base view. For instance, in Figure 3.3, $Level(V_1) = Level(V_2) = Level(V_3) = 0$, $Level(V_4) = 1$, and $Level(V_5) = 2$. We use $MaxLevel(G)$ to denote the maximum $Level$ value of any view in a VDAG G.

View Definitions and Maintenance Expressions

Recall from Chapter 1 that view definitions in our model (denoted $Def(V)$) involve *projection*, *selection*, *join*, and *aggregation* operations. For instance, views $Q3$, $Q5$ and $Q10$ of Figure 3.4 may be defined using TPC-D queries that are `SELECT-FROM-WHERE-GROUPBY` SQL statements.

An edge $(V_j \rightarrow V_i)$ in the VDAG means that V_i appears in $Def(V_j)$. Moreover, it implies that changes of V_i lead to V_j changes.[2] In Chapter 1, we discussed that the changes of V_i include inserted, deleted and updated tuples. For simplicity of presentation, we do not show explicitly these three types of deltas, instead lumping them together in a single delta table. We use *delta table* δV to represent the changes of V.

The changes of the base views arrive periodically at the warehouse. In today's warehouses, the period is often daily or weekly. The changes of the base views are then used to compute the changes of the derived views. If V is a derived view, *view maintenance expressions* based on $Def(V)$ are used to compute δV. For instance, if view V_4 in Figure 3.3 is defined as $\sigma_{\mathcal{P}}(V_2 \times V_3)$, the following standard maintenance expression ([GL95], [Qua96]) that uses three *terms* (*i.e.*, $\sigma_{\mathcal{P}}(\delta V_2 \times V_3)$, $\sigma_{\mathcal{P}}(V_2 \times \delta V_3)$, $\sigma_{\mathcal{P}}(\delta V_2 \times \delta V_3)$) computes δV_4.

$$\delta V_4 \leftarrow \sigma_{\mathcal{P}}(\delta V_2 \times V_3) \cup \sigma_{\mathcal{P}}(V_2 \times \delta V_3) \cup \sigma_{\mathcal{P}}(\delta V_2 \times \delta V_3) \quad (3.1)$$

[2] In some special cases (*e.g.*, if certain integrity constraints hold), V_i changes may not produce V_j changes.

When executing maintenance expressions like (3.1), the inserted, deleted and updated tuples in the delta tables must be handled appropriately. For instance, the term $\sigma_{\mathcal{P}}(\delta V_2 \times V_3)$ involves joining the deleted tuples of δV_2 with V_3 and storing them as deleted tuples of δV_4, and doing the same for the inserted and updated tuples of V_2.

After the changes of a view are computed, they are used in computing changes of other derived views, and installed. The install operation inserts the inserted tuples, and deletes the deleted tuples, and changes the updated tuples.

Compute and Install Expressions

We abstract maintenance computations by the function *Comp*. The formula for computing δV from the changes of the set of views $\mathcal{V}$ is denoted by $Comp(V, \mathcal{V})$. For instance, $Comp(V_4, \{V_2, V_3\})$ represents the δV_4 computation of Expression (3.1). As another example, $Comp(V_4, \{V_2\})$ represents the computation of the changes of V_4 based solely on the changes of V_2, *i.e.*, $\delta V_4 \leftarrow \sigma_{\mathcal{P}}(\delta V_2 \times V_3)$. Note that $Comp(V_4, \{V_2\})$, having just one term (*i.e.*, $\sigma_{\mathcal{P}}(\delta V_2 \times V_3)$), can be obtained from the expression for $Comp(V_4, \{V_2, V_3\})$ by assuming δV_3 is empty, and simplifying the expression.

We use $Inst(V)$ to denote the operation of installing δV into V. For example, $Inst(V_4)$ represents the expression $V_4 \leftarrow V_4 \cup \delta V_4$. We call expressions denoted by *Inst* install expressions. Even though standard view maintenance expressions can be used to obtain the changes of derived views, given the changes of views they are defined over, we show that there are numerous "strategies" for updating a derived view.

3.3 View and VDAG Strategies

We now define *view strategies* which are used to update a single view, and *VDAG strategies* which are used to update a VDAG of views. We also illustrate how one can define the space of correct VDAG strategies based on the notion of correct view strategies for the individual views of the VDAG. Finally, we formally define the "total-work minimization" (TWM) problem as finding the correct VDAG strategy that incurs the minimum amount of work.

3.3.1 View Strategies

For a view V defined over n views $V_1, \ldots, V_n$, there are many possible ways of updating V. We call each way a *view strategy*. One view strategy for V is to compute δV based on all of the changes $\{\delta V_1, \ldots, \delta V_n\}$ simultaneously as shown below.

$$\langle\ Comp(V, \{V_1, \ldots, V_n\}), Inst(V_1), \ldots, Inst(V_n), Inst(V)\ \rangle \tag{3.2}$$

Notice that view strategy (3.2) has two "stages", a stage for propagating the underlying changes (*i.e.*, using the $Comp$ expression), and a stage for installing the changes (*i.e.*, using the $Inst$ expressions). Having two stages is consistent with the framework proposed in [CGL+96] that a view is updated using a propagate stage and an install stage. In this chapter, we call strategies like (3.2) *dual-stage* view strategies.[3]

Another possible view strategy for V is to compute δV by considering each δV_i in $\{\delta V_1, \ldots, \delta V_n\}$ one at a time, as shown below.

$$\langle\ Comp(V, \{V_1\}), Inst(V_1), \ldots, Comp(V, \{V_n\}), Inst(V_n), Inst(V)\ \rangle \tag{3.3}$$

Each $Comp$ expression in view strategy (3.3) computes a subset of the changes of V. We assume that the changes computed by the various $Comp$ expressions for V are gathered in delta table δV, and eventually installed together by $Inst(V)$. We call view strategies like (3.3) *1-way* view strategies. Notice that view strategy (3.3) propagates the changes of V_1 first, then of V_2, and so on. For a view defined over n views, there are a total of $n!$ 1-way view strategies that can be obtained by using different change propagation orders.[4] For instance, another 1-way view strategy for V shown below processes the changes of V_n first, then of V_{n-1}, and so on. As we will see in subsequent sections, view strategies (3.2), (3.3) and (3.4) may incur significantly different amounts of work.

$$\langle\ Comp(V, \{V_n\}), Inst(V_n), \ldots, Comp(V, \{V_1\}), Inst(V_1), Inst(V)\ \rangle \tag{3.4}$$

Dual-stage view strategies as well as 1-way view strategies have been proposed in the literature ([GMS93], [CGL+96]). However, the issue of finding optimal view strategies has not been studied. Moreover, we will see later that difficult problems arise when constructing correct and efficient VDAG strategies by combining optimal view strategies for individual views of the VDAG.

[3] Actually, for a view defined over n other views, a total of $(n+1)!$ dual-stage view strategies can be obtained by reordering the $Inst$ expressions. That is, once $\{\delta V_1, \ldots, \delta V_n\}$ are used to compute δV, the changes can be installed in any order. Fortunately, we only need to consider one dual-stage strategy per view since all dual-stage view strategies for a given view can be shown to incur the same amount of work.

[4] Actually, there are $2(n!)$ 1-way view strategies because the last two $Inst$ expressions, *e.g.*, $Inst(V_n)$ and $Inst(V)$ in view strategy (3.3), can be swapped. However, it can be shown that swapping these expressions does not affect the work incurred by the view strategy. Hence, we only consider $n!$ 1-way view strategies.

Beyond the 1-way and dual-stage view strategies, there is a multitude of other correct view strategies. To see this, we can look at a 1-way view strategy as one that partitions $\{\delta V_1, \ldots, \delta V_n\}$ into n singleton sets, and processes the sets, one at a time. On the other hand, a dual-stage view strategy does not partition $\{\delta V_1, \ldots, \delta V_n\}$ at all, and processes all the changes simultaneously. Other ways of partitioning the view set will yield other view strategies.

Once the partitions are decided upon, the propagation order among the various partitions needs to be chosen. The combined choices of partitioning and their order of processing yields

To illustrate the enormity of the space of view strategies, Table 3.1 shows the number of view strategies for a view defined over n views, where n ranges from 1 to 6. According to Table 3.1, views $Q3$, $Q5$, and $Q10$ of the TPC-D VDAG (Figure 3.4) have 13, 4683, and 75 view strategies respectively.

n	# of view strategies
1	1
2	3
3	13
4	75
5	541
6	4683

Table 3.1: Number of View Strategies for a View Defined Over n Views

Table 3.1 actually counts the number of correct view strategies. In Definition 3.3.1, we formally describe the notion of correctness of a view strategy. Intuitively, conditions **C1** and **C2** state that all the changes must be propagated and installed by a correct view strategy. That is, certain *Comp* and *Inst* expressions must be in the correct view strategy.[5] On the other hand, conditions **C3**, **C4**, and **C5** state that the *Comp* and *Inst* expressions must be in a particular order. Specifically, condition **C3** states that δV_i must not be installed until all *Comp* expressions that use it are done. Condition **C4** states that when the changes of V are computed using multiple *Comp* expressions, the changes of a view used in a *Comp* expression must be installed before the next *Comp* expression for V can be executed.

[5]Conditions **C1** and **C2**, and our algorithms can be extended to avoid using expressions that propagate and install δV_i when δV_i is empty.

Condition **C5** states that the changes computed for V can only be installed after they are completely computed. Finally, condition **C6** states that there are no duplicate expressions in the correct view strategy.

Definition 3.3.1 (Correct View Strategy) Let $E_i < E_j$ if expression E_i is before expression E_j in the view strategy. Given a view V defined over a set of views $\mathcal{V}$, a correct view strategy $\overrightarrow{\mathcal{E}}$ for V is a sequence of $Comp$ and $Inst$ expressions satisfying the following conditions.

- **C1:** $\forall V_i \in \mathcal{V}$: $(Comp(V, \{\ldots V_i \ldots\}) \in \overrightarrow{\mathcal{E}})$.
- **C2:** $\forall V_i \in (\mathcal{V} \cup \{V\})$: $(Inst(V_i) \in \overrightarrow{\mathcal{E}})$.
- **C3:** $\forall V_i \in \mathcal{V}$: $(Comp(V, \{\ldots V_i \ldots\}) < Inst(V_i))$.
- **C4:** $\forall V_i$: $\forall V_j$: $(Comp(V, \{\ldots V_i \ldots\}) < Comp(V, \{\ldots V_j \ldots\})) \Rightarrow$
  $(Inst(V_i) < Comp(V, \{\ldots V_j \ldots\}))$.
- **C5:** $\forall V_i \in \mathcal{V}$: $(Comp(V, \{\ldots V_i \ldots\}) < Inst(V))$.
- **C6:** $\forall E_i \in \overrightarrow{\mathcal{E}}$: $\forall E_j \in \overrightarrow{\mathcal{E}}$: $(i \neq j) \Rightarrow (E_i \neq E_j)$.

□

Notice that combinations of these conditions avoid incorrect view strategies that are not explicitly prohibited in the conditions. For instance, because of conditions **C3** and **C4**, it is not possible to have two $Comp$ expressions that propagate δV_i. For instance, both $Comp(V, \{V_i, V_j\})$ and $Comp(V, \{V_i, V_k\})$ cannot be simultaneously present in a correct view strategy. More specifically, **C3** states that $Inst(V_i)$ must be after both $Comp$ expressions. On the other hand, if $Comp(V, \{V_i, V_j\}) < Comp(V, \{V_i, V_k\})$, **C4** states that $Inst(V_i)$ must be before $Comp(V, \{V_i, V_k\})$, a contradiction. Similarly, $Comp(V, \{V_i, V_j\}) < Comp(V, \{V_i, V_k\})$ also leads to a contradiction.

Note also that for a base view V which is not defined over any warehouse views (*i.e.*, $\mathcal{V} = \{\ \}$), V's correct view strategy is $\langle\ Inst(V)\ \rangle$.

3.3.2 VDAG Strategies

Like a view strategy, a *VDAG strategy* is simply a sequence of compute and install expressions. Informally speaking, a *correct* VDAG strategy uses a correct view strategy to update each view in the VDAG.

EXAMPLE 3.3.1 Consider the VDAG shown in Figure 3.3. A VDAG strategy should indicate how changes are propagated to all the views. One possible VDAG strategy propagates the changes of V_2 to V_4, then propagates the changes of V_3 to V_4, then propagates the changes of V_4 to V_5, and finally propagates the changes of V_1 to V_5.

$$\begin{aligned}\langle\ & Comp(V_4, \{V_2\}), Inst(V_2), Comp(V_4, \{V_3\}), Inst(V_3), \\ & Comp(V_5, \{V_4\}), Inst(V_4), Comp(V_5, \{V_1\}), Inst(V_1), Inst(V_5)\ \rangle \end{aligned} \tag{3.5}$$

Note that VDAG strategy (3.5) "uses" (contains as a subsequence) the following correct view strategies for V_4 and V_5 respectively.

$$\langle\ Comp(V_4, \{V_2\}), Inst(V_2), Comp(V_4, \{V_3\}), Inst(V_3), Inst(V_4)\ \rangle$$
$$\langle\ Comp(V_5, \{V_4\}), Inst(V_4), Comp(V_5, \{V_1\}), Inst(V_1), Inst(V_5)\ \rangle$$

Also, for any base view V_i (*i.e.*, V_1, V_2, V_3), VDAG strategy (3.5) "uses" $\langle\ Inst(V_i)\ \rangle$. □

The previous example illustrated that a correct VDAG strategy uses correct view strategies to update each view in the VDAG. However, we know that starting from a set of correct view strategies, one for each view of the VDAG, we may not be able to construct a correct VDAG strategy (see Example 3.1.2 of Section 3.1). In Sections 3.5 and 3.6, we present algorithms that not only find correct VDAG strategies but also ensure that the strategies they produce are very efficient. In the rest of this section, we formalize our notions of correctness and efficiency of VDAG strategies. First, we define the concept of a view strategy "used" by a VDAG strategy.

Definition 3.3.2 (**View Strategy Used by a VDAG Strategy**) Given a VDAG strategy $\overrightarrow{\mathcal{E}}$, and a view V_j defined over views $\mathcal{V}$, the view strategy *used* by $\overrightarrow{\mathcal{E}}$ for V_j is the subsequence $\overrightarrow{\mathcal{E}_j}$ of $\overrightarrow{\mathcal{E}}$ composed of the following expressions: (1) $Comp(V_j, \{...\})$; (2) $Inst(V_j)$; and (3) $Inst(V_i)$, where $V_i \in \mathcal{V}$. □

The next definition formalizes the conditions that are required of a correct VDAG strategy. Condition **C7** states that a correct VDAG strategy must update each view using a correct view strategy. Condition **C8** states that a correct VDAG strategy can only propagate changes of V_j after they have been computed. Condition **C8** implicitly imposes an order between expressions from view strategies of different views in the VDAG.

Definition 3.3.3 (**Correct VDAG Strategy**) Given a VDAG G with views $\mathcal{V}$ and edges $\mathcal{A}$, a *correct* VDAG strategy is a sequence of $Comp$ and $Inst$ expressions $\overrightarrow{\mathcal{E}}$ such that

- **C7:** $\forall V_i \in \mathcal{V}$: $\overrightarrow{\mathcal{E}}$ uses a correct view strategy $\overrightarrow{\mathcal{E}_i}$ for V_i.
- **C8:** $\forall V_i \in \mathcal{V}$: $\forall V_j \in \mathcal{V}$: $\forall V_k \in \mathcal{V}$: $(Comp(V_k, \{\ldots V_j \ldots\}) \in \overrightarrow{\mathcal{E}}$ and $Comp(V_j, \{\ldots V_i \ldots\}) \in \overrightarrow{\mathcal{E}}) \Rightarrow (Comp(V_j, \{\ldots V_i \ldots\}) < Comp(V_k, \{\ldots V_j \ldots\}))$.

□

3.3.3 Problem Statement

We use a function $Work$ to represent the amount of work involved in executing an expression – $Comp$ or $Inst$. Given a VDAG strategy $\overrightarrow{\mathcal{E}} = \langle E_1, \ldots, E_n \rangle$, we define $Work(\overrightarrow{\mathcal{E}})$ as $\sum_{i=1..n} Work(E_i)$. Notice that $Work(E_i)$ depends on the expressions that precede E_i, since these expressions change the database state that E_i is executed in. The problem we address in this chapter is stated as follows.

Definition 3.3.4 (**Total-Work Minimization (TWM) Problem**) Given a VDAG, find the correct VDAG update strategy $\overrightarrow{\mathcal{E}}$ such that $Work(\overrightarrow{\mathcal{E}})$ is minimized. □

Since TWM is only concerned with correct VDAG strategies, henceforth, "VDAG strategies" refer only to "correct VDAG strategies." Similarly, "view strategies" refer only to "correct view strategies."

In order to estimate $Work(E_i)$, various metrics can be used. We adopt a metric called *linear work metric*. The linear work metric is a simple metric that focuses on the essential components of the work involved in executing the $Comp$ and $Inst$ expressions. The algorithms that we develop in this chapter produce optimal update strategies under the linear work metric. In Section 3.7, we study the relative performance of various update strategies for the TPC-D VDAG by executing the strategies on a commercial RDBMS, and measuring the corresponding update windows. Our study demonstrates that the strategies produced by our algorithms have significantly shorter update windows than conventional update strategies. The results of the study suggest that the linear work metric employed by our algorithms effectively tracks real-world execution of update strategies.

The linear work metric is based on the following execution model of $Comp$ expressions. Recall that $Comp$ typically represents a maintenance expression with a set of terms (*e.g.*, Expression (3.1) of Section 3.2 has three terms). In general, we assume that a compute expression of the form $Comp(W, \mathcal{Y})$ has a total of $2^{|\mathcal{Y}|} - 1$ terms, where each term considers a combination of delta or non-delta forms of the views in $\mathcal{Y}$. For example, in

$Comp(W, \{V_1, V_2\})$, one term evaluates the changes of W based on δV_1 and V_2, a second term computes W changes based on V_1 and δV_2, and the third term considers δV_1 and δV_2. Each of these terms must in addition consider the rest of the views that participate in the definition of W. In our example, if W is defined over V_1, V_2 and V_3, then the first term of $Comp(W, \{V_1, V_2\})$ will have as input δV_1, V_2 and V_3; the second term will have as input V_1, δV_2 and V_3; and the final term will have as input δV_1, δV_2 and V_3. We consider an execution model that evaluates each of these terms separately. Thus, the work estimate for a *Comp* expression is obtained by estimating the work for each of its terms and adding up these estimates.

Notice that our term-execution model is independent of the specifics of the view definitions. Incremental view maintenance expressions for views involving arbitrary select, project, join operations, followed by arbitrary aggregate operations fit this pattern. Thus, the results we develop in this chapter are valid for all these maintenance expressions. We now formally state our work metric based on the term-execution model discussed above.

Definition 3.3.5 (**Linear Work Metric**) The work estimate for an *Inst* expression is proportional to the size of the set of changes being installed. The estimate for a *Comp* expression is the sum of the estimates for each of its terms; the estimate for a term is proportional to the sum of the sizes of the operands of the term. □

EXAMPLE 3.3.2 Consider the VDAG shown in Figure 3.3, with V_4 defined as $\sigma_{\mathcal{P}}(V_2 \times V_3)$. $Comp(V_4, \{V_2\})$ has one term: $\sigma_{\mathcal{P}}(\delta V_2 \times V_3)$. Its work estimate is $c \cdot (|\delta V_2| + |V_3|)$, where c is a proportionality constant. Similarly, the estimate for $Comp(V_4, \{V_2, V_3\})$ can be derived (by considering its 3 terms) as $c \cdot ((|\delta V_2| + |V_3|) + (|\delta V_3| + |V_2|) + (|\delta V_2| + |\delta V_3|))$. Finally, note that the work estimate for $Inst(V_4)$ is $i \cdot |\delta V_4|$, where i is a proportionality constant. □

The linear work metric is similar to metrics that have been used in state-of-the-art algorithms for warehouse design ([HRU96], [SDN98]), and it can be quite effective in modeling complex update computations. Estimating the work of an install expression as being proportional to the size of the delta table is reasonable because the expression needs to scan in the delta table to install the changes. When estimating the work of a compute expression, we note that each term in the compute expression contains at least one delta table. Since delta tables tend to be small, all intermediate results in the evaluation of a term tend to be small. Therefore, the work incurred in evaluating a term is often dominated by scanning into memory the term's operands. Accordingly, we estimate the work of a term as being proportional to the sum of the sizes of its operands. Then, the work estimate of a compute expression is obtained by adding the work estimates of all the terms in the compute expression.

3.4 Optimal View Strategy

In this section, we present algorithm *MinWorkSingle* that produces an optimal view strategy for a given view, under the linear work metric. In Section 3.7, we will show that even if the underlying database does not have a linear work metric, the *MinWorkSingle* view strategy is still very efficient.

We showed previously that there are numerous possible view strategies for a single view. Fortunately, under the linear work metric, we can restrict our attention to 1-way view strategies only.

Theorem 3.4.1 *For any given view, the best 1-way view strategy is optimal over the space of all view strategies.* □

The detailed proof of Theorem 3.4.1, and of other theorems and lemmas that follow, are furnished in Appendix A. The basic intuition is that in any view strategy for V that is not 1-way, a *Comp* expression that computes the changes of V based on multiple deltas can be replaced by a set of *Comp* expressions each involving a single delta such that the total work of this set of *Comp* expressions is smaller than the work incurred by the replaced *Comp* expression.

Theorem 3.4.1 is very significant because the set of 1-way view strategies is much smaller than the set of all view strategies. For instance, the view $Q5$ in Figure 3.4 has a total of 4683 view strategies, out of which only 720 are 1-way. Thus, the search for an optimal view strategy can be limited to the set of 1-way view strategies. Next, we will present another theorem that helps us avoid examining all the 1-way view strategies and identify the best 1-way strategy very efficiently. The following example illustrates how the various 1-way view strategies differ in efficiency and it provides the basic intuition behind the next theorem.

EXAMPLE 3.4.1 Let us again consider view V_4 (Figure 3.3) defined over V_2 and V_3, and compare the two 1-way view strategies for V_4 shown below.

$$\langle\ Comp(V_4, \{V_2\}), Inst(V_2), Comp(V_4, \{V_3\}), Inst(V_3), Inst(V_4)\ \rangle \tag{3.6}$$

$$\langle\ Comp(V_4, \{V_3\}), Inst(V_3), Comp(V_4, \{V_2\}), Inst(V_2), Inst(V_4)\ \rangle \tag{3.7}$$

Clearly, the work incurred by the $Inst$ expressions (*i.e.*, $Inst(V_2)$, $Inst(V_3)$, $Inst(V_4)$) are the same. This is not the case for the $Comp$ expressions. Although the same set of $Comp$ expressions are used, the view extensions accessed by the $Comp$ expressions are different.

To illustrate, we use V_2' to denote V_2 after δV_2 is installed. Similarly, V_3' denotes V_3 after δV_3 is installed. In general, the expression $Comp(V_4, \{V_2\})$ in view strategy (3.6) uses δV_2, and V_3, and possibly V_4. On the other hand, the same expression $Comp(V_4, \{V_2\})$ in view strategy (3.7) uses δV_2, and V_3', and possibly V_4. Hence, the only difference in the use of $Comp(V_4, \{V_2\})$ in the two view strategies is that V_3' is used in view strategy (3.7), while V_3 is used in view strategy (3.6).

In general, the earlier δV_3 is installed in a view strategy, the more often will V_3' be used by the compute expressions in the view strategy. If it so happens that V_3' is larger than V_3, then using V_3' is more expensive than using V_3. In this case, it is good to delay the installation of δV_3. On the other hand, if V_3' is smaller than V_3, then it is good to install δV_3 as early as possible.

In fact, under a linear work metric we can be much more precise about the installation and propagation order of the various changes. For instance, if we first propagate and install the changes of V_3 (as in view strategy (3.7)), any subsequent compute expression that used to access V_3, will access V_3' instead. Hence, the work incurred by these compute expressions is increased by $c \cdot (|V_3'| - |V_3|)$. (Of course, if $(|V_3'| - |V_3|)$ is negative, the work incurred actually decreases.) Similarly if we first propagate and install the changes to V_2 (as in view strategy (3.6)), the work incurred by subsequent compute expressions is increased by $c \cdot (|V_2'| - |V_2|)$. Hence, in this example, we would want to propagate and install the changes of V_3 before the changes of V_2 if $(|V_3'| - |V_3|) < (|V_2'| - |V_2|)$. □

The example illustrated how an optimal 1-way view strategy for some view V can be obtained. Assuming V is defined over the views $\mathcal{V}$, we first obtain a *view ordering* $\overrightarrow{\mathcal{V}}$ that arranges the views in $\mathcal{V}$ in increasing $|V_i'| - |V_i|$ values based on the current set of changes. Given $\overrightarrow{\mathcal{V}}$, an optimal 1-way view strategy is the one that propagates and installs the changes in an order *consistent* with $\overrightarrow{\mathcal{V}}$. A 1-way view strategy for V is consistent with

Algorithm 3.4.1 *MinWorkSingle*
Input: V, defined over views $\mathcal{V}$
Output: an optimal view strategy $\overrightarrow{\mathcal{E}}$ for V
1. $\overrightarrow{\mathcal{E}} \leftarrow \langle\ \rangle$
2. For each $V_i \in \mathcal{V}$ estimate $|V_i'| - |V_i|$ based on the current set of changes
3. $\overrightarrow{\mathcal{V}} \leftarrow$ views in $\mathcal{V}$ ordered by increasing $|V_i'| - |V_i|$ values
4. For each $V_i \in \overrightarrow{\mathcal{V}}$ in order
    5. Append $Comp(V, \{V_i\})$ to $\overrightarrow{\mathcal{E}}$
    6. Append $Inst(V_i)$ to $\overrightarrow{\mathcal{E}}$
7. Append $Inst(V)$ to $\overrightarrow{\mathcal{E}}$
8. Return $\overrightarrow{\mathcal{E}}$

$\diamond$

Figure 3.5: *MinWorkSingle* Algorithm

a view ordering $\overrightarrow{\mathcal{V}}$ if for every $Inst(V_i) < Inst(V_j)$ in the strategy (where $V_i \neq V$ and $V_j \neq V$), then $V_i < V_j$ in $\overrightarrow{\mathcal{V}}$.

Theorem 3.4.2 *Given a view V defined over the views $\mathcal{V}$, let the view ordering $\overrightarrow{\mathcal{V}}$ arrange the views in increasing $|V_i'| - |V_i|$ values, for each $V_i \in \mathcal{V}$. Then, a 1-way view strategy for V that is consistent with $\overrightarrow{\mathcal{V}}$ will incur the least amount of work among all the 1-way view strategies for V.* $\square$

The main intuition behind the proof (in Appendix A) was illustrated by Example 3.4.1.

Based on Theorem 3.4.1 and Theorem 3.4.2, algorithm *MinWorkSingle* (Figure 3.5) produces an optimal view strategy. The view strategy produced by *MinWorkSingle* is correct since it satisfies the conditions for a correct view strategy (Definition 3.3.1). Specifically, *MinWorkSingle* appends all the necessary $Comp$ and $Inst$ expressions (Lines 5–7) required by **C1** and **C2**. By appending $Inst(V_i)$ right after $Comp(V, \{V_i\})$ and before the next $Comp$ expression, *MinWorkSingle* guarantees that the output view strategy satisfies **C3** and **C4**. Appending $Inst(V)$ last ensures that **C5** is satisfied. Since *MinWorkSingle* does not duplicate any expression, **C6** is satisfied.

We summarize the behavior of algorithm *MinWorkSingle* in the following theorem.

Theorem 3.4.3 *Given a view defined over n other views in the warehouse, MinWorkSingle finds an optimal view strategy for the view in $O(n\ log\ n)$ time.* $\square$

3.5 Minimizing Total Work

We have seen that for a derived view V, a 1-way view strategy consistent with a certain view ordering based on the current set of changes of the views that V is defined on is optimal. In this section, we show a similar result for VDAG strategies. That is, for a VDAG, we show that a "1-way VDAG strategy" consistent with a certain ordering of all the VDAG views based on the current set of changes is optimal among all VDAG strategies. Based on this result, we present an efficient algorithm to find optimal VDAG strategies.

Unlike in the case of view strategies, it is not always possible to obtain a "1-way VDAG strategy" consistent with a given view ordering. In such cases, our algorithm finds VDAG strategies that may not be optimal. In this section, we study the conditions required to be satisfied by a VDAG for our algorithm to obtain an optimal VDAG strategy. Based on these conditions, we identify large classes of VDAGs for which optimal VDAG strategies are guaranteed by our algorithm.

3.5.1 Optimal VDAG Strategies

Intuitively, a VDAG strategy that uses good view strategies for its derived views tends to incur less amount of work than one that uses worse view strategies. In the following theorem we capture the relationship between optimal VDAG strategies and the view strategies they use.

Theorem 3.5.1 *Given a VDAG G, a VDAG strategy for G that uses optimal view strategies for all the views of G is optimal over all VDAG strategies for G.* □

Observe that all VDAG strategies for G incur the same amount of work for their $Inst$ expressions. In the proof (presented in Appendix A), we further argue that a VDAG strategy that uses optimal view strategies minimizes the total amount of work incurred by the $Comp$ expressions.

From Section 3.4, we know that given a view V_i that is defined over views $\mathcal{V}_i$, the 1-way view strategy $\overrightarrow{\mathcal{E}_i}$ that is consistent with $\overrightarrow{\mathcal{V}_i}$ that orders the views in $\mathcal{V}_i$ in increasing $|V'|-|V|$ values is optimal. It can be shown that $\overrightarrow{\mathcal{E}_i}$ is also consistent with the view ordering $\overrightarrow{\mathcal{V}}$ that orders *all* of the VDAG views in increasing $|V'| - |V|$ values. This view ordering is called the *desired view ordering*. Note that the desired view ordering depends on the current set of changes.

We say a VDAG strategy is a *1-way VDAG strategy* if it only uses 1-way view strategies. Furthermore, a VDAG strategy is *consistent* with $\overrightarrow{\mathcal{V}}$ if it only uses view strategies that are consistent with $\overrightarrow{\mathcal{V}}$. Clearly, a 1-way VDAG strategy that is consistent with the desired view ordering uses only optimal view strategies. It follows from Theorem 3.5.1 that this VDAG strategy is optimal.

Theorem 3.5.2 *For any VDAG G, a 1-way VDAG strategy for G that is consistent with a desired view ordering is an optimal VDAG strategy for G.* □

We illustrate the interaction between Theorem 3.5.1 and Theorem 3.5.2 by the following example.

EXAMPLE 3.5.1 Consider the VDAG shown in Figure 3.6 (same as Figure 3.3 copied over for local reference). Let $(|V_4'| - |V_4|) < (|V_2'| - |V_2|) < (|V_1'| - |V_1|) < (|V_3'| - |V_3|) < (|V_5'| - |V_5|)$ based on the current set of changes. That is, a desired view ordering $\overrightarrow{\mathcal{V}}$ is $\langle\ V_4, V_2, V_1, V_3, V_5\ \rangle$.

A 1-way VDAG strategy consistent with a desired view ordering is

$$\langle\ Comp(V_4, \{V_2\}), Inst(V_2), Comp(V_4, \{V_3\}), Inst(V_3),$$
$$Comp(V_5, \{V_4\}), Inst(V_4), Comp(V_5, \{V_1\}), Inst(V_1), Inst(V_5)\ \rangle.$$

The above VDAG strategy is optimal and uses the following optimal view strategies for V_4 and V_5:

$$\langle\ Comp(V_4, \{V_2\}), Inst(V_2), Comp(V_4, \{V_3\}), Inst(V_3), Inst(V_4)\ \rangle.$$
$$\langle\ Comp(V_5, \{V_4\}), Inst(V_4), Comp(V_5, \{V_1\}), Inst(V_1), Inst(V_5)\ \rangle.$$

□

3.5.2 Expression Graphs

We have established that a 1-way VDAG strategy consistent with a desired view ordering is optimal. Here, we describe our approach to constructing such a VDAG strategy.

For a given VDAG G, all possible 1-way VDAG strategies for G have the same set of expressions, called the *1-way expressions* of G. The set of 1-way expressions of a given VDAG G contains $Comp(V_j, \{V_i\})$ whenever view V_j is defined over view V_i in G. Also

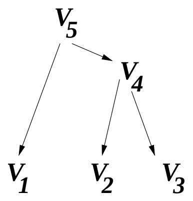

Figure 3.6: VDAG

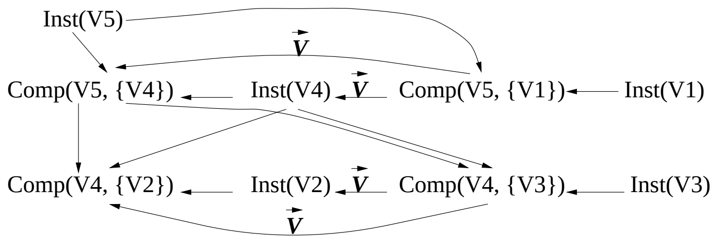

Figure 3.7: Expression Graph (EG)

included is an $Inst(V_i)$ expression for each view V_i in G. The various 1-way VDAG strategies for G differ in the sequencing of the 1-way expressions of G. The correctness conditions (of Section 3.3) impose certain dependencies among these 1-way expressions (*e.g.*, for any two derived views V_i and V_j, $Comp(V_j, \{V_i\})$ must follow $Comp(V_i, \{...\})$). Additional dependencies are imposed when we attempt to find VDAG strategies that are consistent with a particular view ordering (*e.g.*, for a derived view V defined over views V_i and V_j, if V_i precedes V_j in the view ordering, $Comp(V, \{V_i\})$ must precede $Comp(V, \{V_j\})$). A 1-way VDAG strategy for G consistent with a given view ordering is a permutation of the set of 1-way expressions of G that satisfies all dependencies.

We use the notion of an *expression graph* to capture the set of 1-way expressions of a VDAG and their dependencies. Given a VDAG G and a view ordering $\overrightarrow{\mathcal{V}}$, the expression graph of G with respect to $\overrightarrow{\mathcal{V}}$, denoted $EG(G, \overrightarrow{\mathcal{V}})$, has the 1-way expressions of G as its nodes. The expression graph has an edge from expression E_j to expression E_i if a dependency dictates that E_j must follow E_i. Once we construct an expression graph for a VDAG with respect to a desired view ordering, we can obtain an optimal VDAG strategy by topologically sorting the expression graph.

Theorem 3.5.3 *Given a VDAG G, if $EG(G, \overrightarrow{\mathcal{V}})$ is acyclic where $\overrightarrow{\mathcal{V}}$ is a desired view ordering, a topological sort of $EG(G, \overrightarrow{\mathcal{V}})$ yields an optimal VDAG strategy for G.* $\square$

The proof of the theorem is in Appendix A where we show that the topological sort of $EG(G, \overrightarrow{\mathcal{V}})$ results in a 1-way VDAG strategy that is consistent with the desired view ordering $\overrightarrow{\mathcal{V}}$. We now illustrate the generation of an optimal VDAG strategy, based on this theorem.

EXAMPLE 3.5.2 Consider the VDAG shown in Figure 3.6. Let a desired view ordering $\overrightarrow{\mathcal{V}}$ be $\langle V_4, V_2, V_1, V_3, V_5 \rangle$ based on the current set of changes (as in Example 3.5.1).

Figure 3.7 shows the expression graph constructed from the VDAG and the view ordering $\overrightarrow{\mathcal{V}}$. Each derived view has a set of $Comp$ expressions, one for each view it is defined over. Each view in the VDAG has an $Inst$ expression.

The edges of the expression graph indicate the dependencies. For instance, the edge from $Comp(V_5, \{V_4\})$ to $Comp(V_4, \{V_2\})$ indicates that the former should appear after the latter in any 1-way VDAG strategy for this VDAG. This dependency is due to **C8**.

Some edges of the expression graph are shown with a label $\overrightarrow{\mathcal{V}}$ to emphasize that the corresponding dependencies are due to the view ordering with which the 1-way VDAG strategy should be consistent. For instance, the edge from $Comp(V_4, \{V_3\})$ to $Comp(V_4, \{V_2\})$ indicates that $\overrightarrow{\mathcal{V}}$ requires that the changes of V_2 be propagated before the changes of V_3 (note that $V_2 < V_3$ in $\overrightarrow{\mathcal{V}}$).

The expression graph of this example happens to be acyclic. So, a topological sort of the graph is possible, and yields a 1-way VDAG strategy that is consistent with the view ordering $\overrightarrow{\mathcal{V}}$. For instance, we can obtain the following VDAG strategy:

$$\begin{aligned}&\langle\ Comp(V_4, \{V_2\}), Inst(V_2), Comp(V_4, \{V_3\}), Inst(V_3),\\ &\ \ Comp(V_5, \{V_4\}), Inst(V_4), Comp(V_5, \{V_1\}), Inst(V_1), Inst(V_5)\ \rangle.\end{aligned}$$

Note that this is the same optimal VDAG strategy that we discussed in Example 3.5.1. Trivial variations of this optimal VDAG strategy may be obtained by other topological sorts. □

3.5.3 Classes of VDAGs with Optimal VDAG Strategies

We have seen that whenever the constructed expression graph with respect to a desired view ordering is acyclic, we can obtain an optimal VDAG strategy in a straightforward manner. The acyclicity of the expression graph depends not only on the VDAG but also on the desired view ordering being considered. (In fact, we can show that if the edges due to the view ordering dependencies are removed, the resulting expression graph is always acyclic.) The view ordering in turn depends on the current set of changes. In general, a given VDAG may have an acyclic expression graph with one desired view ordering (*i.e.*, based on a set of changes) and a cyclic expression graph with another desired view ordering (*i.e.*, based on another set of changes). However, there are specific classes of VDAGs which will always have acyclic expression graphs. The important thing about these classes of VDAGs is that for these VDAGs we can always find optimal VDAG strategies in a straightforward manner no matter what changes are being propagated. We identify two such classes of VDAGs below.

Definition 3.5.1 (**Tree VDAGs**) A *tree* VDAG is one in which no view is used in the definition of more than one other view. □

The class of tree VDAGs may appear very simple, but it encompasses a large number of VDAGs that occur naturally in many warehouse contexts. A simple example of a tree VDAG is shown in Figure 3.6. Based on the following lemma, one can easily find optimal VDAG strategies for tree VDAGs. The proof of the lemma is furnished in Appendix A.

Lemma 3.5.1 *For a tree VDAG, every view ordering results in an acyclic expression graph.* □

Definition 3.5.2 (**Uniform VDAGs**) A VDAG G is a *uniform* VDAG if every derived view at *Level* i is defined over views all of which are at *Level* $(i - 1)$. □

Uniform VDAGs have a well-defined notion of *Level* for each view. The TPC-D warehouse shown in Figure 3.4 has a uniform VDAG. In this uniform VDAG, all base views have *Level* 0 and all derived views have *Level* 1. The class of uniform VDAGs, although quite large, does not encompass the class of tree VDAGs. For instance, the tree VDAG of Figure 3.6 is not a uniform VDAG. At the same time, there are uniform VDAGs that are not tree VDAGs. For instance, the uniform VDAG for the TPC-D warehouse (Figure 3.4) is not a tree VDAG.

Based on the following lemma, we can easily generate optimal VDAG strategies for uniform VDAGs. The proof of the lemma is furnished in Appendix A.

Lemma 3.5.2 *For a uniform VDAG, every view ordering results in an acyclic expression graph.* □

3.5.4 *MinWork* Algorithm

Based on our observations above, we develop an algorithm called *MinWork* to generate VDAG strategies that minimize the total amount of work. In particular, *MinWork* relies on the approach of expression graph construction in order to find good VDAG strategies. The algorithm is formally presented in Algorithm 3.5.1 of Figure 3.8.

Algorithm 3.5.1 *MinWork*
Input: VDAG G with nodes $\mathcal{V}$ and edges $\mathcal{A}$
Output: 1-way VDAG strategy $\overrightarrow{\mathcal{E}}$
1. $\overrightarrow{\mathcal{E}} \leftarrow \langle\,\rangle$
2. For each $V_i \in \mathcal{V}$ estimate $|V_i'| - |V_i|$ based on the current set of changes
3. $\overrightarrow{\mathcal{V}} \leftarrow \mathcal{V}$ ordered by increasing $|V_i'| - |V_i|$
4. EG $\leftarrow$ *ConstructEG*$(G, \overrightarrow{\mathcal{V}})$
5. If EG is acyclic then
    6. $\overrightarrow{\mathcal{E}} \leftarrow$ topological sort of EG
7. Else
    8. $\overrightarrow{\mathcal{V}'} \leftarrow$ *ModifyOrdering*$(\overrightarrow{\mathcal{V}})$
    9. EG$' \leftarrow$ *ConstructEG*$(G, \overrightarrow{\mathcal{V}'})$
    10. $\overrightarrow{\mathcal{E}} \leftarrow$ topological sort of EG$'$
11. Return $\overrightarrow{\mathcal{E}}$

$\diamond$

Algorithm 3.5.2 *ModifyOrdering*
Input: VDAG G, view ordering $\overrightarrow{\mathcal{V}}$
Output: modified view ordering $\overrightarrow{\mathcal{V}'}$
1. $\overrightarrow{\mathcal{V}'} \leftarrow \langle\,\rangle$
2. For $l = 0$ to $MaxLevel(G)$
    3. $\overrightarrow{\mathcal{V}_l} \leftarrow$ subsequence of $\overrightarrow{\mathcal{V}}$ composed of all and only views with a *Level* value of l
    4. Append $\overrightarrow{\mathcal{V}_l}$ to $\overrightarrow{\mathcal{V}'}$
5. Return $\overrightarrow{\mathcal{V}'}$

$\diamond$

Figure 3.8: *MinWork* Algorithm

As shown in the figure, *MinWork* first computes a desired view ordering based on the current set of changes. Then it constructs the expression graph of the VDAG with respect to this desired view ordering. *ConstructEG* (Figure 3.9) includes one node for each 1-way expression of G. It then connects the nodes based on dependencies imposed by the correctness conditions, and the dependencies imposed by the given view ordering. If the constructed expression graph is acyclic, *MinWork* obtains the optimal VDAG strategy by a topological sort of the expression graph. Otherwise, it computes a modified view ordering (using *ModifyOrdering* shown in Algorithm 3.5.2, Figure 3.8) which is guaranteed to yield an acyclic expression graph of the VDAG . Then, it generates a VDAG strategy for the input VDAG that is consistent with this modified view ordering.

It is clear that given a VDAG that results in an acyclic expression graph with respect to the desired view ordering, *MinWork* produces an optimal VDAG strategy. This leads to the following result that follows from Theorem 3.5.3, Lemma 3.5.1 and Lemma 3.5.2.

Theorem 3.5.4 *Given a VDAG G, and a desired view ordering $\overrightarrow{\mathcal{V}}$, MinWork produces optimal VDAG strategies if $EG(G, \overrightarrow{\mathcal{V}})$ is acyclic. In particular, MinWork always produces*

Algorithm 3.5.3 *ConstructEG*
Input: VDAG $G = \langle \mathcal{V}, \mathcal{A} \rangle$, view ordering $\overrightarrow{\mathcal{V}}$
Output: EG of a 1-way VDAG strategy consistent with $\overrightarrow{\mathcal{V}}$
Initialize EG with no nodes and no edges.
1. For each node $V_i \in G(\mathcal{V})$, add $Inst(V_i)$ as an EG node.
2. For each edge $V_j \rightarrow V_i \in G(\mathcal{V})$, add $Comp(V_j, \{V_i\})$ as an EG node.
3. For nodes $Comp(V, \{V_i\}), Comp(V, \{V_j\})$ in EG
    4. If $V_i < V_j$ in $\overrightarrow{\mathcal{V}}$ Then
        5. Add $Comp(V, \{V_j\}) \rightarrow Comp(V, \{V_i\})$ as an EG edge labeled $\overrightarrow{\mathcal{V}}$.
6. For each node $V_i \in G(\mathcal{V})$, for each edge $V \rightarrow V_i \in G(\mathcal{A})$
    7. Add $Inst(V_i) \rightarrow Comp(V, \{V_i\})$ as an EG edge (for **C3**).
8. For each edge $Comp(V, \{V_j\}) \rightarrow Comp(V, \{V_i\})$ in EG
    9. Add $Comp(V, \{V_j\}) \rightarrow Inst(V_i)$ as an EG edge (for **C4**).
10. For each node $V_i \in G(\mathcal{V})$, for each edge $V \rightarrow V_i \in G(\mathcal{A})$
    11. Add $Inst(V) \rightarrow Comp(V, \{V_i\})$ as an EG edge (for **C5**).
12. For each edge $V_k \rightarrow V_j \in G(\mathcal{E})$, for each edge $V_j \rightarrow V_i \in G(\mathcal{E})$
    13. Add $Comp(V_k, \{V_j\}) \rightarrow Comp(V_j, \{V_i\})$ as an EG edge (for **C8**).
14. Return EG

$\diamond$

Figure 3.9: *ConstructEG* Algorithm

optimal VDAG strategies for tree VDAGs and uniform VDAGs. $\square$

When the given VDAG results in a cyclic expression graph with respect to the desired view ordering, *MinWork* produces a 1-way VDAG strategy that is consistent with a view ordering $\overrightarrow{\mathcal{V}'}$ that is produced by *ModifyOrdering* based on the desired view ordering. *ModifyOrdering* produces $\overrightarrow{\mathcal{V}'}$ by first ordering the views based on their *Level* values (*i.e.*, lower level views first). *ModifyOrdering* then orders the views with the same *Level* value based on the desired view ordering. The following theorem (proven in Appendix A) ensures that *MinWork* will always be able to generate a 1-way VDAG strategy no matter how complex the input VDAG is using the modified view ordering.

Theorem 3.5.5 *Given a VDAG G and a view ordering $\overrightarrow{\mathcal{V}}$, we can come up with a view ordering $\overrightarrow{\mathcal{V}'}$ = ModifyOrdering(G, $\overrightarrow{\mathcal{V}}$) such that EG(G, $\overrightarrow{\mathcal{V}'}$) is acyclic. That is, MinWork will always succeed in producing a VDAG strategy.* $\square$

The use of a modified view ordering when a desired view ordering yields cyclic expression graphs may lead *MinWork* to produce sub-optimal VDAG strategies. However, the modified view ordering reflects as much of the desired view ordering as possible. This results in *MinWork* producing near optimal plans, when it misses optimal plans.

Finally, we note that *MinWork* has a worst case time complexity of $O(n^3)$ where n is the number of views in the VDAG. The most complex part of the algorithm, taking $O(n^3)$ time, is building the expression graph using *ConstructEG*. All other parts take at most $O(n^2)$ time.

3.5.5 Practical Issues

We now outline how to resolve a number of practical and important issues regarding the implementation of *MinWork* on top of a commercial RDBMS. In particular we discuss the following issues: (1) how to implement *MinWork* using SQL stored procedures and a high level programming language like C++; (2) how to determine a desired view ordering. We provide this discussion to show that if the warehouse is built on top of a commercial RDBMS, *MinWork* can be implemented by a WHA easily without changing the internals of the RDBMS.

Implementing *MinWork*

The key observation is that given a VDAG, the set of 1-way expressions used by the *MinWork* VDAG strategy is known a priori. That is, for each edge $V_j \rightarrow V_i$ in the VDAG, a compute expression $Comp(V_j, \{V_i\})$ will be used, and for each node V_i in the VDAG, an install expression $Inst(V_i)$ will be used. Only the order of the expressions in the strategies depends on the changes being processed at the warehouse. Hence, based on the VDAG of the warehouse, a set of stored procedures is defined, one for each compute or install expression. This leads to efficient execution of the VDAG strategy because the stored procedures need not be parsed and go through all the optimization steps every time the warehouse needs to be updated.

Using the above technique, we define the following approach to warehouse update processing:

1. Given a set of view definitions, the corresponding VDAG is generated.

2. Given the VDAG generated in the previous step, the set of stored procedures for the compute and install expressions are defined.

3. Each time the warehouse needs to be updated, *MinWork* is invoked to produce a

VDAG strategy.

4. The resulting VDAG strategy is executed with the help of the stored procedures defined in the second step.

Computing a desired view ordering

Recall that *MinWork* needs to find a desired view ordering which is based on $|V'| - |V|$ values for each view V. Estimates of $|V|$ should be available from the metadata. To estimate $|V'|$, we can first estimate $|\delta V|$, and then compute $|V'|$ based on $|\delta V|$ and $|V|$. Estimates of $|\delta V|$ are obtained easily for base views since the changes are provided before the warehouse update starts. Estimates of $|\delta V|$ for derived views can be obtained using standard query result size estimation methods [Ull89b]. That is, assuming V is defined over views $\{V_1, \ldots, V_n\}$, and estimates of $\{|\delta V_1|, \ldots, |\delta V_n|, |V_1'|, \ldots, |V_n'|\}$ have been obtained (*i.e.*, we proceed bottom-up), $|\delta V|$ can be estimated using standard methods.

3.6 Optimal 1-way VDAG Strategies

We just showed that for VDAGs and view orderings that result in acyclic expression graphs, an optimal VDAG strategy can be obtained efficiently using *MinWork*. If the expression graph is cyclic, finding an optimal VDAG strategy is very hard, and the optimal VDAG strategy may not even be a 1-way VDAG strategy. However, since we showed that certain 1-way view strategies are optimal for single views, and certain 1-way VDAG strategies are optimal for VDAGs and view orderings with acyclic expression graphs, we focus on the problem of finding the best 1-way VDAG strategy.

In this section, we present a search algorithm called *Prune* that avoids examining much of the solution space but is guaranteed to produce the best 1-way VDAG strategy.

Even though *Prune* restricts its search to 1-way VDAG strategies, the set of 1-way VDAG strategies is still potentially very large. *Prune* pares down the search space by partitioning the set of 1-way VDAG strategies and considering only one representative VDAG strategy from each partition. Figure 3.10 illustrates how the technique reduces the search space. In the figure, each point represents a 1-way VDAG strategy, but only the marked points are considered by *Prune*.

Prune partitions the 1-way VDAG strategies based on which view ordering the 1-way VDAG strategies are *strongly consistent* with. A 1-way VDAG strategy $\overrightarrow{\mathcal{E}}$ is strongly

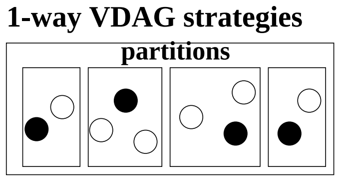

Figure 3.10: Intuition of *Prune*

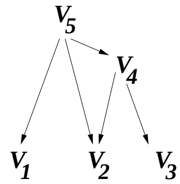

Figure 3.11: Problem VDAG

consistent with a view ordering $\overrightarrow{\mathcal{V}}$ if $Inst(V_i) < Inst(V_j)$ in $\overrightarrow{\mathcal{E}}$ implies that $V_i < V_j$ in $\overrightarrow{\mathcal{V}}$.

Partitioning 1-way VDAG strategies based on which view ordering the VDAG strategies are strongly consistent with is correct because each 1-way VDAG strategy is strongly consistent with exactly one view ordering, as stated in the next lemma.[6] Hence, all 1-way VDAG strategies that are strongly consistent with the same view ordering are placed in the same partition.

Lemma 3.6.1 *Every 1-way VDAG strategy is strongly consistent with some view ordering $\overrightarrow{\mathcal{V}}$. Furthermore, a 1-way VDAG strategy is strongly consistent with exactly one view ordering $\overrightarrow{\mathcal{V}}$.* □

Lemma 3.6.1 follows from the fact that any VDAG strategy $\overrightarrow{\mathcal{E}}$ must have exactly one *Inst* expression for each VDAG view (*i.e.*, by **C6** and **C7**). Hence, $\overrightarrow{\mathcal{E}}$ must be strongly consistent with the view ordering $\overrightarrow{\mathcal{V}}$ (and no other view ordering) that orders all of the VDAG views based on the order of appearance of the *Inst* expressions in $\overrightarrow{\mathcal{E}}$.

While a 1-way VDAG strategy $\overrightarrow{\mathcal{E}}$ is strongly consistent with exactly one view ordering $\overrightarrow{\mathcal{V}}$, there may be a number of 1-way VDAG strategies that are strongly consistent with $\overrightarrow{\mathcal{V}}$. Thus, there may be a number of 1-way VDAG strategies in each partition. The next theorem states that all the VDAG strategies in a partition incur the same amount of work.

Theorem 3.6.1 *Given a view ordering $\overrightarrow{\mathcal{V}}$, all the 1-way VDAG strategies that are strongly consistent with $\overrightarrow{\mathcal{V}}$ incur the same amount of work.* □

The theorem holds because 1-way VDAG strategies use the same set of expressions. Furthermore, it can be shown that if two VDAG strategies are strongly consistent with the same view ordering, each *Comp* expression runs on the same "database state" in both VDAG

[6] On the other hand, 1-way VDAG strategies cannot be partitioned based on which view ordering the 1-way VDAG strategies are consistent with because a 1-way VDAG strategy may be consistent with more than one view ordering.

strategies. Since we know that the work incurred by any *Inst* expression is the same, the theorem follows. For details of the proof, see Appendix A.

In summary, based on the above theorems, *Prune* can search over 1-way VDAG strategies by considering the set of view orderings and by examining one 1-way VDAG strategy that is strongly consistent with each ordering. However, *Prune* needs to handle the complication that given a view ordering $\overrightarrow{\mathcal{V}}$, there may not exist a (correct) 1-way VDAG strategy that is strongly consistent with $\overrightarrow{\mathcal{V}}$. For instance, for the VDAG shown in Figure 3.11, there is no 1-way VDAG strategy that is strongly consistent with $\overrightarrow{\mathcal{V}} = \langle V_4, V_1, V_2, V_3, V_5 \rangle$. This is because $Comp(V_4, \{V_3\})$ must be after $Inst(V_2)$ for **C4** to hold, and for the VDAG strategy to be strongly consistent with $\overrightarrow{\mathcal{V}}$. However, $Comp(V_4, \{V_3\})$ must be before $Inst(V_4)$ and therefore before $Inst(V_2)$, for **C8** to hold.

To handle this complication, *Prune* (Figure 3.13) constructs a *strong expression graph* (SEG) that is similar to the expression graph that *MinWork* constructs. If a cyclic SEG is constructed, then there is no 1-way VDAG strategy that is strongly consistent with the given view ordering. Otherwise, *Prune* produces a candidate 1-way VDAG strategy by topologically sorting the expressions in the SEG. *Prune* returns the 1-way VDAG strategy that incurs the least amount of work.

Algorithm 3.6.1 *ConstructSEG*
Input: VDAG $G = \langle \mathcal{V}, \mathcal{A} \rangle$, view ordering $\overrightarrow{\mathcal{V}}$
Output: SEG of a 1-way VDAG strategy strongly consistent with $\overrightarrow{\mathcal{V}}$
  Initialize SEG with no nodes and no edges
  1–2. Lines 1–2 of *ConstructEG* (Figure 3.9)
  3. For nodes $Inst(V_j), Inst(V_i)$ in SEG
    4. If $V_i < V_j$ in $\overrightarrow{\mathcal{V}}$ Then
      5. Add $Inst(V_j) \rightarrow Inst(V_i)$ as an SEG edge
  6–13. Lines 6–13 of *ConstructEG*
  14. Return SEG
◇

Figure 3.12: *ConstructSEG* Algorithm

To construct the SEG, *Prune* uses *ConstructSEG* which is almost identical to *ConstructEG* (see Figure 3.9). The only difference is that *ConstructSEG* adds an edge $Inst(V_j) \rightarrow Inst(V_i)$ if V_i is before V_j in the input view ordering $\overrightarrow{\mathcal{V}}$. Unlike *ConstructEG*, *ConstructSEG* adds this edge even when there is no view V that is defined on both V_i and V_j. This edge guarantees that $Inst(V_i)$ is before $Inst(V_j)$ in the topological sort (if possible) of the constructed SEG. This in turn guarantees that the 1-way VDAG strategy produced is strongly consistent with $\overrightarrow{\mathcal{V}}$.

Algorithm 3.6.2 *Prune*
Input: $G = \langle \mathcal{V}, \mathcal{A} \rangle$
Output: an optimal VDAG strategy $\overrightarrow{\mathcal{E}}$

1. $\overrightarrow{\mathcal{E}_{best}} \leftarrow \langle\ \rangle$ // incorrect VDAG strategy with infinite amount of work
2. For each view ordering $\overrightarrow{\mathcal{V}}$
   3. SEG $\leftarrow$ *ConstructSEG*$(G, \overrightarrow{\mathcal{V}})$
   4. If SEG is acyclic Then
      5. $\overrightarrow{\mathcal{E}} \leftarrow$ topological sort of the expressions in SEG
      6. If $Work(\overrightarrow{\mathcal{E}}) < Work(\overrightarrow{\mathcal{E}_{best}})$ Then
         7. $\overrightarrow{\mathcal{E}_{best}} \leftarrow \overrightarrow{\mathcal{E}}$
8. Return $\overrightarrow{\mathcal{E}_{best}}$

$\diamond$

Figure 3.13: *Prune* Algorithm

Since *Prune* examines each view ordering, and examines a representative VDAG strategy consistent with each view ordering, it is easy to prove that *Prune* finds the best 1-way VDAG strategy (see Appendix A).

Theorem 3.6.2 *Prune is guaranteed to produce the best 1-way DAG strategy for a given VDAG.* □

We note that *Prune* examines $n!$ view orderings, where n is the number of VDAG views. Also, *ConstructSEG*, like *ConstructEG*, runs in $O(n^3)$ time in building an SEG. Since an SEG needs to be constructed for each view ordering, *Prune* runs in $O(n! \cdot n^3)$ time.

Compared with the space of all 1-way VDAG strategies for a given VDAG, *Prune* searches over a very small set of 1-way VDAG strategies and thus is relatively quite efficient. However, it can be improved further while still guaranteeing that an optimal 1-way VDAG strategy is produced. For instance, it is not necessary to examine all possible view orderings. More specifically, if there are no views defined on V, δV can be installed at any point in the VDAG strategy after δV has been computed. If we remove all such views from the view ordering, we only need to consider $O(m!)$ view orderings, where m is the number of VDAG views with a view defined on them. For instance, for the TPC-D VDAG shown in Figure 3.4, there are $n = 9$ views, but there are only $m = 6$ views with some view defined over them. Hence, *Prune* can be optimized to examine only $6! = 720$ strategies, instead of $9! = 362880$ strategies.

Just like *MinWork*, *Prune* should be implemented by creating SQL stored procedures for each expression that will be used by the *Prune* VDAG strategy. Observe also that *Prune* (Figure 3.13) compares various VDAG strategies in terms of their total work under the linear work metric. In order to estimate the total work of a VDAG strategy, we need estimates for $|V|$, $|\delta V|$ and $|V'|$ for each view V. We already showed in Section 3.5.5 how these values can be obtained using standard result size estimation methods.

3.7 Experiments and Discussion

We have developed algorithms that minimize the work incurred in view or VDAG strategies. However, minimizing the work incurred may not translate to the minimization of the update window. When a strategy is executed, many factors that affect the update window (e.g., buffering of the intermediate results and the particular join and aggregation methods used in computing these intermediate results) are too complex to be modeled by our simple work metric.

In order to understand how well the strategies generated by our algorithms perform in practice, we conducted a series of experiments. In particular, we tested various strategies using Microsoft SQL Server 6.5 running on a Dell XPS D300 with a Pentium II 300 MHz processor and 64 MB of RAM. In our experiments, we measured the actual time it took to execute the strategies. The results of our experiments show that the strategies generated by our algorithms do indeed yield short update windows.

In all of the experiments, we used the TPC-D warehouse shown in Figure 3.4. The base views *CUSTOMER* (denoted C for conciseness), *ORDER* (O), *LINEITEM* (L), *SUPPLIER* (S), *NATION* (N) and *REGION* (R) are copies of TPC-D relations populated with synthetic data obtained from [Com]. The derived views $Q3$, $Q5$ and $Q10$ were defined using the TPC-D "Shipping Priority" query, "Local Supplier" query, and "Returned Item Reporting" query respectively.

Unless otherwise specified, the remote information sources were changed so that base views C, O, L, S, and N decreased in size by 10%. Base view R, the smallest of the six, was left unchanged. According to the sizes of the base views, the desired view ordering is $\langle\, L, O, C, S, N, R \,\rangle$ (*i.e.*, L is the largest base view). (Note that the three derived views can be ignored in the view ordering since there are no views defined on them.) In one of the experiments, we investigated other possible changes to the remote information sources.

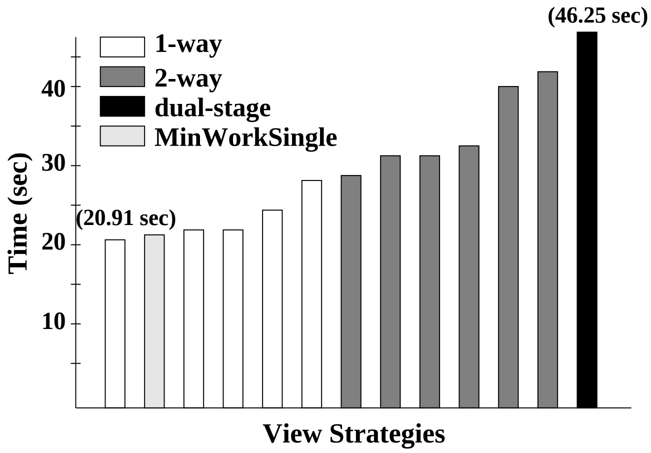

Figure 3.14: $Q3$ View Strategies

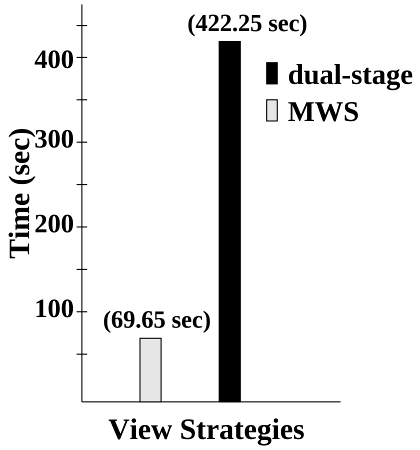

Figure 3.15: $Q5$ View Strategies

Experiment 1

In the first experiment, we examined the various view strategies for $Q3$. Since $Q3$ is only defined over 3 views, there were only 13 view strategies to compare, one from each partition. Figure 3.14 shows the result of the experiment. Each bar depicts a view strategy, and the height of the bar gives the amount of time it took to perform the view strategy. The graph shows numerous results.

First, the graph shows that 1-way view strategies update $Q3$ in the least amount of time. That is, the dual-stage view strategy is worse than all of the 1-way view strategies. Also, any "2-way" view strategy that uses an expression $Comp(Q3, \mathcal{V})$, where $|\mathcal{V}| = 2$, is worse than all of the 1-way view strategies.

Second, the graph shows that the *MinWorkSingle* view strategy, which propagates the changes of L, then of O, and then of C, does not update $Q3$ in the least amount of time. The view strategy that performs the best in this case propagates the changes of L, then of C and then of O. The update window of the *MinWorkSingle* view strategy is however very close to the optimal. Recall that in Section 3.4, we proved that *MinWorkSingle* produces an optimal view strategy under the linear work metric. In the experiment, we used a real system whose behavior naturally deviates from the strictly linear work metric and hence *MinWorkSingle* ends up with a view strategy that is slightly away from the optimum. Notice that the margin of error is small, indicating that using the linear work metric, one can generate near-optimal update windows.

Finally, the graph shows that various view strategies have significantly different update windows. For instance, the update window of the dual-stage view strategy is about 2.3 times longer than that of the optimal view strategy.

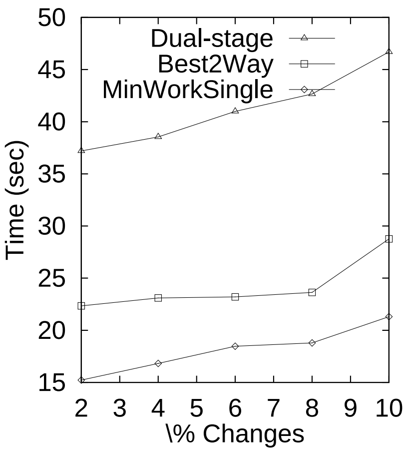

Figure 3.16: $Q3$ View Strategies Under Different Changes

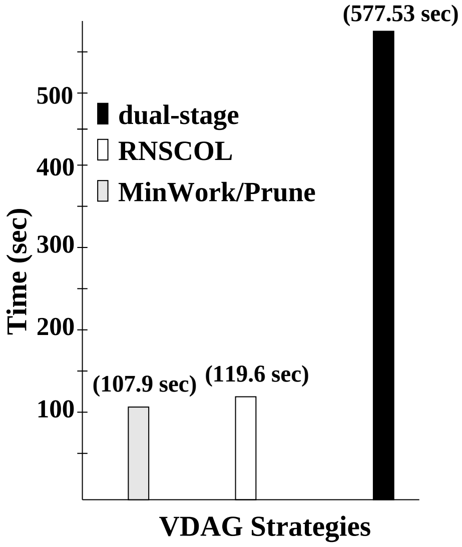

Figure 3.17: VDAG Strategies

Experiment 2

In the next experiment, we focused on the derived view $Q5$ which is defined over the 6 base views. Since $Q5$ is much more complex than $Q3$, it was too time consuming to examine all of the view strategies of $Q5$. Instead, we examined only the *MinWorkSingle* view strategy and the dual-stage view strategy. Recall that the dual-stage view strategy is the one with a compute stage and an install stage, as proposed in [CGL+96]. The results of the experiment are shown in Figure 3.15. Notice that the update window of the dual-stage view strategy is over 6 times longer than that of the *MinWorkSingle* view strategy. On the other hand, the update window of the dual-stage view strategy for $Q3$ was "only" 2.2 times longer than that of the *MinWorkSingle* view strategy (see Figure 3.14). This shows that using the *MinWorkSingle* view strategy instead of the dual-stage view strategy to update complex views is likely to be very beneficial.

Experiment 3

In this experiment, we again focus on $Q3$. Each of C, O, and L is decreased in size by a percentage p of its initial size, for various values of p. When comparing view strategies, we only considered the *MinWorkSingle* view strategy, the best 2-way view strategy in Figure 3.14, and the dual-stage view strategy. Figure 3.16 shows the results of the experiment. The results indicate that the *MinWorkSingle* view strategy improves on the other view strategies over a wide range of amounts of changes to the underlying views.

Experiment 4

So far, we have considered updating a single view. In this experiment, we study the quality of *MinWork* VDAG strategies. Note that, since the TPC-D VDAG is uniform, *MinWork* is guaranteed to pick an optimal VDAG strategy under the linear work metric.

We check how good the *MinWork* VDAG strategy is by comparing it with two others: a "dual-stage" VDAG strategy that only uses dual-stage view strategies, and a 1-way VDAG strategy that propagates the changes in an order opposite that of the *MinWork* VDAG strategy. *MinWork* uses the view ordering $\langle\ L, O, C, S, N, R\ \rangle$, and so the third VDAG strategy in our experiment uses the order $\langle\ R, N, S, C, O, L\ \rangle$. We call this strategy RNSCOL. The results of the experiment are shown in Figure 3.17. As expected, the *MinWork* strategy performed the best. In particular, it is 5 to 6 times better than the dual-stage VDAG strategy, and is about 11% better than the RNSCOL VDAG strategy.

Discussion

Although the dual-stage VDAG strategy has a very long update window compared to the two 1-way VDAG strategies, it does have the advantage of being able to perform all of the *Inst* expressions in the second stage, which minimizes the time in which locking operations are necessary. However, even though the sequence of *Comp* expressions in the first stage do not need to lock the database, they still compete with OLAP queries for resources. On the other hand, the sequence of expressions used by the 1-way VDAG strategies are more efficient and take less resources away from OLAP queries. Moreover, it is often acceptable for OLAP queries to run at lower isolation levels, which allows the *Inst* expressions to run without locking. This diminishes any advantage the dual-stage VDAG strategy has over the 1-way VDAG strategies.

We also note that the results of Experiment 4 suggests that the linear work metric is a good measure of the work incurred by a VDAG strategy. For instance, a variant of the linear work metric may just sum the sizes of the operands. To illustrate, the work estimate for $Comp(V_4, \{V_2, V_3\})$ (Figure 3.11) under this variant is $c \cdot (|\delta V_2| + |V_2| + |\delta V_3| + |V_3|)$, since the number of terms in which an operand V or δV appears in is not modeled. Under this work metric, the dual-stage VDAG strategy would be best contrary to the results of Experiment 4.

3.8 Related Work

There has been a significant amount of work in minimizing warehouse maintenance time. This is because there are many techniques, each solving a different sub-problem.

One of the sub-problems is the efficient maintenance of base views that are defined over remote sources. Hence, there have been previous work ([QGMW96], [Huy97], [GJM96]) that determines when a base view can be maintained without accessing remote sources. If these remote sources do need to be accessed, [AASY97] gives algorithms for base view maintenance. In this chapter, we concentrate on derived view maintenance. Even though maintaining derived views only requires accessing data local to the warehouse, it can be a very expensive process. Furthermore, unlike base view maintenance, derived view maintenance competes with OLAP queries for resources, and thus is one of the main problems that today's warehouses face.

Another important sub-problem is choosing the views to materialize in the warehouse so that some measure like query time, maintenance time, or a combination of the two, is minimized while satisfying a given storage or maintenance time constraint. Warehouse design has been discussed in [Gup97],[HRU96],[BPT97], [YKL97], [TS97]. The warehouse design algorithms are complementary to the algorithms we present. Most of the warehouse design algorithms, such as the greedy algorithm of [Gup97], do not specify how views are actually updated. On the other hand, we give algorithms that update views in a very specific manner. Hence, our algorithms can be combined with design algorithms in many ways. One way is that a design algorithm picks the set of views $\mathcal{V}$ to materialize. The algorithms we present are then used to update the views in $\mathcal{V}$ once they are materialized. Alternatively, a design algorithm assumes that the algorithms we present are used to update the warehouse, affecting which set of views $\mathcal{V}$ is materialized. As before, our algorithms are then used to update the warehouse.

Another sub-problem that needs to be tackled is to develop a good storage representation for views so that incorporating bulk changes into the views can be done efficiently. Recently, [KR98] proposed a variant of R-trees called *Cubetrees* as the storage representation of the views. Reference [JNSS97] also discussed how to incorporate changes quickly into a clustered storage organization using sorting and hashing techniques. The storage representation presented in these papers can be used in conjunction with the algorithms we present.

Another sub-problem that needs to be answered is deciding when to update the warehouse. Reference [CKL+97] presents a framework for supporting different maintenance policies based on when changes are propagated to the views. On the other hand, the algorithms we present are used when changes are actually propagated. Hence, the algorithms we present are complementary.

The only work that we know of that is concerned with the actual algorithm for propagating changes is [MQM97]. More specifically, [MQM97] proposed to represent the changes of summary tables as a *summary delta* (*i.e.*, result of applying the grouping operator and aggregation functions over the changes). Since a summary delta can be incorporated into a summary table very efficiently, the main problem is computing the summary delta. The algorithms we present here can be used to compute the summary deltas more efficiently

Finally, the algorithms we present are different from most of the previous algorithms since our algorithms are concerned with a DAG of views, instead of just one view. In this context, a careful treatment is required to maintain a DAG of views correctly and efficiently.

3.9 Chapter Summary

We have solved the "total-work minimization" (TWM) problem that warehouse administrators face today. To solve TWM, we presented *MinWorkSingle* that identifies optimal view strategies for updating single views. We then presented *MinWork*, an efficient heuristic algorithm that finds an optimal solution for a large class of VDAGs. To find an optimal 1-way VDAG strategy for any VDAG, we presented *Prune*, which is a search technique that avoids considering a large part of the solution space. Both *MinWork* and *Prune* significantly extend the 1-way view strategy ([GMS93]) to the more practical setting of a VDAG of views. Experiments on a TPC-D VDAG showed that the strategies produced by *MinWorkSingle* and *MinWork* are very efficient under commercial RDBMS work metrics. The experiments also showed that the *MinWorkSingle* and *MinWork* result in update windows that are significantly shorter than the update windows of conventional update strategies. We also discussed how the algorithms can easily be implemented without modifying the internals of a commercial RDBMS.

Chapter 4

Optimizing the View Hierarchy

4.1 Introduction

In Chapter 3, we discussed algorithms that propagate and install the changes efficiently up a given VDAG. In this chapter, we further improve the efficiency of warehouse maintenance by manipulating the structure of the VDAG itself. That is, at the time the warehouse and the VDAG are designed, the algorithms we propose in this chapter can be used to "improve" the VDAG.

We improve the VDAG by adding additional views and/or indices. This approach may seem counter-intuitive at first, since the additional views and indices also need to be maintained just like the views in the original VDAG. However, this approach is analogous to building indices in traditional RDBMS's. For example, having an index on the key of a relation can vastly decrease the total time spent locating particular tuples to be updated or deleted, offsetting the fact that the index must be maintained as well. In this chapter we will approximate maintenance time as the number of IO's required and then endeavor to minimize the number of IO's performed. We start with the number of IO's required for maintaining the materialized views in the VDAG (using a 1-way view strategy as discussed in Chapter 3). We then add a set of additional views and indices that themselves must be maintained, but whose *benefit* (reduction in IO's) outweighs the *cost* (increase of IO's) of maintaining them.

To illustrate why additional views and indices may be beneficial, let us focus on a simple VDAG shown in Figure 4.1. The example VDAG has three base views (R, S, and T), and a single derived view $V = R \bowtie S \bowtie T$. The algorithms we propose in this chapter work on

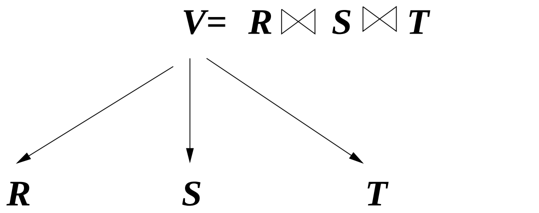

Figure 4.1: Warehouse with Primary View.

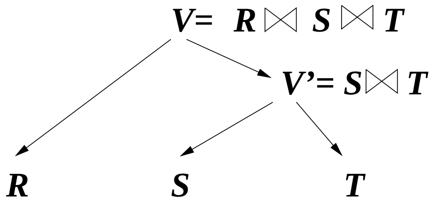

Figure 4.2: Warehouse with Supporting View.

each derived view V and the views they are defined on. The derived view V in question is called the *primary view*, and the views on which V is directly defined on are called *lower views*. Note that a primary view must be a derived view, whereas a lower view can be a derived or a base view.

Suppose that in addition to materializing the primary view V and its lower views, another view, $V' = S \bowtie T$, is also materialized. This situation is shown in Figure 4.2. By materializing view V', the total cost of maintaining both V and V' can be less than the cost of maintaining V alone. For example, suppose that there are insertions to R but no changes whatsoever to S and T. To propagate the insertions to R onto V, we must evaluate the maintenance expression that calculates the tuples to insert into V due to insertions into R, which is $\triangle R \bowtie S \bowtie T$. With V' materialized, it is almost certain that this expression can be evaluated more efficiently as $\triangle R \bowtie V'$, joining the insertions to R with V', instead of with S and T individually. Even if there are changes to S and T, the benefit of materializing V' may still outweigh the extra cost involved in maintaining it. Since the view V' is materialized to support in the maintenance of the primary view V, we call the view V' a *supporting view*.

In addition to materializing supporting views, it may be beneficial also to materialize supporting indices. Indices may be built on the lower views, primary views, and on the supporting views. The general problem, then, is to choose a set of supporting views and a set of indices to materialize such that the total maintenance cost for the warehouse is minimized. We call this problem the *View-Index Selection* (VIS) problem and it is the focus of this chapter.

Below we list the primary contributions of this chapter.

- We propose and implement an optimal algorithm based on A* that prunes as much as 99% of the possible supporting view and index sets to solve the VIS problem.
- Through both cost/benefit analysis and experimentation, we develop a number of

rules of thumb that can help a warehouse administrator (WHA) find a reasonable set of supporting views to materialize in order to reduce the total maintenance cost.

- We develop efficient heuristic algorithms that choose view and index sets that are close to optimal. We also show that our heuristic algorithms are much better than algorithms proposed in previous work.

- We compare the benefit of materializing supporting views as opposed to indices, and discuss which should be chosen when the total storage space at the warehouse is constrained.

- We perform experiments to determine how sensitive the choice of supporting view and index sets are to the input parameters of the optimizer.

The rest of the chapter proceeds as follows. Section 4.2 describes the VIS problem in detail. Section 4.3 presents the scope of our results and our approach to view maintenance. We describe our A*-based algorithm in Section 4.4. Section 4.5 develops rules of thumb for choosing a set of supporting views to materialize. We justify our rules both by a cost model analysis as well as by extensive experimentation using our A*-based algorithm. In Section 4.6, we develop heuristic algorithms that choose sets of supporting views and indices that perform close to the optimal set. We also show that our heuristic algorithms significantly improve on previous heuristic algorithms. In Section 4.7, we report on additional experiments such as comparing the relative importance of building indices versus materializing supporting views when space is constrained. Finally, in Section 4.8, we discuss related work.

4.2 General Problem

Having introduced the VIS problem, in this section we describe it fully and present an exhaustive search algorithm to solve it. We also show the worst case complexity of the VIS problem. Lastly, we present an example schema to illustrate the concepts introduced.

4.2.1 The VIS Problem

The VIS problem focuses on a single derived view, called the primary view, and the views that the primary view is defined on, called lower views. In the VIS problem, we are given a primary view, denoted V_P, and the lower views that V_P is defined on, denoted $\mathcal{B}$. The primary view V_P and the views in $\mathcal{B}$ are assumed to be materialized. We are also given a set of potential supporting views $\mathcal{V}$, called *candidate supporting views* (or *candidate views* in short), that can be materialized in addition to view V_P and the views in $\mathcal{B}$. Henceforth, when we refer to a view, it can be V_P, a view in $\mathcal{B}$ or a view in $\mathcal{V}$. We are also given a set of potential attributes on which indices can be constructed. We call this set of attributes the set of *candidate supporting indices* $\mathcal{I}$ (or *candidate indices* in short). The set $\mathcal{I}$ includes the attributes of view V_P, the attributes of the views in $\mathcal{B}$ and the attributes of the views in $\mathcal{V}$.

The VIS problem we address in this chapter is stated as follows.

Definition 4.2.1 (VIS Problem) Given a primary view V_P, the set of lower views $\mathcal{B}$ of V_P, a set of candidate views $\mathcal{V}$, and a set of candidate indices $\mathcal{I}$, select $\mathcal{V}' \subseteq \mathcal{V}$ and select $\mathcal{I}' \subseteq \mathcal{I}$ to materialize such that such that the maintenance cost of V_P, $\mathcal{B}$, $\mathcal{V}'$ and $\mathcal{I}$ is minimized. □

The maintenance cost referred to in the definition is the cost of computing and installing the changes to V_P and each of the views in $\mathcal{B}$ and $\mathcal{V}'$, plus the cost of updating the indices in $\mathcal{I}$.

The cost of maintaining one view differs depending upon what other views are available. (Recall that a view may refer to V_P, views in $\mathcal{B}$, and/or views in $\mathcal{V}$.) It is therefore incorrect to calculate the cost of maintaining each view in isolation. Moreover, in order to optimize the total cost it is necessary to consider view selection and index selection together. For example, if view selection is performed separately from index selection, it is not hard to concoct cases wherein a supporting view V is considered to be too expensive to maintain without indices. However, with indices V becomes part of the optimal solution since it may become feasible to maintain V once the proper indices are built.

Importance of the VIS Problem

Although the warehouse data is a VDAG of views in general, solving the VIS problem (for a single primary view) is essential in improving the maintenance of the warehouse. For instance, it may be the case that there are specific derived views that are problematic, *i.e.*, these views are very expensive to maintain. In this case, it makes sense to solve the VIS problem for each of these problematic views. Furthermore, the algorithms for solving the

Algorithm 4.2.1 *VIS-Exhaustive*
Input: V_P, $\mathcal{B}$, $\mathcal{V}$
Output: $\mathcal{V}'$, $\mathcal{I}'$
1. For each possible subset $\mathcal{V}'$ of the candidate views $\mathcal{V}$
   2. Compute $\mathcal{I}$ based on V_P, $\mathcal{B}$ and $\mathcal{V}'$
   3. For each possible subset of indices $\mathcal{I}'$ of the candidate indices $\mathcal{I}$
      4. Compute the maintenance cost of V_P, $\mathcal{B}$, $\mathcal{V}'$ and $\mathcal{I}'$
         keep track of the supporting views $\mathcal{V}'$ and indices $\mathcal{I}'$ that obtain the minimum cost
5. Return $\mathcal{V}'$ and $\mathcal{I}'$

$\diamond$

Figure 4.3: *VIS-Exhaustive* Algorithm

VIS problem can be used as the building blocks of an algorithm that takes as input a VDAG, and "redesigns" the VDAG so that it is more efficient to update. One possible strategy is to solve the VIS problem for each derived view and then combine the supporting views and indices determined for each derived view. Hence, it is important to develop solutions to the VIS problem.

An Exhaustive Algorithm

One possible approach to finding the optimal solution to the VIS problem, proposed in Ross et al. [RSS96] (although their work does not consider indices), is to exhaustively search the solution space. Although exhaustive search is impractical for large problems, it illustrates the complexity of the problem and provides a basis of comparison for other solutions. The exhaustive algorithm is shown in Figure 4.3.

Choosing the views

In Line 1 of *VIS-Exhaustive*, we consider all possible subsets $\mathcal{V}'$ of the candidate views $\mathcal{V}$. As proposed in [RSS96], the candidate views are all the distinct nodes that appear in some query plan for the definition of the primary view V_P. Note that the primary view and the views in $\mathcal{B}$ are not not included in the set of candidate views as they are assumed to be materialized. For example, given a primary view $V_P = R \bowtie S \bowtie T$, $\mathcal{V} = \{RS, RT, ST\}$. In general, for a view joining n relations there are roughly $O(2^n)$ different nodes that appear in some query plan for the view, one joining each possible subset of the lower views. Thus, to consider all possible subsets of $\mathcal{V}$, we need to evaluate roughly $O(2^{2^n})$ different $\mathcal{V}'$ sets.

Choosing the indices

In Line 2 of *VIS-Exhaustive*, the set of candidate indices $\mathcal{I}$ is determined. At one extreme, all of the attributes of V_P, all of the attributes of the views in $\mathcal{B}$, and all of the attributes of the views in $\mathcal{V}'$ can be in $\mathcal{I}$. However, rather than considering all of the attributes of all the views to be candidate indices, we restrict candidate indices to the following types of attributes (as proposed in [FST88]):

- attributes that are referred to in a selection or a join condition in V_P's definition.[1]
- key attributes for lower views where changes to the lower view include deletions or updates. When a supporting view is materialized, attributes of the supporting view corresponding to key attributes of the contributing lower views also qualify.
- attributes used for grouping (using SQL `GROUPBY`) or ordering (using SQL `ORDERBY`) in V_P's definition.

Additional attributes can be candidates depending on the query optimizer being used. The reader is referred to [FST88] for more detail.

The cardinality of $\mathcal{I}$ considered in Lines 2–3 of *VIS-Exhaustive* is roughly proportional to the number of views in $\mathcal{V}'$, plus the number of views in $\mathcal{B}$, plus one (for V_P). Since there can be as many as $O(2^n)$ views, and each possible subset of candidate indices is considered, the number of subsets of candidate indices examined is $O(2^{2^n})$. (See Section 4.2.2 for an explanation of why standard approaches for index selection are not appropriate.)

Computing the total update cost

Once $\mathcal{V}'$ and $\mathcal{I}'$ are chosen, determining the maintenance cost (*i.e.*, Line 4 of *VIS-Exhaustive*) of all of the views and indices is a difficult problem in and of itself. Obtaining the maintenance cost is a query optimization problem since it involves finding the most efficient query plan for each of the view-maintenance expressions. The VIS problem for a single primary view joining n lower views thus contains roughly $O(2^{2^n})$ query optimization problems in the general case.

[1] In addition, the system must be able to use an index to process the predicate. This usually implies that the predicate is a simple comparison (except for $\neq$) or range operator and that the other operand is a constant or a column from a different table.

Query optimization is complicated by two issues: the presence of (materialized) views and the opportunity to optimize multiple maintenance expressions together. The presence of views requires the optimizer to determine if it can use some of the views in the query plan evaluating a maintenance expression. For example, given a view $RST = R \bowtie S \bowtie T$, insertions to R are propagated onto RST by the maintenance expression $\triangle R \bowtie S \bowtie T$. Suppose the view $ST = S \bowtie T$ is also materialized. The query optimization algorithm must consider the possibility of evaluating $\triangle R \bowtie S \bowtie T$ as $\triangle R \bowtie ST$ in finding the best query plan. This problem is known as "answering queries using views" [LMSS95].

Multiple maintenance expressions must be optimized due to different types of changes being propagated from different lower views. There is an opportunity to optimize the maintenance expressions for all changes and views as a group because of possible common subexpressions [RSS96]. This problem is known as the "multiple-query optimization" problem [Sel88].

4.2.2 Example

Consider the following lower views and primary view definition.

Lower views:

R(R0,R1), S(S0,S1), T(T0,T1)

Primary view definition:

```
CREATE VIEW V_P(R0,R1,SO,S1,T0,T1) AS
SELECT *
FROM R, S, T
WHERE R.R1 = S.S1 AND S.S0 = T.T0 AND T.T1 <= 10
```

Figure 4.4 shows an *expression DAG* [RSS96] that includes all the nodes that could appear in a query plan for V_P, assuming the selection on $T.T1$ is pushed down. The view T' is the result of applying the selection condition to T. Under each view is the set of operations (join or select) that could be used to derive the view. For example, the view RST could be derived as the result of $R \bowtie S$ joined with T', or the result of $R \bowtie S$ joined with the result of $S \bowtie T'$, and so on. Each of the intermediate results could be materialized as a supporting view. Following the definition in Section 4.2.1, the set of candidate views, $\mathcal{V}$, is $\{RS, ST', RT', T'\}$. Assuming V_P is materialized at a data warehouse (as well as the lower views), any possible subset of $\mathcal{V}$ might also be materialized as supporting views at

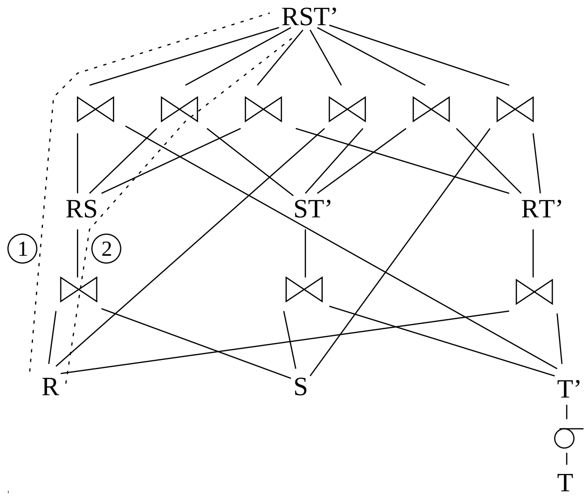

Figure 4.4: Example Schema.

the warehouse in order to minimize the total maintenance cost. In addition, indices on V_P, the lower views, and the supporting views need to be considered.

It is useful to think of the expression DAG in Figure 4.4 when considering the different *update paths* [RSS96] changes to lower views can take as they are propagated to the view. An update path corresponds to a specific query plan for evaluating a view maintenance expression. For example, the maintenance expression for propagating insertions to R onto V_P is to insert the result of $\triangle R \bowtie S \bowtie T'$ into V_P. The DAG in Figure 4.4 depicts two update paths for the expression $\triangle R \bowtie S \bowtie T'$: (1) $(\triangle R \bowtie S) \bowtie T'$, (2) $(\triangle R \bowtie S) \bowtie (S \bowtie T')$. One can easily check that there are five other update paths for the expression. Notice that the choice of update path can affect which indices are beneficial to materialize. If update path (1) is chosen, an index may be built on the join attribute of T' to help compute the maintenance expression $\triangle R \bowtie S \bowtie T'$. If however path (2) is chosen and view ST' is materialized, an index may be built on the join attribute of ST'. The use of different indices depending upon which update paths are chosen implies that it is not possible to choose an optimal set of views and indices simply by choosing the best views followed by the best indices on those views. Instead, views and indices must be chosen together to obtain an optimal set. This issue is discussed further in Section 4.8.

Changes to lower views need to be propagated both to the primary view as well as to the supporting views that have been materialized. When propagating changes to several lower views onto several materialized views there are opportunities for multiple-query optimization. Results of maintenance expressions for one view can be reused when evaluating maintenance expressions for another view, and common subexpressions can be detected between several maintenance expressions. As an example of reusing maintenance expressions, suppose view $RS = R \bowtie S$ is materialized. The result of propagating insertions to R onto RS, $\triangle R \bowtie S$, can be reused when propagating insertions to R onto V_P, $\triangle R \bowtie S \bowtie T'$, so that only the join with T' need be performed.

4.3 Assumptions

The algorithms we develop in this chapter are quite general and can be used to solve the general VIS problem discussed in the previous section. However, the algorithm developed in Section 4.4, and the rules of thumb proposed in Section 4.5 require a specific database model and change propagation model so that the maintenance cost for a set of views and indices can be determined. We now discuss the database model and the change propagation model that we assume. Our assumptions are similar to those made previously in the literature.

4.3.1 Database Model

The following are the assumptions we make about views and indices.

- Relations and views are stored as (unsorted) heaps.
- All indices are stored as B+-trees and are built on single attributes only.

Primary views are defined using select, project and join operations. The joins are foreign key to primary key joins. We assume the keys of the lower views are preserved in the primary view. Although this view definition seems restrictive, many warehouse views are defined this way following a "star" or a "snowflake" schema. The schema joins fact tables with dimension tables, and the join conditions are typically equijoins between foreign keys (of fact tables) to primary keys (of dimension tables). Furthermore, preserving the keys of the lower views can improve the installation of deletions as we will show in Chapter 7. More specifically, preserving keys allows the use of a single SQL `DELETE` statement, as opposed to using cursors for installing deletions.

We assume that the primary view(s) and replicas of the lower views are materialized in the warehouse. When considering what additional data structures to materialize, we restrict ourselves to data structures that are themselves easily maintainable using SQL data manipulation statements. To this end we consider materializing supporting views that are *subviews* of the primary view. That is, they join a subset of the lower views in the primary view. We also consider building indices on attributes in the lower views, primary view, and supporting views that are involved in selection or join conditions.

In choosing which structures to materialize, we implemented a VIS optimizer that considers the two most common physical join operators: nested-block joins and index joins. In addition, the VIS optimizer always "pushes down" projections and *local selection conditions* (involving attributes of a single lower view) as far as possible.

4.3.2 Change Propagation Model

As in most of the chapters of this thesis, we consider three types of deltas: insertions, deletions, and updates. We distinguish between two types of updates: Updates that alter the values of key attributes (if any) or attributes involved in selection or join conditions are called *exposed updates*; all other updates are called *protected updates*. Exposed updates can result in tuples being deleted from or inserted into the view. For this reason, we propagate exposed updates as deletions followed by insertions. Henceforth, all references to "updates" should be interpreted to mean "protected updates." Protected updates could also be propagated as deletions followed by insertions, but they can be applied directly to the view since they only change attribute values of tuples in the view, and never insert or remove tuples from the view.

We assume for the purposes of determining the cost of maintaining a view that each type of change to each lower view is considered separately. Therefore, the cost of maintaining a view or supporting view V is the sum of the costs of propagating (onto V) each type of change to each of the lower views involved in V. For example, assuming $V = R \bowtie S \bowtie T$:

- **Insertions**: The cost of propagating insertions to R onto V is the cost of evaluating $\triangle R \bowtie S \bowtie T$, inserting the result into V, and updating the indices of V. When propagating insertions it is often possible to reuse the result of propagating insertions onto one view in propagating insertions onto another. For example, if V is a supporting view of $V' = R \bowtie S \bowtie T \bowtie U$, then we can reuse the result of propagating insertions to R onto V ($\triangle R \bowtie S \bowtie T$) when propagating insertions to R onto V' ($\triangle R \bowtie S \bowtie T \bowtie U$). In this respect we consider a limited but important form of multiple-query optimization.

- **Deletions**: The cost of propagating deletions to R ($\bigtriangledown R$) onto V is the cost of evaluating $V \ltimes \bigtriangledown R$ (we use $\ltimes$ for semijoin), removing those tuples from V, and updating the indices of V.
- **Updates**: The cost of propagating updates to R (μR) onto V is the cost of evaluating $V \ltimes \mu R$ and updating those tuples in V. Note that because we allow propagating only protected updates in this manner, we do not have to update the indices of V since we build indices only on attributes involved in selection conditions or join conditions or keys, and these attributes cannot be modified by a protected update.

4.4 Optimal Solution Using A* Algorithm

In this section we describe an optimal algorithm to solve the VIS problem and then show through experimental results that it vastly reduces the number of candidate solutions that must be considered.

4.4.1 Algorithm Description

In this section we describe how we have used the A* algorithm to solve the VIS problem. (For further details on the A* algorithm itself, the reader is referred to [Nil71].) An algorithm based upon A* is guaranteed to derive an optimal solution to a problem but attempts to prune the parts of the search space that cannot contain the optimal solution.

The algorithm takes as input the set of all possible views and indices to materialize, $\mathcal{M}$. $\mathcal{M}$ does not include the lower views ($\mathcal{B}$) nor the primary view V_P but includes indices that can be defined on them. (Recall that V_P and the lower views are constrained to be materialized.) The goal of the algorithm is to choose a subset $\mathcal{M}'$ of $\mathcal{M}$ to materialize such that the total cost, $\mathcal{C}$, is minimized. The total cost given a particular subset of views and indices $\mathcal{M}'$ can be expressed as

$$\mathcal{C}(\mathcal{M}') = \sum_{m \in (\mathcal{M}' \,\cup\, \mathcal{B} \,\cup\, \{V_P\})} \mathit{maint\_cost}(m, \mathcal{M}').$$

Function $\mathit{maint\_cost}(m, \mathcal{M}')$ returns the cost of propagating all changes to view or index m assuming only the views and indices in $\mathcal{M}'$ (along with $\mathcal{B}$ and V_P) are materialized.

Instead of directly searching the power set of $\mathcal{M}$, we set up the A* search to build the solution incrementally. It begins with an empty materialization set ($\mathcal{M}' = \phi$) and then *considers* adding single views or indices. The algorithm terminates when a solution is found that has considered every view and index and is guaranteed to have the minimum total cost. We will call the intermediate steps reached in the algorithm as *partial states*. Each partial state is described by the tuple $\langle \mathcal{M}_C, \mathcal{M}' \rangle$ where $\mathcal{M}_C$ is the set of features (view or index) from $\mathcal{M}$ that have been considered and $\mathcal{M}'$ is the set of features from $\mathcal{M}_C$ that have been chosen to be materialized. For convenience, we will also refer to the set of unconsidered features, $\mathcal{M}_U$, which is $\mathcal{M} - \mathcal{M}_C$.

Presented with a set of partial states from which to incrementally search, A* attempts to choose the most promising. It does so by *estimating* the cost of the best solution $\mathcal{M}' \cup \mathcal{M}'_U$ that can be achieved from each partial state. $\mathcal{M}'_U$ is the unconsidered features that would be chosen to be materialized in addition to $\mathcal{M}'$.

The *exact* cost of the best solution given a partial state can be decomposed as

$$\mathcal{C} = g + h,$$

where g is the maintenance cost for the features chosen so far ($\mathcal{M}'$) and h is the maintenance cost for the features in $\mathcal{M}'_U$. In general, g also needs $\mathcal{M}'_U$ for its computation; that is, it is necessary to know which unconsidered features will be chosen in order to compute the maintenance cost of features in $\mathcal{M}'$. Fortunately, we can compute g using only $\mathcal{M}'$ so long as we impose a partial ordering on the features in $\mathcal{M}$ so that we only consider a feature when a decision has been made on every feature that affects its maintenance cost. Formally, a partial order $\prec$ is imposed upon $\mathcal{M}$ such that if a feature m_2 is an index on a view m_1, then $m_1 \prec m_2$. Otherwise, if a feature m_1 can be used in a query plan for propagating insertions to view m_2, then $m_1 \prec m_2$.

The exact formula for h is

$$\min_{\mathcal{M}'_U \subseteq \mathcal{M}_U} \Big(\sum_{m \in \mathcal{M}'_U} \mathit{maint\_cost}(m, \mathcal{M}' \cup \mathcal{M}'_U) \Big).$$

Unfortunately, this formula requires an exhaustive search to find the $\mathcal{M}'_U$ that minimizes the equation.

Instead of performing this exhaustive search, we calculate a lower bound on h denoted $\hat{h}$. Using the lower bound, the A* algorithm can prune some of the partial states while still guaranteeing an optimal solution. (The amount of pruning depends on how close $\hat{h}$ estimates h.) Using $\hat{h}$, for any partial state we can compute a lower bound on $\mathcal{C}$ as

$$\hat{\mathcal{C}} = g + \hat{h}.$$

Input: $\mathcal{M}$, $\prec$
Output: Optimal $\mathcal{M}'$

Let state set $\mathcal{S} = \{s\}$, where s is a partial state having
  $\mathcal{M}_C(s) = \mathcal{M}'(s) = \phi$, and $\mathcal{M}_U(s) = \mathcal{M}$ (lower views and V_P are materialized)
Loop
  Select the partial state $s \in \mathcal{S}$ with the minimum value of $\hat{C}$
  If $\mathcal{M}_C(s) \equiv \mathcal{M}$, return $\mathcal{M}'(s)$
  Let $\mathcal{S} = \mathcal{S} - \{s\}$
  For each view or index $m \in \mathcal{M}_U(s)$ such that for all $m' \prec m$: $m' \in \mathcal{M}_C(s)$
    Construct partial state s' such that
      $\mathcal{M}_C(s') = \mathcal{M}_C(s) \cup \{m\}$, $\mathcal{M}_U(s') = \mathcal{M}_U(s) - \{m\}$, $\mathcal{M}'(s') = \mathcal{M}'(s) \cup \{m\}$
    Construct partial state s'' such that
      $\mathcal{M}_C(s'') = \mathcal{M}_C(s) \cup \{m\}$, $\mathcal{M}_U(s'') = \mathcal{M}_U(s) - \{m\}$, $\mathcal{M}'(s'') = \mathcal{M}'(s)$
    Let $S = S \cup \{s'\} \cup \{s''\}$
  Endfor
Endloop

Figure 4.5: A* Algorithm.

Note that if $\mathcal{M}_C \equiv \mathcal{M}$ then $\hat{\mathcal{C}} = \mathcal{C}$. We will develop an expression for $\hat{h}$ below but first we present the A* algorithm for the VIS problem.

The algorithm appears in Table 4.5. The state set $\mathcal{S}$ contains all active partial states. It initially contains only the partial state where none of the views and indices have been considered. Each time through the loop the algorithm selects the partial state with the minimum lower bound on the cost. If the selected state has $\mathcal{M}_C \equiv \mathcal{M}$, it is guaranteed to be the optimal choice. Other active states in S need not be expanded further. If the selected state is not a complete state, it is removed from the set of active states and for each view or index that can be added to the set of considered views and indices without violating the partial order, two states are added to the set of active states: one with the view or index added to the chosen set ($\mathcal{M}'$), and one without.

The formula for $\hat{h}$ computes the cost of maintaining views and indices in $\mathcal{M}_U$ minus the upper bound of their benefit toward maintaining other views (including V_P).

$$\hat{h} = \sum_{m \in \mathcal{M}_U} (h\_maint\_cost(m, \mathcal{M}') - max\_benefit(m, \mathcal{M}'))$$

We guarantee that any overestimation of the actual maintenance cost of m is more than compensated for by the overestimation of the benefit. Note that our function $\hat{h}$, although it achieves considerable pruning, can be improved. Deriving a tighter lower bound for h that can be computed efficiently is a subject for future research.

The function $h\_maint\_cost(m, \mathcal{M}')$ differs depending on whether m is a view or an index. If m is an index, the function returns the cost of maintaining m for all insertions and deletions that will be propagated to the view that m is on. (The details of our cost model are found in Appendix B.)

If m is a view, the function returns the cost of propagating onto m insertions to each of the lower views referenced in m, plus the cost of propagating onto m deletions and updates to each of the lower views referenced in m assuming an index exists in m for the key attribute of each lower view. Note that when m is a view, we might overestimate the cost for propagating insertions since we are assuming that all other views in $\mathcal{M}_U$ are not materialized (this overestimation is compensated for in *max_benefit*).

The function $max\_benefit(m, \mathcal{M}')$ also differs depending on whether m is a view or an index. First we consider the case where m is an index.

1. If m is an index on a view v for the key attribute of a lower view R that is referenced in v, the function returns the cost of propagating deletions and updates from R to v without m minus the cost of propagating deletions and updates from R to v with m.

2. If m is an index on a view v for a join attribute that joins v to some relation R not referenced in v, the function sums for each view $v' \in \mathcal{M}_U$ that includes R as well as all the relations in v and for every relation S in v' but not in v, the cost of scanning v (the maximum savings due to an index join using m when propagating insertions from s onto v').

3. If m is an index for both a key and a join attribute, the two benefits described are added.

Next we consider the case where m is a view. Intuitively, the maximum benefit of m is the cost of materializing m when propagating insertions to views for which m is a subview. The *max_benefit* function sums for each view $v' \in \mathcal{M}_U$ that includes all the relations in m and for every relation S in v' but not in m, the cost of materializing m given the views and indices in $\mathcal{M}_C$. Any overestimation in the cost of propagating insertions in the function

# of relations	# of selections	# of states visited exhaustive	A*	% pruned
2	0	32	11	67.7
2	1	192	21	89.1
2	2	960	28	97.1
2	4	960	29	97.0
3	1	2115072	17735	99.2
3	2	10575360	22809	99.8

Table 4.1: Comparison of A* and Exhaustive Algorithms.

$h\_maint\_cost(m, \mathcal{M}')$ is offset by including the cost of materializing the views in the function $max\_benefit(m, \mathcal{M}')$.

4.4.2 Experimental Results

To test our A* algorithm, we implemented both the A* algorithm and the exhaustive algorithm described in Section 4.2. We then ran simulations using both algorithms on a variety of sample schemas. A summary of the results is presented in Table 4.1. Clearly, the A* algorithm is performing very well, pruning the vast majority of the search space. As the problems gets larger, due to more views or selection predicates, its relative performance increases as well. While it may still be possible to derive a tighter lower bound on h, our algorithm is a vast improvement over other algorithms previously proposed.

4.5 Rules of Thumb

The A* algorithm presented in the last section yields optimal solutions while achieving impressive pruning. Still, because the solution space of the VIS problem grows doubly exponentially with the number of lower views, primary views that are computed from many lower views (*i.e.*, 4 or more) may be still too large to handle. Fortunately, finding an optimal solution is not critical since there are often many solutions that are close to optimal. The space of solutions is illustrated by Figure 4.6. Each point on the x-axis represents a particular view set, and the y-axis measures the total maintenance cost for the given view set. The range of values depicted by the bar on the y-axis shows the total maintenance costs due to choosing the best and worst index sets for the given view set. This figure emphasizes two things: (1) there are several view sets that are close to optimal, and (2) index selection

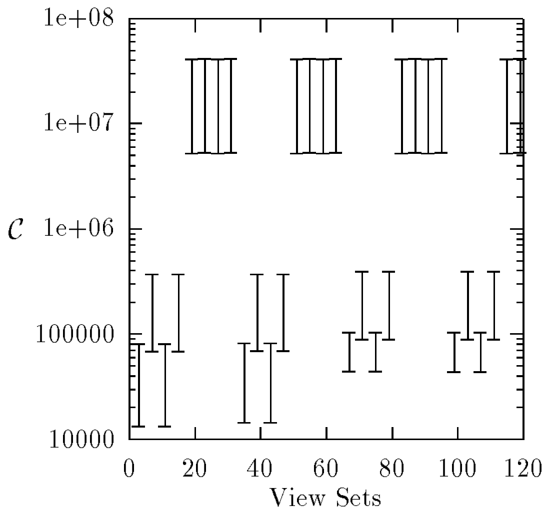

Figure 4.6: A Sample Solution Space.

is very important even after picking a good view set. What is required then to come up with a reasonable solution to the VIS problem is to avoid poor view sets and then to pick a good index set.

In this section we propose rules of thumb that can help guide a warehouse administrator (WHA) in choosing a reasonable set of supporting views without resorting to the A* algorithm. Later, in Section 4.6, we show how some of these rules of thumb can be used in an efficient heuristic algorithm to address the VIS problem. The underlying theme of these rules of thumb is to materialize a supporting view if its *benefit* (reduction in IO cost) is greater than its *cost* (increase in IO cost). The rules of thumb function similarly to the rule "join small relations first" in query optimization. These are not hard and fast rules: many factors come into play and some rules tend to work against others. But we have found that the rules apply in general. Even when the rules do not apply, the cost-benefit analysis introduced in explaining each rule can help the WHA decide what to materialize in a specific situation.

We justify each rule of thumb through analysis and also through experimentation. The formulas we use in the analysis are rough approximations of the actual benefits and costs. However, a significantly more detailed and accurate cost model was included in the VIS optimizer used in the experiments (see Appendix B). Since the rules of thumb are supported by the results of the VIS optimizer, it seems that the approximations used in these simpler formulas are reasonable.

The supporting experiments are performed for views composed of only three lower views to keep the problem tractable. We expect that for a view joining more than three relations, the differences in the graphs would be more pronounced because there would be more opportunities to apply the rule.

Notation	Description
$\mathcal{C}$	Total maintenance cost for the primary view, supporting views, and indices
V	Primary view, supporting view, or lower view
$\mathcal{R}(V)$	lower views involved in V
$\lvert\mathcal{R}(V)\rvert$	Number of lower views involved in V
$\overline{\mathcal{R}(V)}$	lower views not involved in V but in primary view
$\mathcal{E}(V)$	Elements (materialized supporting views or lower views) joined in V
$\lvert\mathcal{E}(V)\rvert$	Number of elements joined in V
P_m	Number of pages of memory for database buffer
$P(V)$	Number of pages in V
$T(V)$	Number of tuples in V
$I(V)$	Number of insertions to V
$D(V)$	Number of deletions from V
$U(V)$	Number of updates to V
$P(\mathcal{R}(V))$	Sum of the pages in all of the lower views involved in V
$I(\mathcal{R}(V))$	Sum of the number of insertions to all of the lower views involved in V
$D(\mathcal{R}(V))$	Sum of the number of deletions from all of the lower views involved in V
$U(\mathcal{R}(V))$	Sum of the number of updates to all of the lower views involved in V
$P(\mathcal{E}(V))$	Sum of the pages in all of the elements of V
$P(V, R.A)$	Number of pages in an index on V for attribute $R.A$
$S(V, p)$	Number of tuples in V passing the selection condition p (if p is a join condition then it is the number of tuples in V that join with a single tuple in the other relation)

Table 4.2: Notation Used in Rules of Thumb.

4.5.1 Schema and Notation

The statistics given in Table 4.2 are used in evaluating the rules of thumb. Since the rules of thumb are very approximate, the WHA needs only rough approximations of the statistics. The sensitivity of the results to estimation errors is studied in Section 4.7.3. Two points about the table need to be made. First, we define the functions $P(\overline{\mathcal{R}(V)})$, $I(\overline{\mathcal{R}(V)})$, $D(\overline{\mathcal{R}(V)})$, and $U(\overline{\mathcal{R}(V)})$ to have their expected meaning. For instance, $P(\overline{\mathcal{R}(V)})$

Relations	$T(V)$	$I(V)$	$D(V)$	Select	Join
$R(\underline{R0}, R1)$	90M	1%	0.01%		$S(R,\ R1 = SO) = 3$
$S(\underline{S0}, S1)$	30M	1%	0.01%		$S(S,\ S0 = R1) = 1$
					$S(S,\ S1 = T0) = 3$
$T(\underline{T0}, T1)$	10M	1%	0.01%	$S(T,\ T1 <= 10)$ =1M	$S(T,\ T0 = S1) = 1$
$R(\underline{R0}, R1, R2)$	20M	1%	0.01%		$S(R,\ R0 = S1) = 1$
$S(\underline{S0}, S1, S2)$	20M	1%	0.01%	$S(S,\ S2 <= 20)$ =2M	$S(S,\ S1 = R0) = 1$
					$S(S,\ S2 = T0) = 1$
$T(\underline{T0}, T1, T2)$	20M	1%	0.01%		$S(T,\ T0 = S2) = 1$

Table 4.3: View Schemas.

denotes the sum of the pages in all of the lower views that are in the primary view but not in the view V. Second, if for the definition of $\mathcal{E}(V)$ there is more than one possible set of materialized supporting views and lower views that can be joined to derive V, then we assume that a set having the fewest number of elements (lower views or supporting views) is chosen. For example, suppose that view V is defined as $R \bowtie S \bowtie T$, then $\mathcal{E}(V) = \{R, S, T\}$ and $\mathcal{R}(V) = \{R, S, T\}$. If another view $V' = R \bowtie S$ is then materialized, $\mathcal{E}(V) = \{V', T\}$ because this set has only two elements (but $\mathcal{R}(V) = \{R, S, T\}$ still holds).

As mentioned, the rules of thumb proposed in this section will be supported with experimental results. All of the tests were run with one of the two primary view schemas depicted in Table 4.3. The first four rows of Table 4.3 depict Schema 1, while the last four rows depict Schema 2. The "Relations" column of Figure 4.3 shows the attributes in each relation with the key attribute underlined. The next column ($T(V)$, using the notation in Table 4.2), gives the cardinality of each relation. The $I(V)$ and $D(V)$ columns give the number of insertions and deletions, respectively, as a percentage of $T(V)$. The updates were set to 0. The next two columns show the selection and join conditions using notation in Table 4.2. Schema 1 is a *linear join*, $V_P = R \bowtie S \bowtie \sigma_{T1 \leq 10} T$, where both joins are on foreign keys and $S(T, T1 \leq 10) = 0.10 \cdot T(T)$. The relative cardinalities are $T(R) = 3 \cdot T(S) = 9 \cdot T(T)$. Schema 2 is also a linear foreign key join, $V_P = R \bowtie \sigma_{S2 \leq 20} S \bowtie T$, where $S(S, S2 \leq 20) = 0.10 \cdot T(S)$, but with all of the relations having the same cardinalities.

4.5.2 When to Materialize Supporting Views

We now give several rules of thumb governing which supporting views to materialize. The rules of thumb are based upon formulas estimating the benefit and cost of materializing a supporting view assuming that updates are done in batches, which is common in a data warehousing environment. We list the rules of thumb first, then analyze them using the formulas, and graph the results of our supporting experiments.

Rule 4.5.1 (Materialize Selective Views) *Materialize a supporting view* V *when* $P(V) \ll P(\mathcal{E}(V))$. □

Rule 4.5.2 (Materialize Views Having No Deletions or Updates) *Materialize a supporting view* V *when* $D(\mathcal{E}(V)) + U(\mathcal{R}(V)) = 0$. □

Intuitively, Rules 4.5.1 and 4.5.2 guide the WHA to materialize a supporting view V either when the view will be much smaller than the sum of the sizes of the contributing lower views, or when no deletions or updates are expected to the contributing lower views. We assume in these rules that a supporting view V does not *overlap* with any other materialized supporting view V' (although it is acceptable if the relations of V are a subset of the relations of V'). That is, for every other materialized supporting view V', either $\mathcal{R}(V) \cap \mathcal{R}(V') = \phi$, $\mathcal{R}(V) \subset \mathcal{R}(V')$, or $\mathcal{R}(V') \subset \mathcal{R}(V)$. The rule of thumb governing when to materialize overlapping supporting views is presented later in this section.

Rule 4.5.1 can support materializing a view even when Rule 4.5.2 doesn't hold. An example where Rule 4.5.1 is likely to hold is when $V = S \bowtie \sigma_p T$ and the selectivity of the selection condition p is low. These conditions imply that the sizes of S and T together would exceed the size of V (*i.e.*, $P(V) \ll P(\mathcal{E}(V))$).

In order to justify our rules of thumb and give a more detailed analysis of when to materialize a supporting view, we give approximate formulas for calculating the benefit and cost of materializing a supporting view (denoted as $Benefit_v$ and $Cost_v$). In general, a supporting view V should be materialized when $Benefit_v(V) > Cost_v(V)$. The formula for the benefit of a supporting view is:

$$Benefit_v(V) \approx \begin{cases} (|\mathcal{E}(V)| - 1) \cdot I(\overline{\mathcal{R}(V)}) & \text{if } V \text{ is indexed on the appropriate join attributes} \\ & \quad \text{and } (|\mathcal{E}(V)| - 1) \cdot I(\overline{\mathcal{R}(V)}) < P(\mathcal{E}(V)) - P(V) \\ P(\mathcal{E}(V)) - P(V) & \text{otherwise} \end{cases}$$

A materialized supporting view is beneficial because it contains pre-joined relations. Consider a materialized supporting view V. The benefit of V to a view-maintenance expression E, where E propagates insertions from a lower view S that is outside of V (*i.e.*, in $\overline{\mathcal{R}(V)}$) to the primary view V^P is the difference between the cost of performing the joins between the insertions to S, the other relations in $\overline{\mathcal{R}(V)}$, and the elements of V^P, and the cost of performing just the joins between the insertions to S, the other relations in $\overline{\mathcal{R}(V)}$, V. That is, the benefit lies in not having to recompute V.

The actual benefit thus depends upon the type of join that would be used to join with the elements that make up V. If index joins would be used, then the benefit of materializing V is proportional to $(|\mathcal{E}(V)| - 1) \cdot I(\overline{\mathcal{R}(V)})$, the approximate number of index joins that would be saved (the difference between joining insertions to the relations that are not in V to all of the relations in V, and the cost of joining the insertions just to V). Otherwise, if the number of insertions to propagate is large enough that nested-block joins are used, then we assume that the insertions are always the smaller relation and that they will always fit in memory, so we can calculate the benefit as the sum of the number of pages in the elements of V (roughly the cost of joining with each of the elements of V) minus the number of pages in V (roughly the cost of joining with V).

The cost of materializing a supporting view V is estimated as the cost of propagating deletions and updates from the relations in V onto V itself, plus the cost of maintaining the indices on V. The formula is:

$$Cost_v(V) \approx \begin{cases} D(\mathcal{R}(V)) + U(\mathcal{R}(V)) + Cost_i(V, *) & \text{if } V \text{ is indexed on the keys of relations} \\ & \text{in } \mathcal{R}(V) \text{ and} \\ & D(\mathcal{R}(V)) + U(\mathcal{R}(V)) < P(V) \cdot |\mathcal{R}(V)| \\ P(V) \cdot |\mathcal{R}(V)| + Cost_i(V, *) & \text{otherwise} \end{cases}$$

In the formula, $Cost_i(V, *)$ denotes the cost of maintaining all of the indices built on V. We observe that the cost of maintaining an index on view V is proportional to the number of insertions and deletions to V if the index is too large to fit into memory, or proportional to the number of pages in the index if the index fits entirely into memory. The cost of maintaining an index on V is summarized below. (The index is on a V attribute derived from $R.A$.)

$$Cost_i(V, R.A) \approx \begin{cases} P(V, R.A) & \text{if } P(V, R.A) < P_m \\ I(\mathcal{R}(V)) + D(\mathcal{R}(V)) & \text{otherwise} \end{cases}$$

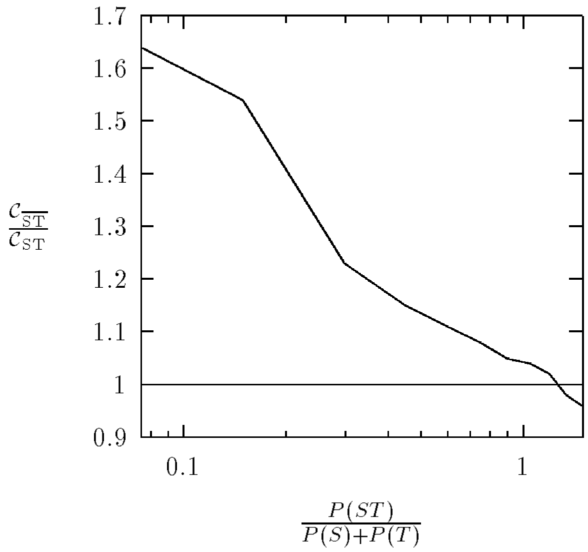

Figure 4.7: Support for Rule 4.5.1

Hence, $Cost_i(V, *)$ is just the sum of the cost of maintaining each index built on V. The cost of maintaining each index is obtained from $Cost_i(V, R.A)$.

If V is indexed on the keys of the lower views and the cost of index joins is less than that of nested-block joins, then the cost of maintaining V is proportional to the number of deletions and updates to $\mathcal{R}(V)$, since each deletion and update results in tuple lookups through the index. Otherwise, if nested-block joins are used, the cost is proportional to $P(V)$ times the number of lower views in V, since we have to scan V to find the tuples deleted or updated due to the changes to each lower view.

One might notice that we have not included the cost of propagating insertions onto V in the cost formula above. To see why, consider a primary view $RST = R \bowtie S \bowtie T$ and a supporting view $ST = S \bowtie T$. The reason for ignoring the cost of propagating insertions onto ST is that the expressions that propagate insertions onto ST are subexpressions of the expressions that propagate insertions onto RST. For example, the expression propagating insertions from T onto ST ($S \bowtie \triangle T$) is a subexpression of the expression propagating insertions from T onto RST ($R \bowtie S \bowtie \triangle T$). Therefore we can reuse the result of propagating insertions onto ST when maintaining RST, and thus ignore the cost of propagating insertions onto ST. The only significant effect insertions have is in $Cost_i(ST, *)$, maintaining the indices of ST.

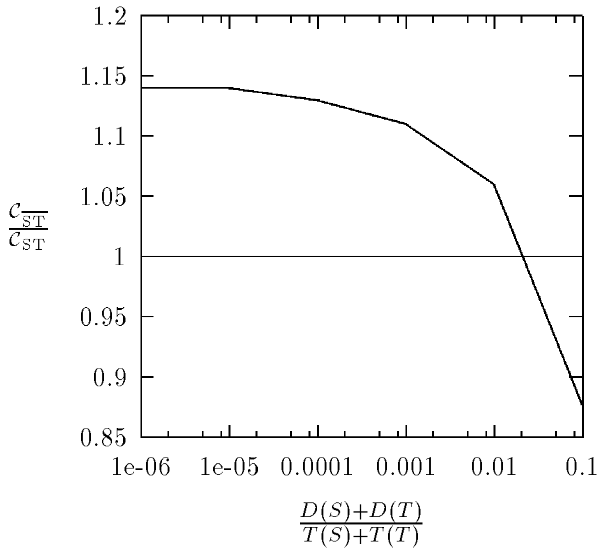

Figure 4.8: Support for Rule 4.5.2

Rule 4.5.1 and Rule 4.5.2 are examples of rules that may work for or against each other. For instance if $D(\mathcal{E}(V)) + U(\mathcal{R}(V))$ is very high, it may not be beneficial to materialize V even if $P(V) \ll P(\mathcal{E}(V))$ holds. In the experiments we fix the parameters involved in one rule while we vary the parameters in the other to show the effect.

Figure 4.7 shows the experimental support for Rule 4.5.1 using Schema 1. In this experiment we consider two view sets, one with the supporting view $ST = S \bowtie T$ and one without. For both view sets the whole index space was searched to obtain the best index set at each point in the graph. (As shown in Table 4.3, the deletion rate is set at 0.01% and the update rate is set at 0% for Schema 1.) The graph shows how the ratio of the total update cost without view ST over the total update cost with view ST materialized ($\mathcal{U}_{\overline{ST}}/\mathcal{U}_{ST}$) varies with $P(ST)/(P(S)+P(T))$. Therefore, it is beneficial to materialize ST when the line in the graph is above 1.0. As $P(ST)$ gets larger, it is less and less beneficial to materialize ST as predicted by Rule 4.5.1. Note that in our scenarios with so few deletions and updates, even when $P(ST) = P(S)+P(T)$ it is still beneficial to materialize ST for this schema because evaluating the maintenance expression $\triangle R \bowtie S \bowtie T$ is still more expensive than evaluating $\triangle R \bowtie ST$ (when ST is materialized). The reason is that there are more tuples in $\triangle R$ that match with S than with ST because of the selection condition on T.

Our next experiment (also on Schema 1) shows experimental support for Rule 4.5.2. In this experiment the ratio $P(ST)/(P(S)+P(T))$ was set to 0.5. For simplicity the update rate were set to 0 and only the deletion rates were varied. Figure 4.8 shows that as the deletion rates to S and T (as a fraction of $T(S)$ and $T(T)$, the number of tuples in S and T) increase, it is less and less beneficial to materialize ST.

Rule 4.5.1 and Rule 4.5.2 assume that the view V does not overlap with any other supporting view. That is, for each supporting view V' distinct from V, either $\mathcal{R}(V) \cap \mathcal{R}(V') = \phi$, or $\mathcal{R}(V) \subset \mathcal{R}(V')$, or $\mathcal{R}(V') \subset \mathcal{R}(V)$. We now give the rule for materializing V when it overlaps with other views.

Rule 4.5.3 (Materialize Non-Overlapping Views) *In general, materialize supporting views that do not overlap. Materialize overlapping supporting views V_1 and V_2 only when $I(\mathcal{R}(V_1) \cap \mathcal{R}(V_2)) = 0$.* □

Intuitively, Rule 4.5.3 directs the WHA to materialize overlapping supporting views only when there are no insertions to lower views in the intersection of the overlapping views.

Suppose that one supporting view V_1 is contained in (but does not overlap) another supporting view V_2. That is, $\mathcal{R}(V_1) \subset \mathcal{R}(V_2)$. We showed previously (with views RST and ST) that the result of propagating insertions to V_1 can be reused in propagating insertions to V_2, so in this case little additional cost is incurred and Rule 4.5.3 does not recommend against materializing both views (assuming both views are deemed beneficial by Rules 4.5.1 and 4.5.2).

Now, suppose that two supporting views overlap. For example, let the primary view be $RST = R \bowtie S \bowtie T$, with supporting views $RS = R \bowtie S$ and $ST = S \bowtie T$. The problem with materializing both RS and ST is that insertions to S must be propagated to both RS and ST. The result of propagating insertions to S onto RS cannot be reused in propagating insertions to S onto ST, or vice-versa. Furthermore, since RS and ST are both subsets of the primary view RST, *only one* of the two results can be reused when propagating insertions to RST. Therefore, propagating insertions to overlapping views creates additional work that cannot be masked by containing views.

The additional work required to propagate insertions requires that we modify the formulas for the cost and benefit of overlapping views, $Benefit_{ov}$ and $Cost_{ov}$ respectively. Using the new formulas we can state the requirement for materializing overlapping supporting

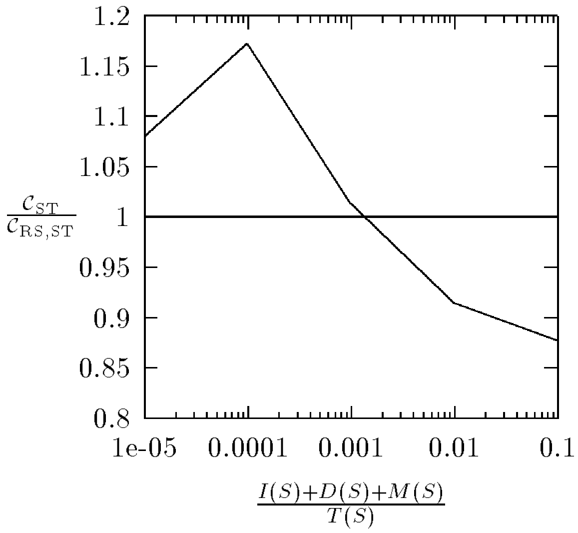

Figure 4.9: Support for Rule 4.5.3.

views V_1 and V_2 as

$$\begin{aligned}&(\mathit{Benefit}_{ov}(V_1) - \mathit{Cost}_{ov}(V_1)) + (\mathit{Benefit}_{ov}(V_2) - \mathit{Cost}_{ov}(V_2))\\&\quad > \mathit{max}((\mathit{Benefit}_{v}(V_1) - \mathit{Cost}_{v}(V_1)), (\mathit{Benefit}_{v}(V_2) - \mathit{Cost}_{v}(V_2))).\end{aligned}$$

In other words, materialize both V_1 and V_2 if the gain (*i.e.*, benefit minus cost) obtained by materializing both is greater than the maximum gain obtained by materializing one or the other.

The upper limit on $\mathit{Benefit}_{ov}(V)$ is $\mathit{Benefit}_{v}(V)$, and the following example illustrates why $\mathit{Benefit}_{ov}(V)$ is less than $\mathit{Benefit}_{v}(V)$. Let $RST = R{\bowtie}S{\bowtie}T$, $RS = R{\bowtie}S$, and $ST = S{\bowtie}T$ as before. The benefit of ST if it were non-overlapping is roughly the cost of evaluating $(\triangle R{\bowtie}S{\bowtie}T)$ minus the cost of evaluating $(\triangle R{\bowtie}ST)$. However, due to the overlapping RS, $(\triangle R{\bowtie}S)$ must be performed to maintain RS. Therefore, the benefit of the overlapping ST is just the cost of the single join between the result of $(\triangle R{\bowtie}S)$ and T, minus the cost of evaluating $(\triangle R{\bowtie}ST)$.

The formula for Cost_{ov} appears below. Insertions into the intersecting relations must now be taken into account, so the additional cost of propagating insertions is added to the

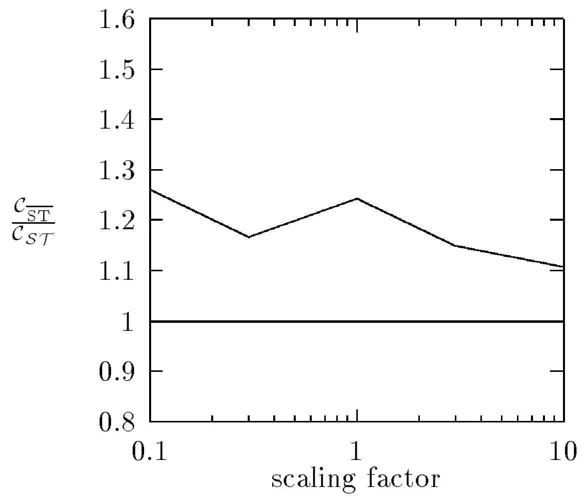

Figure 4.10: Support for Rule 4.5.4.

cost of materializing a (non-overlapping) view.

$$Cost_{ov}(V_1) \approx I(\mathcal{R}(V_1) \cap \mathcal{R}(V_2)) + Cost_v(V_1)$$

If, as in Rule 4.5.3, the number of insertions to intersecting relations is zero, then there is no additional cost in maintaining the overlapping views (over that for maintaining the two views as if each were non-overlapping). In that case, if each view on its own is beneficial to materialize, then it is likely to be worthwhile to materialize both of them even though they overlap.

The experimental support for this rule is shown in Figure 4.9. In this experiment we considered two sets of supporting views for Schema 2, one with overlapping supporting views ($RS = R \bowtie S$ and $ST = S \bowtie T$) and one with $ST = S \bowtie T$ only. (As shown in Table 4.3, the deletion rate is set at 0.01% and the update rate is set at 0% for Schema 2.) For each view set, the whole index space was searched to obtain the best index set at each point in the graph. We graphed the ratio of the update costs of these view sets and varied the insertion rate to S. Figure 4.9 shows that as the insertion rate is increased, it is less beneficial to materialize the overlapping views.

The previous three rules were not concerned about the size of main memory. The following rule tells the WHA that when considering whether or not to materialize a supporting view V, the size of V in relation to the size of main memory doesn't matter.

Rule 4.5.4 (Size Doesn't Matter) *In considering whether to materialize a supporting view V, the ratio of $P(V)$ to P_m doesn't matter.* □

That is, the total number of pages of a supporting view relative to the number of pages of memory does not significantly impact the choice of whether to materialize the supporting view (unless of course the WHA is also trying to conserve space—see Section 4.7.1). Note that in the approximate formulas given for benefit and cost, P_m does not come into play. In our more detailed cost model (Appendix B), $P(V)$ relative to P_m has an impact primarily for index joins and index maintenance in which case small supporting views and indices that fit entirely in memory have an advantage. But once a supporting view and its indices grow beyond the size of memory then its size is not significant.

Figure 4.10 graphs the cost of maintaining two sets of supporting views for a primary view $RST = R \bowtie S \bowtie T$: one that includes a supporting view $ST = S \bowtie T$ and another where ST is not materialized. We vary the sizes of all lower views as well as the number of changes to the lower views proportionately, while holding the number of pages of memory constant. Note that the size of V has little effect on the decision of whether to materialize it.

4.6 Heuristic Approaches

The VIS problem involves selecting a set of supporting views to materialize along with index selection on the chosen set of views. We have already shown that both components of the problem are doubly exponential. Thus, heuristic algorithms are required when the primary view involves a fair number of lower views. From our experience, when $n \geq 4$ lower views are involved, the A* optimal algorithm becomes unwieldy. In this section, we outline heuristic algorithms that pare down on both the view search space and the index search space.

4.6.1 Algorithm Descriptions

The first two heuristic algorithms we present, called *Rete* and *A-TREAT*, come from previous research in production rules systems and active databases. In such systems, the user specifies an *action* that is to be performed when a *condition* becomes true. The condition is normally expressed as a query over the database that is true when its result is non-empty. Thus, the problem of checking when a condition is true is equivalent to maintaining a materialized view defined by the condition query and executing the action if the view becomes non-empty. In addition to *Rete* and *A-TREAT*, we propose three new heuristic algorithms for the VIS problem – *NOGI*, *GVGI*, and *VSIS*. We compare the performance of all the algorithms in Section 4.6.2.

All of the 5 heuristic algorithms use a greedy index selection algorithm, called *GI*, which we now describe. Given a set of supporting views $\mathcal{V}$, *GI* first identifies the set of indices $\mathcal{I}$ that can be built over the views in $\mathcal{V}$, the lower views $\mathcal{B}$, and the primary view V_P. *GI* then greedily chooses to materialize the index that attains the largest reduction of the update cost $\mathcal{C}$. The chosen index is then removed from $\mathcal{I}$. *GI* chooses indices to materialize until no index in $\mathcal{I}$ reduces $\mathcal{C}$.

Clearly, the complexity of *GI* is $O(|\mathcal{I}|^2)$. We have already shown that given any set of supporting views $\mathcal{V}$, the number of indices in $\mathcal{I}$ is at most $k \cdot 2^n$, where k is the maximum number of attributes in a view and n is the number of views involved in the primary view. Hence, *GI* significantly pares down the doubly exponential index search space $(O(2^{2^n}))$.

Rete Algorithm

The *Rete* algorithm [For82] examines the various left-deep join trees of the primary view definition query. For instance, given a primary view $V_P = \sigma_p R \bowtie S \bowtie T$, one left-deep join tree of the definition query is shown in Figure 4.11. Given that V_P involves n lower views, it is not hard to see that there are $O(n!)$ left-deep join trees. *Rete* considers the various sets of supporting views obtained from the various left-deep join trees. More specifically, for each left-deep join tree, *Rete* considers the set of supporting views $\mathcal{V}$ that corresponds to the interior nodes of the left-deep join tree (except for the node that represents the primary view V_P). Hence, in Figure 4.11, *Rete* considers materializing the set of supporting views $\mathcal{V} = \{\sigma_p R, \sigma_p R \bowtie S\}$, and evaluates the update cost $\mathcal{C}$. *Rete* then chooses the set of supporting views that achieves the smallest update cost. Since *Rete* only considers one set of supporting views for each left-deep join tree, only $O(n!)$ sets of supporting views are examined.

Strictly speaking, the *Rete* algorithm does not consider index selection. Since *Rete* was used in active databases and changes are propagated immediately in such databases, [For82]

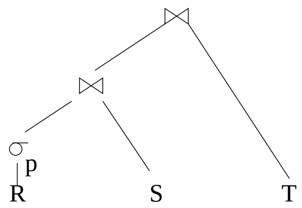

Figure 4.11: A Left-deep Join Tree Considered by *Rete*.

suggests that all indices that may be helpful must be chosen. However, we have already seen that careful index selection is critical in order to obtain low update costs. Therefore, we will enhance *Rete* with the *GI* algorithm explained previously. That is, for each set of supporting views $\mathcal{V}$ considered by *Rete*, *GI* is used to choose the indices on $\mathcal{V} \cup \mathcal{B} \cup \{V_P\}$ before the update cost $\mathcal{C}$ is evaluated.

A-TREAT Algorithm

The *A-TREAT* algorithm is different from *Rete* in that it only considers materializing interior nodes of the primary view definition query tree that correspond to selection nodes. The original *A-TREAT* algorithm presented in [Han92] materializes a selection node depending on the selectivity. In this chapter, we will search through all possible choices which makes the view search space for *A-TREAT* $\mathcal{O}(2^n)$ if every relation has a selection condition. For instance, given a primary view $V_P = \sigma_p R \bowtie S \bowtie \sigma_{p'} T$, the various sets of supporting views considered are: $\{\}$, $\{\sigma_p R\}$, $\{\sigma_{p'} T\}$, $\{\sigma_p R, \sigma_{p'} T\}$. Like *Rete*, *A-TREAT* does not specify how indices are selected. Therefore, we will also enhance *A-TREAT* with the *GI* algorithm.

NOGI Algorithm

In Section 4.5.2, we developed rules of thumb to guide the WHA in view selection. One of the rules of thumb stated that it is not usually beneficial to materialize two overlapping views (Rule 4.5.3). Hence, the *NOGI* algorithm does not consider a set of supporting views that contains overlapping supporting views. Table 4.4 shows that by using Rule 4.5.3, *NOGI* prunes a large number of view sets. For each view set considered, *NOGI* uses *GI* to select the indices.

# of relations	# of view sets	# of view sets with overlapping views	% view sets pruned
3	8	4	50%
4	1024	998	97.46%
5	33554432	33554196	99.9993 %

Table 4.4: Views Sets Pruned by *NOGI*.

GVGI Algorithm

Algorithm *GVGI* greedily chooses both the view and index sets. It begins by materializing only the lower views and the primary view. That is, the initial set of supporting views and indices $\mathcal{M}$ is empty. For each supporting view $V \notin \mathcal{M}$, *GVGI* evaluates the reduction in the update cost $\mathcal{C}$ if V and a set of indices $\mathcal{I}$ on V is added to $\mathcal{M}$. The set of indices $\mathcal{I}$ is chosen by *GI*. *GVGI* then adds to $\mathcal{M}$ the view V plus the indices $\mathcal{I}$ that attains the maximum reduction in $\mathcal{C}$. If no such view V (with indices $\mathcal{I}$) reduces the update cost, *GVGI* stops. This algorithm pares down the view search space to $O(2^n)$ view sets. In practice, it is one of the fastest algorithms and our performance studies show that it is also one of the best.

VSIS Algorithm

Unlike the previous algorithms, the *VSIS* algorithm considers view and index selection separately. That is, *VSIS* first greedily chooses the view set that attains the maximum reduction in the update cost $\mathcal{C}$. When *VSIS* examines a view set $\mathcal{V}$, it assumes that there are indices on every attribute of each view in $\mathcal{V}$ but ignores their update cost. After the view set is chosen, *VSIS* uses *GI* to greedily select the indices. This algorithm pares down the view search space to $\mathcal{O}(2^n)$ view sets.

4.6.2 Performance Results

Since the various heuristic algorithms significantly pare down both the view and index search space, we are now able to perform experiments on more complex view schemas. In particular, a new primary view schema, denoted Schema 3, is a linear join of four lower views as depicted in the first 6 rows in Table 4.5. (Schema 3 extends Schema 1 which was a linear join of three lower views.) We also use a primary view schema, denoted Schema 4, that is a *star join* of four lower views. Lower view R in Schema 4 is a *fact table* and the other lower views are *dimension tables*. Details of Schema 4 are also shown in the last

Relations	$T(V)$	$I(V)$	$D(V)$	Select	Join
$R(\underline{R0}, R1)$	90M	1%	0.01%		$S(R,\ R1 = SO) = 3$
$S(\underline{S0}, S1)$	30M	1%	0.01%		$S(S,\ S0 = R1) = 1$
					$S(S,\ S1 = T0) = 3$
$T(\underline{T0}, T1, T2)$	10M	1%	0.01%	$S(T,\ T2 <= 20)$	$S(T,\ T0 = S1) = 1$
				$= 2M$	$S(T,\ T1 = U0) = 3$
$U(\underline{U0}, U1)$	3.3M	1%	0.01%		$S(U,\ U0 = T1) = 1$
$R(\underline{R0}, R1, R2, R3)$	90M	1%	0.01%		$S(R,\ R1 = S0) = 90$
					$S(R,\ R2 = T0) = 90$
					$S(R,\ R3 = U0) = 90$
$S(\underline{S0}, S1)$	10M	1%	0.01%		$S(S,\ S0 = R1) = 1$
$T(\underline{T0}, T1, T2)$	10M	1%	0.01%	$S(T,\ T2 <= 20) =$	$S(T,\ T0 = R2) = 1$
$U(\underline{U0}, U1)$	10M	1%	0.01%	$= 2M$	$S(S,\ U0 = R3) = 1$

Table 4.5: Complex View Schemas.

6 rows of Table 4.5. We compared the heuristic algorithms using Schema 3 and Schema 4 under various insertion and deletion rates. We now discuss two representative experiments.

In the first experiment, we ran the five heuristic algorithms using Schema 4. The insertion rates were varied from 0.1% to 1.0%. That is, for each lower view V involved in the primary view, the number of insertions to V was varied from $0.001 \cdot T(V)$ to $0.01 \cdot T(V)$. The deletion rates were set to be 1/100 of the insertion rates, and the update rates were set at 0. Figure 4.12 plots the ratio $\mathcal{C}/\mathcal{C}_{GVGI}$ as the insertion rates are varied. In this graph, $\mathcal{C}$ ($\mathcal{C}_{GVGI}$) is the update cost incurred by the view and index set chosen by an algorithm (respectively, *GVGI*). As shown in Figure 4.12, *GVGI*, *VSIS* and *NOGI* perform equally well. All of the three algorithms chose to materialize the supporting view set $\{\sigma_{T2\leq 20}T, R \bowtie \sigma_{T2\leq 20}T\}$ in addition to the base relations and the primary view. All three algorithms chose to build indices on the keys $R0$, SO, TO, and $U0$ and on the join attributes of the fact table R (*i.e.*, $R1$, $R2$, $R3$). On the other hand, *RETE* chose to materialize $\{\sigma_{T2\leq 20}T, R \bowtie \sigma_{T2\leq 20}T, R \bowtie \sigma_{T2\leq 20}T \bowtie U\}$. As shown in Figure 4.12, this choice results in an update cost that is 15% to 20% higher than the update cost incurred by the view and index sets chosen by the *GVGI* algorithm. This is primarily because the update cost of $R \bowtie \sigma_{T2\leq 20}T \bowtie U$ outweighs its benefits. Finally, *A-TREAT* chose to materialize $\{\sigma_{T2\leq 20}T\}$ which results in an update cost that is 42% to 53% higher than the update cost attained by *GVGI*'s choices. This result illustrates that materializing supporting views that join a

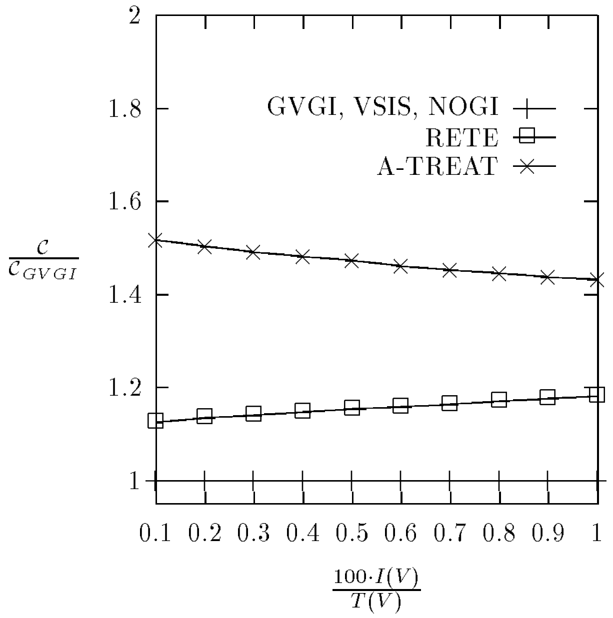

Figure 4.12: Star Join with Low Update Rate.

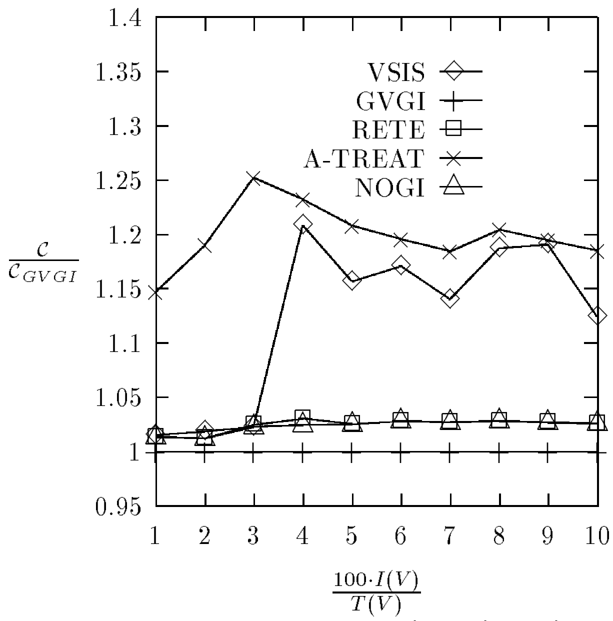

Figure 4.13: Linear Join with High Update Rate.

number of lower views can be very beneficial.

In the next experiment, we ran the five heuristic algorithms using Schema 3. The insertion rates were varied from 1% to 10%. The deletion rates were again set to be 1/100 of the insertion rates, and the update rates were set at 0. Figure 4.13 plots the ratio $\mathcal{C}/\mathcal{C}_{GVGI}$ as the insertion rates are varied. As shown in the figure, the view and index sets chosen by *GVGI* incur the least update cost. That is, the view and index sets chosen by *A-TREAT* result in an update cost that is 15% to 27% higher than the update cost attained by *GVGI*'s choices. The sets chosen by *RETE* incur an update cost that is 2% to 22% higher than the cost attained by *GVGI*'s choices. *NOGI* and *VSIS* perform just slightly worse than *GVGI*. Note that because the number of deletions is so small and the number of updates is zero, it is slightly more beneficial in our experiment to consider materialized view sets that include overlapping views (*GVGI*) than to exclude them (*NOGI*). These two experiments are representative of the other experiments performed wherein *GVGI* always chose view and index sets that incur the least update cost.

Summary

In terms of running time, *A-TREAT* is the fastest while *VSIS* is the slowest. The other three algorithms are only slightly slower than *A-TREAT*. In summary, we recommend the algorithm *GVGI* since it chooses the best view and index sets in our experiments, and it does so fairly efficiently.

4.7 Additional Experiments

So far, we have assumed that there are no space constraints in the VIS problem. That is, a set of views and indices is chosen without regard to how much space is occupied by the views and indices in the chosen set. We have also assumed that it is better to propagate (protected) updates separately from insertions and deletions. Finally, we have assumed that the WHA's estimates of the system parameters (*e.g.*, insertion rates) are exactly right. In this section we attempt to answer the following questions related to our three assumptions:

- Are views or indices better to materialize when space is constrained?
- Is there really a benefit in propagating updates separately?
- How sensitive is the VIS problem solution to the WHA's estimates of the system parameters?

We present the results of only one or two representative experiments for each question, although many more were performed. The experiments shown in this section were all run on Schema 1 (described in Section 4.5.1) using our A* based optimal algorithm. Although this schema is composed of only 3 relations, we believe our results to be more general because we have explored a number of larger schema using our heuristic algorithms and the results so far support those reported here.

4.7.1 Are Views or Indices Better When Space is Constrained?

Up to now we have shown how to find the optimal set of supporting views and indices to materialize without regard to storage space. Sometimes, however, the amount of additional storage required is prohibitive. In these cases one may ask how much additional storage is necessary to attain the majority of the performance gains and which structures should be materialized. We consider these questions for Schema 1 under two different update loads. In both experiments, we gradually increase the available storage from that required to materialize the primary view (RST) to that required by the optimal solution for the unconstrained problem. For generality, we measure the additional space as a fraction of the space required to store the base relations. At each point we find the best solution that fits in the available storage. The cost of this solution relative to the non-constrained optimum is plotted on the y-axis.

The results of the experiments are shown in Figures 4.14 and 4.15. As the graphs indicate, the schemas evolve in discrete steps - only changing when enough storage becomes available to add a new index or materialized view. The number of steps in the progression is too large (52 in Figure 4.14 and 25 in Figure 4.15) to show every schema change but the results are summarized in Figure 4.16. The numbers next to features indicate in what order they are added as storage increases. Using Figure 4.16 (a) as an example, the experiment starts with only the base relations and primary view materialized – they are numbered 0. The next features to be added are indices on the keys of the base relations present in the view RST, starting with $T0$ and then adding $S0$ and $R0$. Next, the selection node T' is materialized and an index built on its attribute $T'0$. The reason that it takes 52 steps to add all 10 numbered feature sets is that a new feature is often added at the expense of an older one. For instance, when the view T' is materialized, the index on $R0$ in RST is dropped until enough space is available to add it again. The graphs in Figures 4.14 and 4.15 are also annotated with the feature numbers to help indicate which features most impact the update performance.

The first important point to note from this experiment is that under both update loads, a large portion of the total update savings can be achieved with a reasonably small amount of additional storage. Note the large drop in I/Os for the high-update experiment that results from materializing view T' (feature 3) and then adding indices on $T0$ and $S0$ again (they were dropped earlier to make space for T'). The next large drop occurs after enough space is found to materialize ST (feature 5). By the time point A (which corresponds to features 1,2 and 5) is reached, the update cost is within 5% of the optimal cost. This result is encouraging for warehouses that have space constraints. It should be noted that even though the extra storage required for the views and indices does not seem that large compared to the warehouse relation sizes ($\approx$ 25%), there will typically be many views defined over the same relations so the total storage required by views and indices can be larger than that of warehouse relation when the warehouse is considered in its entirety.

It is interesting to see how the two images of Figure 4.16 are supported by our rules of thumb. Because of the selection condition on T, the materialized view T' is much smaller than T. Therefore, by Rule 4.5.1 view T' should be materialized. Finally, note that view ST is not materialized until near the end. Even though the number of pages in ST is less than the sum of the pages in S and T and should be materialized by Rule 4.5.1, ST is a relatively large structure to materialize in comparison to the indices. Therefore, we find that the maintenance cost is minimized overall in this case by materializing several small beneficial structures (*i.e.*, indices) than by materializing one large one (*i.e.*, view ST). It isn't until the most useful indices have already been materialized that view ST is chosen for materialization.

4.7.2 The Importance of Propagating Updates Separately

Previous work in physical database design has rarely considered updates separately, opting instead to treat them as pairs of deletions and insertions. However, [GJM96, BCL89] show that if updates do not change the values of attributes that are involved in selection or join conditions, then they can be applied to the view directly so long as the view includes a key of the updated base relation. We have called updates of this class protected updates and have applied them to the view directly instead of splitting them into insertions and deletions. The benefit of supporting protected updates is illustrated in Figure 4.17. In this experiment, protected updates were fed to the optimizer as insert/delete pairs. The

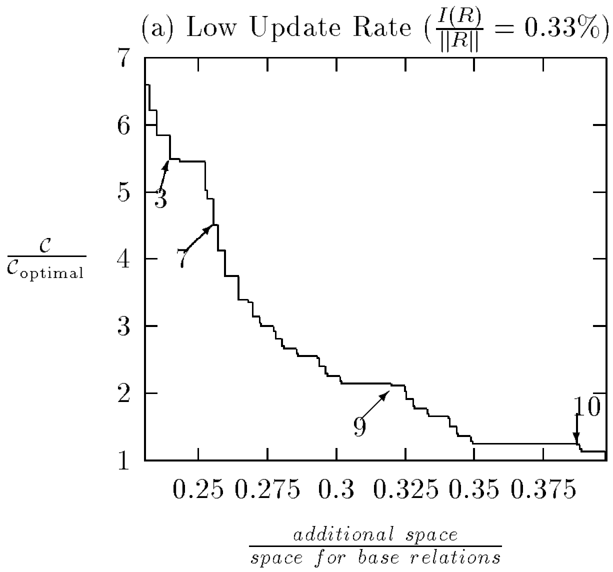

Figure 4.14: Effects of Space on Update Cost (Low Update Rate).

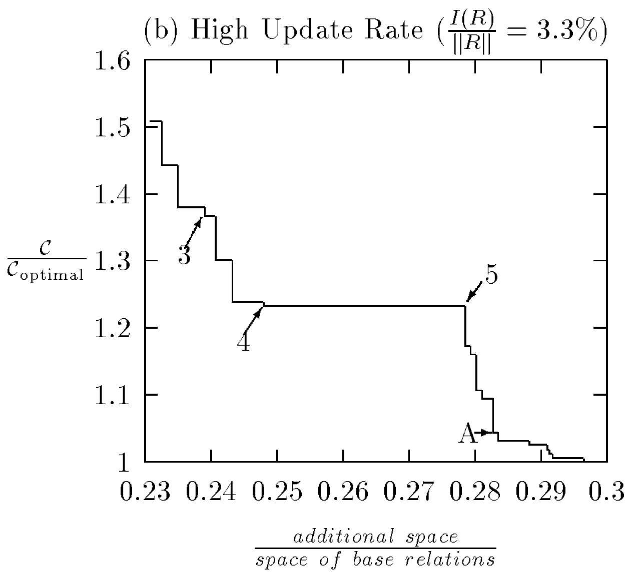

Figure 4.15: Effects of Space on Update Cost (High Update Rate).

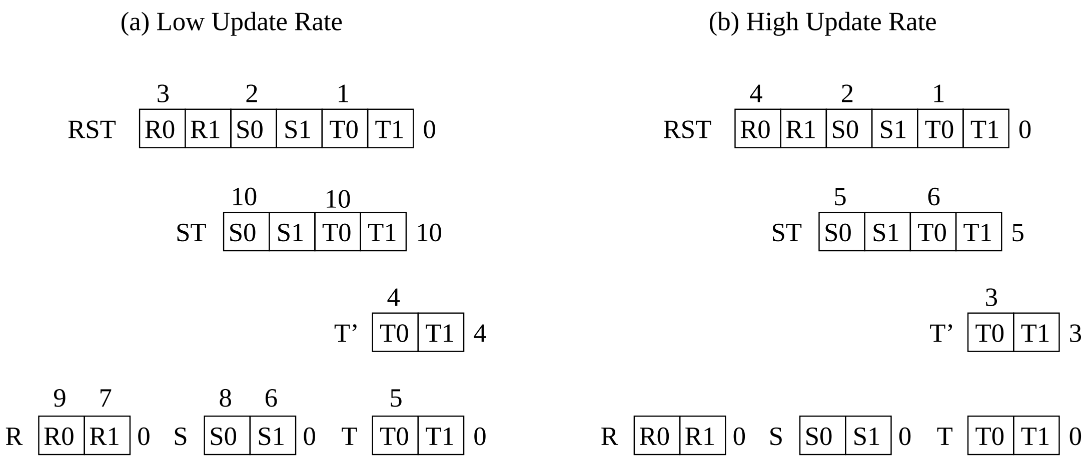

Figure 4.16: Evolution of the Physical Design.

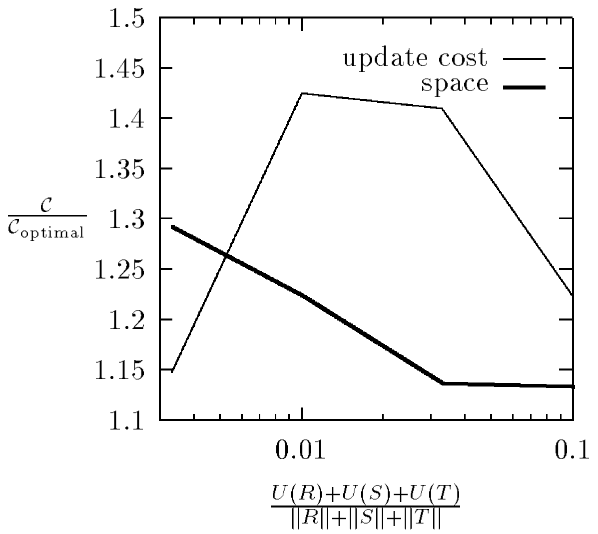

Figure 4.17: Effects of Simulating Updates with Insert/Delete.

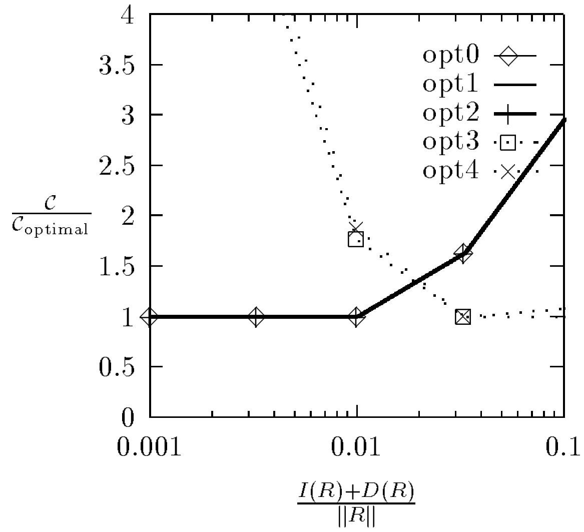

Figure 4.18: Sensitivity of Optimal Solutions to Insert/Delete Rates.

resulting view and index sets chosen were then compared to the optimal choices if the updates are propagated directly. The results, shown in Figure 4.17, show that simulating protected updates as deletion followed by insertion leads to solutions that require both more space and more maintenance time than the optimum!

4.7.3 Sensitivity Analysis

So far, this chapter has focused on finding a solution to the VIS problem. Just how well this solution works on the actual warehouse depends on how closely the input parameters, such as relation sizes and delta rates, match the real values of the system.[2] An important question for the WHA, then, is just how sensitive the optimizer is to the estimates of the input parameters. Clearly, one would hope that the solution obtained given the WHA's parameter estimates is at least a reasonably good solution for systems with only slightly different parameter values. In this section, we investigate just how badly optimal solutions decay at neighboring points. To simplify the analysis, we consider only the estimate of

[2]It also depends on how closely the VIS optimizer's cost model follows that of the dbms. This concept is discussed in [FST88].

insertion and deletion rates.

In this experiment, we varied the combined insertion and deletion rates to each base relation such that the ratio $\frac{I(R)+D(R)}{T(R)} = \frac{I(S)+D(S)}{T(S)} = \frac{I(T)+D(T)}{T(T)}$ increased from 0.001 to 0.1 in five steps. At each step, we found the optimal solution and then plotted its performance over the entire range. The results, which are shown in Figure 4.18, suggest that except for a small region in the middle of the graph, the choice of optimum in not sensitive to the combined insertion-deletion rate. For instance, the optimal solution for an estimated ratio of 0.001 is still optimal even when the ratio grows to 0.01. The only area where the optimizer seems sensitive is in the range shown in the middle of the graph where an order of magnitude error in estimation can lead to a three-fold performance hit or worse. This sensitive region corresponds to the point when the insertion-deletion rate to the base relations becomes large enough that it is no longer worthwhile to build indices on their attributes.

This experiment is typical of many sensitivity analyses that we have performed. The optimal solutions perform well across a wide range of parameter values except for a few small regions that correspond to major schema changes. This result is reassuring. One must be careful, however, in over-generalizing this result. It is likely that in schemas with more relations there will be more frequent shifts in the optimal schema. Whether these shifts will result in large differences in the maintenance cost is a subject for future research.

4.8 Related Work

Previous work related to this chapter falls into two categories, depending on the context in which it was written: physical database design and rule condition maintenance.

4.8.1 Physical Database Design

Three costs must be balanced in physical database design for warehouses: (1) the cost of answering queries using warehouse relations and additional structures, (2) the cost of maintaining additional structures, and (3) the cost of secondary storage. We have assumed that the primary view is materialized, which minimizes the cost of (1), and focused on choosing supporting view and indices such that the cost of (2) is minimized. We have also considered how constraining cost (3) affects our results.

This problem was first studied by Roussopoloulos [Rou82]. The additional structures considered for materialization are view indices, rather than the views themselves, to save on storage. A view index is similar to a materialized view except that instead of storing the tuples in the view directly, each tuple in the view index consists of pointers to (or equivalently, tuple id's of) the tuples in the lower views that derive the view tuple. No other type of index are considered. (In this chapter we choose to maintain the actual views since the cost of secondary storage is now much lower and no commercial database supports view indices.)

The Roussopoloulos paper presents an elegant algorithm based on A* and the approximate knapsack problem to find an optimal solution to the view selection problem. The algorithm, however, works because of two simplifying assumptions. First, it uses a very simple cost model for updating a view: the cost is proportional to the size of the view. But when views are incrementally maintained, the cost of maintenance is proportional not only to the size of the view but also to the sizes of the changes, the lower views, and subviews. We have shown in Section 4.2 that the cost of maintenance is a complex query optimization problem and cannot be estimated without knowing which subviews are materialized. Second, the Roussopoloulos algorithm does not consider index selection (other than view indices). We have shown in Section 4.7.1 that index selection has a significant impact on choosing which subviews to materialize, since the proper indices can make a materialized subview less costly to maintain. Relaxing either of the above two assumptions invalidates the use of the Roussopoloulos algorithm. Still, this approach is a good first treatment of the subject and the author presents experimental results for the algorithm.

Ross et al. [RSS96] considers the same problem but focuses on the view selection part of the VIS problem. They describe an exhaustive search algorithm to solve the view selection problem. They also propose heuristics for pruning the space to search. We have extended their work by considering indices, developing rules of thumb for choosing supporting views using cost model analysis, and presenting an improved optimal algorithm. We have also implemented our algorithm and used it generate experimental results that support the rules of thumb as well as answer questions such as whether to materialize indices or views when space is constrained. We have also developed new heuristic algorithms and compared them against previously developed algorithms.

More recently, Gupta [Gup97] examines the problem of minimizing the combined costs of (1) answering queries and (2) maintaining the warehouse views and indexes, while ensuring that the cost of (3) secondary storage is not too high. Gupta investigates this specific problem under a theoretical framework. He develops greedy heuristic algorithms and proves that the solutions obtained by the algorithms perform no worse than $k\%$ compared to the optimal solution (for some constant k). However, the proof only works when indices are not considered and under some assumptions on the delta rates. Since [Gup97] is a theoretical study, no experiments are performed.

Other recent work have focused on the same problem that Gupta investigated. All of them focus on either the view selection exclusively or the index selection exclusively. [YKL97] focuses on view selection; [BPT97] focuses on view selection in a multidimensional database; and [CN97] focuses on index selection. Furthermore, apart from [CN97], none of the recent work performed extensive experimental studies.

Other work has looked at the initial problem of choosing a set of primary views such that the cost of (1) answering queries is minimized, while ensuring that the costs of (2) maintaining warehouse views and indexes and (3) secondary storage are not too high. [SP89] considers this problem in the case of distributed views. [HRU96] has investigated this problem for the case of aggregate views. Tsatalos et al. [TSI94] consider materializing views in place of the lower views in order to improve query response time. Rozen et al. [RS91] look at this problem as adding a set of "features" to the database.

In particular, the index selection part of our VIS problem has been well-studied [FST88, CBC93] in the context of physical database design. Choosing indices for materialized views is a straightforward extension. What is troublesome, however, is that the previous algorithms require the queries (and their frequencies) on each lower view as inputs. This information is used in pruning the search space of indices to consider. In the VIS problem there are no user generated queries on the base relations or supporting views since they are all handled by the primary views: The only queries on lower views or supporting views are generated by maintenance expressions. Unfortunately, the set of generated queries depends on the update paths chosen for each type of delta. Recall from the example that if a view ST exists, the maintenance expression $\triangle R \bowtie S \bowtie T$ could be answered either from the lower views or as $\triangle R \bowtie ST$. The choice between the two update paths depends on whether there is an index on ST, which has not yet been determined. Thus one cannot determine in advance the query set on each lower view and supporting view without knowing which indices are present, which makes the algorithms proposed in previous work unusable here.

4.8.2 Rule Condition Maintenance

Previous work on active database and production systems also relates to the VIS problem we have described. Many authors have considered how to evaluate trigger conditions for rules. This can be considered a view maintenance problem where a rule is triggered whenever the view that satisfies its condition becomes non-empty. Wang and Hanson [WH92] study how the production system algorithms Rete [For82] and TREAT [Mir87] perform in a database environment. An extension to TREAT called A-TREAT is considered in [Han92]. Fabret et al. [FRS93] took an approach similar to ours by considering how to choose supporting views for the trigger condition view. Translated into our terminology, the rule of thumb they developed is essentially to materialize a supporting view when it is *self-maintainable*; *i.e.*, when it can be maintained for the expected changes to the lower views by referencing the changes and the view itself, but without referencing any lower views. For example, given a primary view $V^P = RST$ with only deletions (no insertions) expected to lower views S and T, then supporting view view $V^S = ST$ is a self-maintainable view. We have found through the results of our experimentation that for our environment almost the opposite is true. We have found (Rule 4.5.2) it to be worthwhile to materialize a view when no deletions (insertions are fine) are expected to the lower views involved in the view, since the work of propagating insertions can be reused in maintaining the top-level view.

Segev et al. [SF91, SZ91] consider a similar problem in expert systems. They also assume small deltas and ubiquitous indices. They do not, however, consider maintaining subviews of the primary view, but instead describe *join pattern indices*, which are specialized structures for maintaining materialized views. Join pattern indices are an interesting approach, but require specialized algorithms to maintain. They cannot be maintained with SQL data manipulation statements, which is necessary for our approach because we want the WHA to be able to choose a set of supporting views and indices and maintain them without having to write specialized code.

A major difference between all of these studies and this one is that they consider a rule environment where changes in the underlying data are propagated immediately to the view. Hence, the size of the deltas sets are relatively small, which means that index joins will usually be much cheaper than nested-block joins. They therefore assume that indices exist on all attributes involved in selection and join conditions. However, in the data warehousing environment studied here, a large number of changes are propagated at once, and the cost of maintaining the indices often outweighs any benefit obtained by doing index joins, so it is not correct to assume that indices exist on all attributes involved in selection and join conditions.

4.9 Chapter Summary

This chapter considered the VIS problem, which is one aspect of choosing good physical designs for relational databases used as data warehouses. We described and implemented an optimal algorithm based on A* that vastly prunes the search space compared to previously proposed algorithms [RSS96]. Since even the A* algorithm is impractical for many real-world problems, we developed "rules of thumb" for view selection. These rules were validated through both analysis and experimental results. We also developed heuristic algorithms that are vastly more efficient than the A* based optimal algorithm. Furthermore, we showed that the quality of the solution obtained by some of the heuristic algorithms is comparable to that of the optimal solution.

By running experiments with the optimal algorithm, we studied how space can be best used when it is constrained: whether for materializing indices or supporting views. Our results indicate that building indices on key attributes in the primary view leads to solid maintenance cost savings with modest storage requirements. We also showed that treating all updates as pairs of deletions and insertions, as has often been proposed in the literature, can lead to solutions that have larger maintenance costs *and* storage costs than those chosen when protected updates are handled explicitly.

Note that the cost An optimal algorithm must minimize the total cost of maintaining the warehouse. The total cost that we attempt to minimize is the sum of the costs of: (1) computing the changes to the primary view, (2) installing the changes of the primary view, lower views and any supporting view, and (3) modifying affected indices. The cost of maintaining one view differs depending upon what other views are available. It is therefore incorrect to calculate the cost of maintaining the original view and each of the additional views in isolation. Moreover, in order to optimize the total cost it is necessary to consider view selection and index selection together. For example, if view selection is performed separately from index selection, it is not hard to concoct cases wherein a supporting view V is considered to be too expensive to maintain without indices. However, with indices V becomes part of the optimal solution since it may become feasible to maintain V once the proper indices are built.

Chapter 5

Expiring Warehouse Data

5.1 Introduction

The previous three chapters focused on the problem of maintaining the warehouse efficiently. In this chapter, we focus on a problem that is equally important – the problem of reducing the storage cost of the materialized views in the data warehouse. The traditional way of removing data from materialized views is deletion. When tuples are deleted from a view or a source relation, the effect must be propagated to all "higher-level" views defined on the view/relation undergoing the deletion. However, the desired semantics are different when the data is removed due to space constraints alone, where it is preferable not to affect the higher-level views if possible. In this chapter, we propose a framework that gives us the option to gracefully *expire* data, so that higher-level views remain unaffected by lower-level expired data, and can be maintained consistently with respect to future changes. The difference between deletion and expiration is illustrated further in the next example.

EXAMPLE 5.1.1 Suppose the following base views are copies of source relations external to the warehouse. These base views will be used as a running example in this chapter.

- *Customer*$(custID, info)$ contains information about each customer identified by the key $custID$. For conciseness, we shall refer to $Customer$ as C.
- *Order*$(ordID, custID, clerk)$, denoted O, contains for each order, the customer who requested the order and the clerk who processed the order.

O	$ordID$	$custID$	$clerk$
	1	456	`Clerk1`
	3	789	`Clerk2`

L	$partID$	$ordID$	qty	$cost$
l_1:	`a`	1	1	19.99
l_2:	`b`	1	2	250.00
l_3:	`c`	3	1	500.00

V	$partID$	qty	$cost$	$custID$	$clerk$
v_1:	`b`	2	250.00	456	`Clerk1`
v_2:	`c`	1	500.00	789	`Clerk2`

Figure 5.1: Current state of O, L, and V.

- $Lineitem(partID, ordID, qty, cost)$, denoted L, details the quantity of the parts and the unit cost of each part requested in each order.

Consider a simple derived view V storing order information for expensive parts. V is defined as a natural join of O and L, with the selection condition $L.cost > 99$, followed by a projection onto relevant attributes. The current state of O, L, and V is depicted in Figure 5.1.

In reality, base views O and L (often called fact tables) can become quite large. Suppose that the warehouse administrator decides to *delete* "old" L tuples with $ordID < 2$. Thus, l_1 and l_2 are deleted, as if they never existed in L. As a result, v_1 is deleted from V, which might not be desirable if users still expect V to reflect information about old tuples (especially if queries over the view ask for summary data).

The method we propose instead is to *expire* L tuples with $ordID < 2$. Tuple l_1 can be safely removed from L because $l_1.cost$ is less than 99. On the other hand, l_2 must be retained because it might be needed to correctly update V if another tuple with $ordID = 1$ is inserted into O. Notice that V remains unaffected by the expiration of L tuples. Furthermore, after the expiration, there is still enough information in L to maintain V with respect to future updates.

If we know the types of modifications that may take place in the future, we may even be able to remove tuples like l_2. For example, suppose both O and L are "append-only." That is, the source relations (that O and L are based on) never delete tuples. Moreover, an insertion to O always has an $ordID$ greater than the current maximum $ordID$ in O; insertions to L always refer to the most recent order, *i.e.*, the O tuple with the maximum $ordID$. In this case, we can expire both l_1 and l_2 since they will never be needed to maintain V. In fact, it is possible to expire the entire L and O views except for the tuple recording the most recent order. In our framework, one can define applications constraints, such as "append-only," using a general constraint language, so that the system can remove as much data as possible when the warehouse administrator so wishes it. □

Although expired tuples are *physically* removed from the extension of a view, they still exist *logically* from the perspective of the higher-level views. Our expiration scheme guarantees that expiration never results in incomplete or incorrect answers for the maintenance expressions that are used to maintain the higher-level views, given any possible source updates. Knowledge of constraints on these updates can further improve the effectiveness of expiration. User queries may, however, request data that has been expired. In such cases an incomplete answer must be provided, with an appropriate warning that describes which of the requested data was actually available.

Unfortunately, current warehouse products provide very little support for gracefully expiring data. Every time there is a need to expire data, it is up to the administrator to *manually* examine view definitions and maintenance expressions and to check if underlying data is needed for maintenance. This "solution" is clearly problematic since not only is it inefficient, but it is prone to human error which can easily lead to the expiration of needed data. Furthermore, deciding what is needed and what can be expired is complicated by the presence of constraints. If a conservative approach is used (*e.g.*, constraints are not taken into account), then the storage requirement of the warehouse may become prohibitively large.

In this chapter we propose a framework wherein expiration of data is managed, not manually, but by the system. In particular:

- The administrator or users can declaratively request to expire part of a view, and the system automatically expires as much unneeded data as possible.
- The administrator can declare in a general way constraints that apply to the application data as well as changes to the data (*e.g.*, base view O is append-only), and the system uses this knowledge to increase the amount of data that may be expired.
- The administrator or users can change framework parameters (*e.g.*, by defining additional views or changing application constraints) dynamically, and the system determines the effects of these changes on what data is deemed needed and what data can be expired.

For this framework we develop efficient algorithms that check what data can be expired, handle insertions of new data, and manage changes to views and constraints. We also illustrate, using the TPC-D benchmark [Com], the benefits of incorporating constraints into the management of expired data.

The rest of the chapter proceeds as follows. In Section 5.2, we introduce our expiration framework and identify problems that need to be solved. The central problem of identifying the needed tuples is solved in Section 5.3, while Section 5.4 extends the mechanism to take constraints into account. We illustrate in Section 5.5 that the "constraint-aware" solution can lead to much more data being expired. In Section 5.6, we develop algorithms that handle changes to the framework parameters. We discuss related work in Section 5.7 and conclude the chapter in Section 5.8.

5.2 Framework

In this section, we present our framework for expiration. We then give an overview of the problems that we address in the rest of the chapter to implement the framework.

Views and Queries

As usual, we consider two types of warehouse views: base views and derived views. Each base view (*e.g.*, *Order*) has an *extension* that stores persistently the answer to its *view definition*,*Def*(V), which is of the form $\pi_{\mathcal{A}}\sigma_{\mathcal{P}}(\times_{R\in\mathcal{R}}R)$, (We assume that the π, σ, $\times$ operators have bag semantics.) This form of a view definition can express base view definitions based on SQL `SELECT-FROM- WHERE` clauses, which is consistent with the assumptions in the previous chapters. Each derived view V has an extension that stores the answer to its view definition, also denoted *Def*(V), which is of the form $\pi_{\mathcal{A}}\sigma_{\mathcal{P}}(\times_{R\in\mathcal{R}}R)$. However, the π operator used in the view definition of a derived view is the *generalized projection* operator (introduced in [GHQ95]) that can perform aggregations. This form of a view definition can express view definitions based on SQL `SELECT-FROM- WHERE-GROUP BY` clauses (without subqueries) as we illustrate next.

For instance, we can define a view $ClerkCust$ to obtain the sum of the purchases made by a customer from some clerk. Furthermore, $ClerkCust$ only considers old customers that placed an order recently for an expensive item. The view definition of $ClerkCust$ is as follows.

$$\pi_{O.clerk, C.custID, \mathtt{SUM}(L.qty * L.cost) \text{ AS } sum, \mathtt{COUNT}() \text{ AS } cnt}$$
$$\sigma_{L.cost>99 \wedge C.custID<500 \wedge O.ordID>1000 \wedge L.ordID=O.ordID \wedge O.custID=C.custID} (C \times O \times L).$$

In general, the projection list $\mathcal{A}$ of a view definition is a set of attributes and aggregate functions (*e.g.*, `SUM`). If $\mathcal{A}$ contains aggregate functions, any element in $\mathcal{A}$ that is not an aggregate function is a grouping attribute (*e.g.*, $C.custID$). Condition $\mathcal{P}$ is a conjunction of atomic conditions, like join condition $O.custID = C.custID$, and selection condition $O.ordID > 1000$. Finally, $\mathcal{R}$ is a set of views (*i.e.*, self-joins are not considered).

A view V needs to be maintained when there are insertions, deletions and updates to the views that V is defined on. Although the algorithms we develop in this chapter can handle updates, our example queries/expressions will not show updates to simplify the queries/expressions. To illustrate how changes are computed, let us assume that *Def*(V) is $\sigma_{S.b=T.c}(S \times T)$. To compute the insertions to V (*i.e.*, $\triangle V$), the *maintenance expression* given by Query (5.1) below is used. The deletions to V (*i.e.*, $\triangledown V$) are computed using Query (5.2). These queries use the *pre-state* of S and T, *i.e.*, before the insertions, and then the deletions, are applied. (Dual-stage view strategies in Chapter 3 use the pre-state of the views as well.) We use *Maint*(V) to denote the set of maintenance expressions for computing the insertions to and deletions from V.

$$\begin{aligned} &\sigma_{\triangle S.b=T.c}(\triangle S \times T) \;\cup\; \sigma_{S.b=\triangle T.c}(S \times \triangle T) \;\cup \\ &\sigma_{\triangle S.b=\triangle T.c}(\triangle S \times \triangle T) \;\cup\; \sigma_{\triangledown S.b=\triangledown T.c}(\triangledown S \times \triangledown T) \end{aligned} \tag{5.1}$$

$$\begin{aligned} &\sigma_{\triangledown S.b=T.c}(\triangledown S \times T) \;\cup\; \sigma_{S.b=\triangledown T.c}(S \times \triangledown T) \;\cup \\ &\sigma_{\triangledown S.b=\triangle T.c}(\triangledown S \times \triangle T) \;\cup\; \sigma_{\triangle S.b=\triangledown T.c}(\triangle S \times \triangledown T) \end{aligned} \tag{5.2}$$

Expiration

A user may issue an *expiration request* of the form $\sigma_{\mathcal{P}}(T)$ on any view T. This request asks that all the T tuples in $\sigma_{\mathcal{P}}(T)$ be removed from T's extension. Once a tuple is expired, it can no longer be accessed by any query. However, in our framework, we only expire $\sigma_{\mathcal{P}}(T)$ tuples that are not "needed" (later defined formally) by maintenance expressions. Conceptually, we partition the extension of each view T into T^+, T^-, and T^{exp}, as shown in Figure 5.2. The tuples in T^+ are accessible to any query and are needed by maintenance expressions. The tuples in T^- are accessible to any query but are not needed by maintenance expressions. The tuples in T^{exp} are expired, are not accessible, and are not needed by maintenance expressions. The tuples in T^+ and T^- comprise T's *real extension*, which is the extension kept persistently. The tuples in T^+, T^-, and T^{exp} comprise T's *full extension.* (The full extension of T is referred to in queries simply as "T".) The conceptual partitions T^+ and T^- are realized in T's real extension by keeping a boolean attribute *needed* for each tuple. The *needed* attribute of a tuple t is set to `true` if $t \in T^+$ and `false` otherwise. Given an expiration request $\sigma_{\mathcal{P}}(T)$, conceptually the request is satisfied by removing $\sigma_{\mathcal{P}}(T^-)$ from T^- and "moving" them to T^{exp}, as depicted in Figure 5.3. We assume that for any two consecutive expiration requests on T, denoted $\sigma_{\mathcal{P}_i}(T)$ and $\sigma_{\mathcal{P}_j}(T)$, the subsequent request asks for more tuples to be expired than the earlier one (*i.e.*, $\mathcal{P}_i$ implies $\mathcal{P}_j$). This requirement is satisfied by keeping the most recent expiration request on T in *LastReq*$(T) = \sigma_{\mathcal{P}'}(T)$. When a new expiration request $\sigma_{\mathcal{P}}(T)$ is issued, the request is modified as $\sigma_{\mathcal{P} \vee \mathcal{P}'}(T)$ and *LastReq*(T) is set to $\sigma_{\mathcal{P} \vee \mathcal{P}'}(T)$.[1]

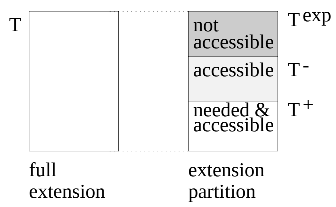

Figure 5.2: Extension Partition of T

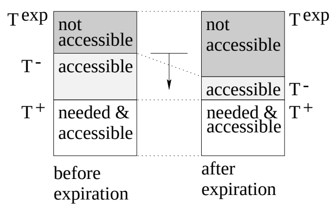

Figure 5.3: Effect of Expiration on T^- and T^{exp}

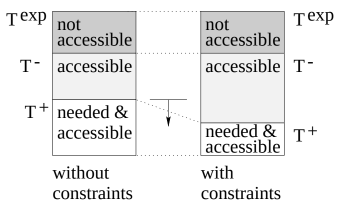

Figure 5.4: Effect of Constraints on T^+ and T^-

Effect of Expiration on Queries

Although all queries (user queries, maintenance expressions and view definitions) are formulated in terms of full extensions, only the tuples in the real extensions can be used in answering the query. Conceptually, the answer returned for Q is the answer for the "query" *Access*(Q), which is the same as Q but with each T referred to in Q replaced by $T^+ \cup T^-$. Similarly, the complete answer to Q is the answer returned for the "query" *Complete*(Q), which is the same as Q but with each T referred to in Q replaced by $T^+ \cup T^- \cup T^{exp}$ (*i.e.*, suppose that tuples in T^{exp} are accessible to *Complete*(Q)). We say the answer to Q is *complete* if the answer to *Access*(Q) is the same as the answer to *Complete*(Q). Otherwise, the answer is *incomplete*. We say that a tuple $t \in T$ (*i.e.*, $t \in (T^+ \cup T^- \cup T^{exp})$) is *needed* in

[1] Algorithms for removing redundant conditions in $\mathcal{P} \vee \mathcal{P}'$ can certainly be employed.

answering Q if the answer to $Complete(Q)$ is different depending on whether t is removed from T's extension or not.[2]

Since we guarantee that only tuples not needed by maintenance expressions can be expired, the answer to any maintenance expression Q is always complete. On the other hand, the answer to a user query or view definition Q may be incomplete. In case of a user query, a query Q', where $Access(Q) = Complete(Q')$, is returned in addition to Q's incomplete answer. Q' is used to help describe the incomplete answer returned. Incidentally, we believe that many database systems return incomplete answers, because databases cannot hold all possible data. However, in current systems, users are simply not told about missing data. We think returning descriptive information like Q' is an improvement. In case of a view definition $Q = Def(V)$, if the answer to Q is incomplete, V is not initialized and a query Q', where $Access(Q) = Complete(Q')$, is returned as an alternative view definition for V. Note that for both user queries and view definitions, it may be possible to obtain more answer tuples by accessing not only the views referred to in the query, but also the underlying views these views are defined on. Such an extension is feasible in our framework, but it is not considered in this chapter.

Constraints

To help decrease the number of tuples that are deemed needed (see Figure 5.4), we may associate with each view T a set of constraints, $Constraints(T)$, that describe in some language (Section 5.4) the contents of the delta tables $\triangle T$ and $\bigtriangledown T$. The constraints of base views are provided by the administrator based on his knowledge of the application (*e.g.*, "view O is append-only"). The constraints of a view V are computed from the constraints of the views that V is defined on. We do *not* assume that the input constraints characterize the application completely. We only assume that the administrator inputs constraints that he knows are implied by the application. In the worst case, the administrator may not know any guarantees on the delta tables and may set $Constraints(T)$ to be empty.

Framework Summary

Table 5.1 gives a summary of the concepts used in the framework. Henceforth, we denote the set of all views as $\mathcal{T}$, the set of all constraints as $\mathcal{C}$ (*i.e.*, $\bigcup_{T \in \mathcal{T}} Constraints(T)$), and the set of all maintenance expressions as $\mathcal{E}$ (*i.e.*, $\bigcup_{\text{view } V \in \mathcal{T}} Maint(V)$).

[2] This definition of *needed* works for aggregate views since we require the `COUNT` aggregate function to be included. This requirement is reasonable because `COUNT` is helpful in maintaining views with `AVG`, `SUM`, `MAX` or `MIN` ([Qua96]).

base view T	1. real extension ($T^+ \cup T^-$); 2. full extension ($T^+ \cup T^- \cup T^{exp}$); 3. *Constraints*($T$); 4. *Def*($T$) 5. *LastReq*($T$)
derived view T	1. real extension ($T^+ \cup T^-$); 2. full extension ($T^+ \cup T^- \cup T^{exp}$); 3. *Constraints*($T$); 4. *Def*($T$); 5. *Maint*($T$); 6. *LastReq*($T$)
delta table $\triangle T$	extension (with no conceptual partitions) containing insertions to T
delta table $\triangledown T$	extension (with no conceptual partitions) containing deletions from T
expiration request $\sigma_{\mathcal{P}}(T)$	satisfied by removing $\sigma_{\mathcal{P}}(T^-)$ from T's real extension
query Q	refers to full extensions (*e.g.*, as "T") only and never partitions
user query Q	1. cannot refer to delta tables; 2. if answer is incomplete, Q' ($Access(Q) = Complete(Q')$) is returned to describe incomplete answer
view definition Q	1. cannot refer to delta tables; 2. if answer is incomplete, Q' ($Access(Q) = Complete(Q')$) is returned as alternative definition
maintenance expression Q	1. can refer to delta tables; 2. answer is always complete
$\mathcal{T}$	set of all warehouse views
$\mathcal{C}$	$\bigcup_{T \in \mathcal{T}} Constraints(T)$
$\mathcal{E}$	$\bigcup_{\text{view } V \in \mathcal{T}} Maint(V)$

Table 5.1: Summary of Framework

Problems

There are several problems that need to be solved to implement our framework:

1. **Initial Extension Marking:** Given an initial configuration of views $\mathcal{T}$ where none of the views have any expired tuples yet, we must identify and mark which tuples are needed by the maintenance expression $\mathcal{E}$ by setting the *needed* attribute of these tuples to `true`.

2. **Initial Extension Marking With Constraints:** This problem is the same as (1) but in addition, we are also given a set of constraints $\mathcal{C}$, which can potentially decrease the number of tuples whose *needed* attribute is set to `true`.

3. **Constraints of Views:** In solving the first two problems, we must compute the constraints of each view $V \in \mathcal{T}$ from the constraints of underlying views.

4. **Incomplete Answers:** For each possible user query Q, we must be able to determine if the answer to Q is complete. If not, we must determine a modified query Q' whose complete answer is the same as the incomplete answer returned for Q.

5. **Changes to $\mathcal{T}$:** When a new view V is being added to the initial configuration of views $\mathcal{T}$, we must determine if the answer to $Q = Def(V)$ is complete. Techniques for (4) apply here. If the answer to Q is not complete, we must determine a modified view definition Q' as a suggested alternative view definition. Once $Def(V)$ has a complete answer, for each view T that V is defined on, we must determine which tuples are now needed because of the addition of V, and mark these tuples appropriately.

6. **Changes To $\mathcal{C}$:** If the constraints are changed to expire more tuples, we must determine the effects of the change on the extension marking of each view T.

7. **Insertions:** If there are insertions $\triangle T$ to a view T, we must determine the *needed* attribute value of each tuple inserted. (Nothing needs to be done for deletions.)

Note that the first two problems need to be solved once, when the initial configuration is given. Hence, efficiency is not at a premium. The third, fifth and sixth problems are also solved infrequently. On the other hand, the fourth and seventh problems are solved fairly frequently and require reasonably efficient solutions. In the rest of the chapter, Section 5.3 is devoted to the first problem; Section 5.4 is devoted to the second problem; and Section 5.6 is devoted to the last three problems. Solutions to the third and fourth problems are important future work. Since we do not solve the third problem, we assume that the administrator provides not only the constraints of the base views but also the constraints of the derived views.

5.3 Extension Marking

In this section, we assume we are given an initial configuration $\mathcal{T}$ (base and derived views) and none of the views have any expired tuples yet. For each view $T \in \mathcal{T}$, we identify which T tuples are needed by maintenance expressions. We mark the needed tuples by setting the *needed* attribute.

As mentioned earlier, this marking is done only when the initial configuration is submitted and not for each expiration request. Once the marking is done, any subsequent expiration request $\sigma_{\mathcal{P}}(T)$ is satisfied very efficiently by removing the tuples $\sigma_{\mathcal{P} \wedge needed=\texttt{false}}(T)$ from T's real extension.

Before we present how the needed tuples are identified, we introduce *maintenance subexpressions*, which are the subqueries of the maintenance expressions that we work with. For instance, suppose we have a view V whose view definition is of the form $\pi_{\mathcal{A}}\sigma_{\mathcal{P}}(\times_{R\in\mathcal{R}}R)$, where $\mathcal{A}$ does not have any aggregate functions. The maintenance expressions (*e.g.*, Queries (5.1) and (5.2)) of V are of the form

$$\bigcup_i \pi_{\mathcal{A}_i}\sigma_{\mathcal{P}_i}(\times_{R\in\mathcal{R}_i}R),$$

where $\mathcal{R}_i$ may include delta tables. We call each subquery $\pi_{\mathcal{A}_i}\sigma_{\mathcal{P}_i}(\times_{R\in\mathcal{R}_i}R)$ a *maintenance subexpression*. Notice that if a tuple is needed by some maintenance subexpression, it is needed by some maintenance expression. Also, if a tuple is not needed by any maintenance subexpression, it is not needed by any maintenance expression. Later in this section, we show that the maintenance expressions of views that use aggregates can also be decomposed into maintenance subexpressions. (Note that our example view *ClerkCust* has aggregates.) Henceforth, we use $\mathcal{E}$ for the maintenance subexpressions of $\mathcal{T}$.

We now present a lemma that defines a function $\texttt{Needed}(T, \mathcal{E})$ and identifies using this function, all and only the T tuples that are needed by the maintenance subexpressions in $\mathcal{E}$. We refer to the following functions in the lemma: `Closure`, `Ignore`, and `Map`.

Function $\texttt{Closure}(\mathcal{P})$ returns the closure of the input conjunctive condition ([Ull89a]). For instance, if $\mathcal{P}$ is $R.a > S.b \wedge S.b > T.c$, $\texttt{Closure}(\mathcal{P})$ returns $R.a > S.b \wedge S.b > T.c \wedge R.a > T.c$. (`Closure` is an $O(n^3)$ operation, where n is the number of distinct attributes in $\mathcal{P}$.)

Function $\texttt{Ignore}(\mathcal{P}, \mathcal{T})$ modifies the conjunctive condition $\mathcal{P}$ by replacing any atomic condition that uses an attribute of a view in $\mathcal{T}$ with `true`. For instance, if $\mathcal{P}$ is $R.a > S.b \wedge S.b > T.c$, $\texttt{Ignore}(\mathcal{P}, \{S\})$ is $\texttt{true} \wedge \texttt{true}$ or simply `true`. Notice that $\texttt{Ignore}(\texttt{Closure}(\mathcal{P}),\{S\})$ is $R.a > T.c$.

Finally, function `Map` acts on a maintenance subexpression $E = \pi_{\mathcal{A}}\sigma_{\mathcal{P}}(\times_{R\in\mathcal{R}}R)$, where there is a set $\mathcal{D} \subseteq \mathcal{R}$ of delta tables (possibly empty) involved in E's cross product. Function $\texttt{Map}(E, T)$ is defined as follows, where E is $\pi_{\mathcal{A}}\sigma_{\mathcal{P}}(\times_{R\in\mathcal{R}}R)$.

$$\texttt{Map}(\pi_{\mathcal{A}}\sigma_{\mathcal{P}}(\times_{R\in\mathcal{R}}R), T) = \begin{cases} \{\} & \text{if } T \notin \mathcal{R} \\ \pi_{\texttt{Attrs}(T)}\sigma_{\texttt{Ignore}(\texttt{Closure}(\mathcal{P}),(\mathcal{D}-\{T\}))}(\times_{R\in(\mathcal{R}-(\mathcal{D}-\{T\}))}R) & \text{otherwise} \end{cases}$$

That is, if T is not referred to in E, `Map` returns $\{\}$. This is the common case since most maintenance subexpressions do not refer to a specific view T. If T is referred to in E, `Map` returns a new subexpression obtained by first removing the delta tables in $\mathcal{D}$ from the cross product (except T if T is a delta table). Then, the closure of the condition $\mathcal{P}$ is computed.

Then, $\mathcal{P}$ is modified to ignore any atomic condition that refers to any delta table (except T if T is a delta table). Finally, the projected attributes is changed to $\texttt{Attrs}(T)$, the attributes of the view T.

Lemma 5.3.1 *Given a view T and a set of maintenance subexpression $\mathcal{E}$,* `Needed`*(T, E) is defined as*

$$\bigcup_{E \in \mathcal{E}} \texttt{Map}(E, T).$$

The query $T \rhd\!\!< _{\texttt{Attrs}(T)} \texttt{Needed}(T, \mathcal{E})$ returns all and only the tuples in T that are needed by the maintenance subexpressions in $\mathcal{E}$. □

Note that `Needed` may list a needed tuple $t \in T$ more times than t appears in T. (This situation is illustrated in the next example.) Hence, the semijoin ($\rhd\!\!<$) operation, which is equivalent to an `exists` condition (*e.g.*, SQL EXISTS condition), is used to obtain the T tuples needed for $\mathcal{E}$. The proof of Lemma 5.3.1 is in Appendix C. We give the intuition behind the proof in the next example.

EXAMPLE 5.3.1 Let us consider one of the maintenance subexpressions of $ClerkCust$ as the maintenance subexpression E in question.

$$E = \pi_{\triangle O.clerk, C.custID, L.qty, L.cost}$$
$$\sigma_{L.cost>99 \,\wedge\, C.custID<500 \,\wedge\, \triangle O.ordID>1000 \,\wedge\, L.ordID=\triangle O.ordID \,\wedge\, \triangle O.custID=C.custID}$$
$$(C \times \triangle O \times L)$$

Let us consider what L tuples are needed by E. We claim that $\texttt{Map}(E, L)$, shown below, identifies all these L tuples.

$$\pi_{\texttt{Attrs}(L)} \sigma_{L.cost>99 \,\wedge\, C.custID<500 \,\wedge\, L.ordID>1000}(C \times L)$$

Notice that $\texttt{Map}(E, L)$ excludes $\triangle O$ from the cross product and consequently ignores all the atomic conditions in E that refer to $\triangle O$ attributes. Intuitively, this means that we cannot say that an L tuple t_L is not needed even if there does not exist a $\triangle O$ tuple that t_L can join with. This procedure is reasonable because although t_L may not join with any of the current insertions to O (*i.e.*, current extension of $\triangle O$), it may join with future insertions (*i.e.*, extension of $\triangle O$ at some later point in time). We can only set $t_L.needed$ to `false` if for *any* $\triangle O$, t_L only joins with $\triangle O$ tuples that are not needed themselves. For instance, any $\triangle O$ tuple that has an $ordID$ less than or equal to 1000 is not needed in answering E. Since there is an atomic condition $L.ordID = \triangle O.ordID$ in E, any L tuple that has an $ordID$ less than or equal to 1000 is also not needed in answering E. This illustrates the need for computing the closure of the atomic conditions before ignoring the atomic conditions that use delta table attributes. Thus, in our example, $\texttt{Map}(E, L)$ has the atomic condition $L.ordID > 1000$.

While $\texttt{Map}(E, L)$ identifies all the needed L tuples, it may list an L tuple t_L more times than t_L appears in L. For instance, $\texttt{Map}(E, L)$ performs a cross product between C and L without applying any conditions between them. Hence, $\texttt{Map}(E, L)$ lists t_L as many times as there are C tuples. Thus, to obtain the correct bag of tuples, the query $L \ltimes_{\texttt{Attrs}(L)} \texttt{Map}(E, L)$ is used. □

The example illustrated that $\texttt{Map}(E, T)$ may perform cross products. Cross products can be easily avoided by constructing a join graph for E, whose nodes represent the views in E. An edge between views R and S is in the E's join graph if there is an atomic condition in E that uses both R and S attributes. Given E's join graph, $\texttt{Map}(E, T)$ can be modified as follows. If a view R is not reachable from T, remove R from the cross product and ignore all the atomic conditions that refer to R attributes. This simple procedure can be used to avoid all cross products.

5.3.1 Aggregates

We now show that the maintenance expressions of views that use aggregates can also be decomposed into maintenance subexpressions. An aggregate view is one whose view definition uses aggregate functions. More specifically, given that $\pi_{\mathcal{A}} \sigma_{\mathcal{P}} (\times_{R \in \mathcal{R}} R)$ is the definition of a view V, V is an aggregate view if $\mathcal{A}$ has aggregate functions. In this chapter, we focus on the SQL aggregate functions: `SUM`, `COUNT`, `AVG`, `MAX`, `MIN`. Note that `AVG` can be computed using `SUM` and `COUNT`. Thus, we focus on the aggregate functions `SUM`, `COUNT`, `MAX` and `MIN` here.

It is useful to define the *plain view* of V whose definition is similar to V's but without aggregates. That is, the definition of V's plain view is $\pi_{\mathcal{A}'} \sigma_{\mathcal{P}} (\times_{R \in \mathcal{R}} R)$, where $\mathcal{A}'$ includes all the attributes referred to in $\mathcal{A}$, including the attributes referred to in the aggregate functions. We denote the plain view of V as V^{plain}.

In [Qua96], the maintenance expressions of aggregate views were determined assuming the delta tables of V^{plain} were computed beforehand. It was also shown in [Qua96] that aggregate views that do not use MAX and MIN can be incrementally maintained, while aggregate views that use MAX and MIN cannot be incrementally maintained in general.

We now discuss how the maintenance expressions of an aggregate view V that does not use MAX and MIN can be decomposed into maintenance subexpressions. In summary, the maintenance subexpressions of V are just the maintenance subexpression of V^{plain} plus the maintenance subexpression $\pi_{\texttt{Attrs}(V)}\sigma_{\texttt{true}}(V)$. We illustrate in the next example why this is so.

EXAMPLE 5.3.2 We examine the maintenance expressions of view V_{sum} defined as

$$\pi_{a,\texttt{SUM}(b)\ \texttt{AS}\ sum,\texttt{COUNT}()\ \texttt{AS}\ cnt}(V_{sum}^{plain}).$$

We assume that the insertions and deletions of V_{sum}^{plain} have been computed. To compute the deletions and insertions to V_{sum}, the following maintenance expressions are used. These maintenance expressions are derived from [Qua96].

$$\nabla V_{sum} \leftarrow V_{sum} \ltimes_a (\triangle V_{sum}^{plain} \cup \nabla V_{sum}^{plain}) \tag{5.3}$$

$$\begin{aligned}\triangle V_{sum} \leftarrow\ & \sigma_{cnt>0}\pi_{a,\texttt{SUM}(sum)\ \texttt{AS}\ sum,\texttt{SUM}(cnt)\ \texttt{AS}\ cnt} \\ & (\nabla V_{sum} \cup \pi_{a,\texttt{SUM}(b)\ \texttt{AS}\ sum,+1\ \texttt{AS}\ cnt}(\triangle V_{sum}^{plain}) \\ & \cup \pi_{a,-\texttt{SUM}(b)\ \texttt{AS}\ sum,-1\ \texttt{AS}\ cnt}(\nabla V_{sum}^{plain}))\end{aligned} \tag{5.4}$$

Notice that the maintenance expressions access $\triangle V_{sum}^{plain}$ and ∇V_{sum}^{plain}. The delta tables $\triangle V_{sum}^{plain}$ and ∇V_{sum}^{plain} are computed using the maintenance expression of V_{sum}^{plain}. Since V_{sum}^{plain} does not have aggregates, the maintenance subexpressions can be easily derived from the maintenance expressions of V_{sum}^{plain} as we showed earlier in this section.

The maintenance expressions also access V_{sum}. Hence, another maintenance subexpression of V_{sum} is $\pi_{\texttt{Attrs}(V)}\sigma_{\texttt{true}}(V_{sum})$. □

An aggregate view V that uses MAX and MIN in general cannot be incrementally maintained in the presence of deletions. Thus V needs to be recomputed from scratch. One possible maintenance expression of V is to apply the aggregate functions in the definition of V on the view definition of V^{plain}.

$$\pi_{\mathcal{A}}(\pi_{\mathcal{A}'}\sigma_{\mathcal{P}}(\times_{R\in\mathcal{R}} R))$$

The maintenance subexpression of V then is just the view definition of V^{plain}.

5.4 Extension Marking With Constraints

Given a set of views $\mathcal{T}$, maintenance subexpressions $\mathcal{E}$, and now a set of constraints $\mathcal{C}$, our goal is to mark the tuples that are needed by the maintenance subexpressions. The constraints may lead to a decrease of the number of needed tuples.

Marking tuples entails solving two problems. First, the maintenance subexpressions in $\mathcal{E}$ need to be modified using $\mathcal{C}$ to produce a new set of subexpressions $\mathcal{E}_\mathcal{C}$. Second, the function $\texttt{Needed}(T, \mathcal{E})$ needs to be modified to $\texttt{Needed}_\mathcal{C}(T, \mathcal{E}_\mathcal{C})$ that acts on the new set of maintenance subexpressions. `Needed` is not adequate because it assumes a maintenance subexpression of the form $\pi_\mathcal{A}\sigma_\mathcal{P}(\times_{R\in\mathcal{R}}R)$, which is devoid of `exists` and `not exists` conditions (expressed using the $\ltimes$ and $\overline{\ltimes}$ operators). Unfortunately, the subexpressions in $\mathcal{E}_\mathcal{C}$ may contain such conditions.

Before we solve these two problems, we present a simple constraint language *CL* for specifying the constraints in $\mathcal{C}$. In Section 5.4.2, we give the algorithm that uses $\mathcal{C}$ for producing $\mathcal{E}_\mathcal{C}$ from $\mathcal{E}$. We present in Section 5.4.3 the function $\texttt{Needed}_\mathcal{C}$ that acts on $\mathcal{E}_\mathcal{C}$. We illustrate in Section 5.5 that $\texttt{Needed}_\mathcal{C}$ may return a much smaller bag of tuples compared to `Needed`.

5.4.1 Constraint Language

A *CL* constraint is an equivalence conforming to one of the two forms shown below, where each R and T is either a base view, a delta table or a derived view.

$$\sigma_{\mathcal{P}_{LHS}}(\times_{R\in\mathcal{R}}R) \equiv \sigma_{\mathcal{P}_{RHS}}(\times_{R\in\mathcal{R}}R)\ltimes T \quad \text{or} \quad \sigma_{\mathcal{P}_{LHS}}(\times_{R\in\mathcal{R}}R) \equiv \sigma_{\mathcal{P}_{RHS}}(\times_{R\in\mathcal{R}}R)\overline{\ltimes} T$$

A *CL* constraint c states that the query on c's left hand side is guaranteed to return the same bag of tuples as the query on c's right hand side. We denote the query on the right hand side and the left hand side of a constraint c as $RHS(c)$ and $LHS(c)$, respectively. In any constraint c, the conditions in $RHS(c)$ logically imply the conditions in $LHS(c)$ (*i.e.*, $\mathcal{P}_{RHS} \Rightarrow \mathcal{P}_{LHS}$). Also, `exists` or `not exists` (*i.e.*, $\overline{\ltimes}$ operator) conditions can be introduced in $RHS(c)$. Even though $RHS(c)$ has more conditions than $LHS(c)$, constraint c states that the two queries are equivalent.

In the discussion, we often refer to a constraint c of the form $R \equiv \sigma_{\mathcal{P}_{RHS}}(R)\ltimes T$ (or $\overline{\ltimes} T$) as *context-free*, since R can be substituted by $RHS(c)$ in any query that R is in. More general constraints that have selection or join conditions on the left hand side are called *context-sensitive*.

CL can express many constraints that occur in warehousing applications. For instance, it can express equality generating dependencies (*e.g.*, functional dependencies, key constraints) and many tuple generating dependencies (*e.g.*, inclusion dependencies, referential integrity constraints). In addition to these conventional database constraints, CL can also express "semantic" constraints [SO89] such as "transition" constraints [NY82] and implication constraints. Examples of these constraints are append-only constraints and ad hoc constraints like "`Clerk1` handles `CustA`". CL cannot however express join dependencies and extending CL to handle these dependencies makes the algorithms we introduce later very complex. Even with this deficiency, we believe CL is expressive enough to capture many constraints that occur in practice as illustrated next. Furthermore, we will see that CL's syntax is particularly well suited for modifying maintenance subexpressions.

EXAMPLE 5.4.1 We give the CL constraints which an administrator may input because they are implied by the scenario in Example 5.1.1. Note that most of the constraints are context-free.

Append-only constraints

We alluded in Example 5.1.1 that O is append-only. That is, no tuple is ever deleted from O and every inserted O tuple has an $ordID$ value greater than the maximum $ordID$ value so far. The append-only behavior of O is captured by Constraint (5.5), which states that $\bigtriangledown O$ is always empty, and by Constraint (5.6), which states that the $ordID$ values of the inserted O tuples are greater than the maximum $ordID$ value so far.

$$\bigtriangledown O \equiv \sigma_{\texttt{false}}(\bigtriangledown O) \tag{5.5}$$

$$\triangle O \equiv \triangle O \overline{\ltimes}_{\triangle O.ordID \leq O.ordID} O \tag{5.6}$$

L also has an append-only behavior which is captured in Constraints (5.7), (5.8) and (5.9). Intuitively, insertions to L represent new line items of the most recent order (O tuple with maximum $ordID$) or of new incoming orders ($\triangle O$ tuples). Constraints (5.8) and (5.9) are used to describe the insertions to L. That is, inserted L tuples that join with $\triangle O$ have $ordID$ values greater than the maximum $ordID$. Inserted L tuples that join with O have $ordID$ values equal to the maximum $ordID$.

$$
\begin{aligned}
\nabla L &\equiv \sigma_{\texttt{false}}(\nabla L) &(5.7)\\
\sigma_{\triangle O.ordID=\triangle L.ordID}(\triangle O \times \triangle L) &\equiv \sigma_{\triangle O.ordID=\triangle L.ordID} \\
&\qquad (\triangle O \times (\triangle L \overline{\ltimes}_{\triangle L.ordID \leq O.ordID} O)) &(5.8)\\
\sigma_{O.ordID=\triangle L.ordID}(O \times \triangle L) &\equiv \sigma_{O.ordID=\triangle L.ordID} \\
&\qquad (O \times (\triangle L \overline{\ltimes}_{\triangle L.ordID < O.ordID} O)) &(5.9)
\end{aligned}
$$

Key constraints The schema in Example 5.1.1 assumes that *custID* is the key of C. The constraints below are implied by this key constraint. Constraints (5.10) and (5.11), which use the table renaming operator ρ, enforce the functional dependency implied by the key constraint. Finally, Constraint (5.12) enforces that none of the keys of the inserted tuples are in C. Similar constraints are implied by the assumptions that *ordID* is the key of O and both *ordID* and *partID* make up the key of L.

$$
\begin{aligned}
C &\equiv C \overline{\ltimes}_{(C.custID=C'.custID) \,\wedge\, (C.info \neq C'.info)} \rho_{C'}(C) &(5.10)\\
\nabla C &\equiv \nabla C \overline{\ltimes}_{(\nabla C.custID=\nabla C'.custID) \,\wedge\, (\nabla C.info \neq \nabla C'.info)} \rho_{\nabla C'}(\nabla C) &(5.11)\\
\triangle C &\equiv \triangle C \overline{\ltimes}_{\triangle C.custID=C.custID} C &(5.12)
\end{aligned}
$$

Referential integrity constraints

Given the schema introduced in Example 5.1.1, it is reasonable to assume that there is a referential integrity constraint from attribute *O.custID* to key *C.custID*. The following constraints express this assumption. Similar constraints are used to express a referential integrity constraint from attribute *L.ordID* to key *O.ordID*.

$$
\begin{aligned}
O &\equiv O \ltimes_{O.custID=C.custID} C &(5.13)\\
\triangle O &\equiv \triangle O \ltimes_{\triangle O.custID=C.custID} C &(5.14)\\
\nabla O &\equiv \nabla O \ltimes_{\nabla O.custID=C.custID} C &(5.15)
\end{aligned}
$$

Weak minimality constraints

It is also reasonable to assume that deletions from C are *weakly minimal* [GL95]. That is, all the deleted C tuples were previously in C.

$$\triangledown C \quad \equiv \quad \triangledown C \ltimes_{(\triangledown C.custID=C.partID)\ \wedge\ (\triangledown C.info=C.info)} C \tag{5.16}$$

Ad hoc constraints

Finally, we illustrate that CL can be used to express fairly ad hoc constraints. For instance, the constraint $\sigma_{custID<1000}(O) \equiv \sigma_{(custID<1000)\ \wedge\ (clerk=\text{"}\mathtt{Clerk1}\text{"})}(O)$, expresses that customers with $custID < 1000$ are handled by `Clerk1`.

□

5.4.2 Modifying Maintenance Subexpressions

Given a maintenance subexpression E, we now modify E by applying a given set of CL constraints to it. Intuitively, since $LHS(c)$ and $RHS(c)$ of a CL constraint c are equivalent, whenever $LHS(c)$ "matches" a subquery of E, we can substitute $RHS(c)$ for $LHS(c)$ in E. We say a constraint c is *applied* to E when we successfully match $LHS(c)$ to a subquery of E and replace the matching subquery with $RHS(c)$. The challenge is of course in determining whether $LHS(c)$ matches some subquery of E since a syntactic check does not suffice. For instance, if E is $\sigma_{a>10}(\triangle R)$ and $LHS(c)$ is $\sigma_{a>5}(\triangle R)$, $LHS(c)$ matches a subquery of E since E can be rewritten as $\sigma_{a>10}(\sigma_{a>5}(\triangle R))$. The next example provides additional illustration of how a constraint is applied.

EXAMPLE 5.4.2 Most of the constraints in Example 5.4.1 are context-free and applying them is trivial. For instance, applying Constraint (5.5) (*i.e.*, $\triangledown O \equiv \sigma_{\mathtt{false}}(\triangledown O)$) simply requires finding occurrences of $\triangledown O$ in a maintenance subexpression E and replacing it with $\sigma_{\mathtt{false}}(\triangledown O)$. Since E has a conjunctive condition that includes `false`, E is guaranteed to result in an empty answer.

To make the current example more interesting, let us consider applying the context-sensitive constraint c (*i.e.*, Constraint (5.9), Example 5.4.1)

$$\sigma_{O.ordID=\triangle L.ordID}(O\times\triangle L) \quad \equiv \quad \sigma_{O.ordID=\triangle L.ordID}(O\times(\triangle L\ \overline{\ltimes}_{L.ordID<O.ordID}O)),$$

to the following maintenance subexpression E of the $ClerkCust$ view.

$$\pi_{O.clerk,C.custID,\triangle L.qty,\triangle L.cost}$$

$$\sigma_{\triangle L.cost>99 \wedge C.custID<500 \wedge O.ordID>1000 \wedge O.ordID=\triangle L.ordID \wedge O.custID=C.custID}$$
$$(C \times O \times \triangle L)$$

The previous maintenance subexpression can be rewritten as

$$\pi_{O.clerk,C.custID,\triangle L.qty,\triangle L.cost}$$
$$\sigma_{\triangle L.cost>99 \wedge C.custID<500 \wedge O.ordID>1000 \wedge O.custID=C.custID}$$
$$(C \times \sigma_{O.ordID=\triangle L.ordID}(O \times \triangle L)).$$

Clearly $LHS(c)$ matches a subquery of E. Hence, we can replace the matching subquery with $RHS(c)$, yielding the following maintenance subexpression.

$$\pi_{O.clerk,C.custID,\triangle L.qty,\triangle L.cost}$$
$$\sigma_{\triangle L.cost>99 \wedge C.custID<500 \wedge O.ordID>1000 \wedge O.ordID=\triangle L.ordID \wedge O.custID=C.custID}$$
$$(C \times O \times (\triangle L \;\overline{\rhd\!\!\lhd}_{L.ordID<O.ordID}\; O))$$

□

The previous example illustrated algorithm *Apply* (Algorithm 5.4.1, Figure 5.5) for applying a constraint c on a maintenance subexpression E. *Apply* first checks if the views in $LHS(c)$ are also in E (Step 1).[3] It then checks if the conditions in E imply the conditions in $LHS(c)$ (Step 2). This check can be done efficiently because the conditions involved are conjunctive [Ull89a].[4] If both checks are passed, then $LHS(c)$ matches a subquery of E. For instance, suppose that E is

$$\pi_{\mathcal{A}}\sigma_{\mathcal{P}}(\times_{R\in\mathcal{R}}R) \rhd\!\!\lhd S \ldots \overline{\rhd\!\!\lhd} T \ldots,$$

and $LHS(c)$ is $\sigma_{\mathcal{P}_{LHS}}(\times_{U\in\mathcal{U}}U)$. If $\mathcal{U} \subseteq \mathcal{R}$ and $\mathcal{P} \Rightarrow \mathcal{P}_{LHS}$, it is guaranteed that E is equivalent to

$$\pi_{\mathcal{A}}\sigma_{\mathcal{P}}((\times_{R\in(\mathcal{R}-\mathcal{U})}R) \times \sigma_{\mathcal{P}_{LHS}}(\times_{U\in\mathcal{U}}U)) \rhd\!\!\lhd S \ldots \overline{\rhd\!\!\lhd} T \ldots .$$

[3]This check suffices since we only handle view definitions with no self-joins. Otherwise, all possible mappings from the views in c to those in E have to be checked.

[4]It can be done in $O(n^3)$ time, where n is the number of distinct attributes in the conditions. This assumes that the cardinality of the domain of the attributes is greater than or equal to n to handle $\neq$'s.

The subquery of E that matches $LHS(c)$ can then be replaced by $RHS(c)$. Redundant conditions are eliminated in Step 3 of *Apply* by solving another implication problem. Finally, any conditions added are pulled out of the cross product to facilitate the application of other constraints.

Algorithm 5.4.1 *Apply*
Input: maintenance subexpression E, CL constraint c
Output: `true` if c is applied, `false` otherwise **Side effect:** may modify E
Let E be of the form: $\pi_{\mathcal{A}}(\sigma_{\mathcal{P}}(\times_{R \in \mathcal{R}} R) \ltimes S \ldots \overline{\ltimes} T \ldots)$
Let c be of the form: $\sigma_{\mathcal{P}_{LHS}}(\times_{U \in \mathcal{U}} U) \equiv \sigma_{\mathcal{P}_{RHS}}(\times_{U \in \mathcal{U}} U) \ltimes V$ (or $\overline{\ltimes} V$)
1. If $\mathcal{U} \subseteq \mathcal{R}$
  2. If $\mathcal{P} \Rightarrow \mathcal{P}_{LHS}$
    3. Remove any conditions in $\mathcal{P}$ that are implied by $\mathcal{P}_{RHS}$
    4. $E \leftarrow \pi_{\mathcal{A}}(\sigma_{\mathcal{P} \wedge \mathcal{P}_{RHS}}(\times_{R \in \mathcal{R}} R)) \ltimes S \ldots \ltimes V \ldots \overline{\ltimes} T \ldots$
    5. Return `true`
6. Return `false`

$\diamond$

Algorithm 5.4.2 *Modify*
Input: maintenance subexpression E, a set of CL constraints $\mathcal{C}$
Side effect: may modify subexpression E
1. *change* $\leftarrow$ `true`
2. While (*change* $=$ `true`)
  3. *change* $\leftarrow$ `false`
  4. For (each constraint c in $\mathcal{C}$)
    5. If (*Apply*$(E, c) =$ `true`)
      6. Remove c from $\mathcal{C}$, *change* $\leftarrow$ `true`

$\diamond$

Figure 5.5: Algorithm For Modifying a Maintenance Subexpression

Although *Apply* always modifies E to an equivalent subexpression, it is not complete since it may not apply a constraint even when equivalence is preserved. This is because Step 2 only takes into account the selection and join conditions in $\mathcal{P}$, but not the `exists` and `not exists` conditions given by the $\ltimes$ and $\overline{\ltimes}$ operators. (`Exists` conditions can be handled but it is not shown in *Apply*.) To obtain a complete algorithm, the implication problem $\mathcal{P}' \Rightarrow \mathcal{P}_{LHS}$ must be solved, where $\mathcal{P}'$ is the conjunction of all the selection, join, `exists` and `not exists` conditions. Unfortunately, there are no known complete algorithms to solve the general implication problem with a mixture of existential and universal quantifiers

([YL87]).

In Section 5.4.3, we develop an algorithm to compute the closure of a conjunctive condition which may include `exists` conditions but only atomic `not exists` conditions. This algorithm can be useful in solving a more general implication problem than the one in Step 2. However, we do not show it here since taking into account `exists` and `not exists` conditions is not critical in *Apply*. This is because in practice, many constraints are context-free and can be applied easily. Context-sensitive constraints, like the append-only and implication constraints in Example 5.4.1, usually only require examining the selection and join conditions of E.

So far, we have discussed how a single constraint is applied to E. When there is a set of constraints to be applied, the order of application does not matter. More specifically, applying a constraint c_1 to E before c_2 does not jeopardize the "applicability" of c_2 because applying c_1 only adds conditions to E. On the other hand, if initially c_2 cannot be applied, applying c_1 may add enough conditions to E so that c_2 can now be applied. Thus, after a constraint is applied, we must check if any of the unapplied constraints can be applied. Also note that any constraint can only be applied once and it can only match one subquery of E since E has no self-joins.

Algorithm 5.4.2 (Figure 5.5) shows the algorithm *Modify* for applying a set of constraints $\mathcal{C}$ to E. Although efficiency is not at a premium when marking extensions, *Modify* has a tolerable overall complexity of $O(|\mathcal{C}|^2 \cdot n^3)$, assuming the check in Step 1 of *Apply* is done in constant time. $|\mathcal{C}|$ is the number of constraints and n is the number of distinct attributes used in $\mathcal{P}$ of E.

5.4.3 Deriving $\texttt{Needed}_{\mathcal{C}}$

Given the maintenance subexpressions $\mathcal{E}$, we can use *Modify* to alter each subexpression in $\mathcal{E}$ based on $\mathcal{C}$, and produce a new set of subexpressions $\mathcal{E}_{\mathcal{C}}$. In this section, we first discuss why using `Needed` on $\mathcal{E}_{\mathcal{C}}$ is not satisfactory. A function that identifies all and only the tuples needed by $\mathcal{E}_{\mathcal{C}}$ is complex since it needs to solve hard problems, *e.g.*, closure of a non-conjunctive condition. Thus, in this section, we develop a fairly efficient $\texttt{Needed}_{\mathcal{C}}$ function which handles `exists` and some `not exists` conditions, namely, those composed of one or a disjunction of atomic conditions. In the latter part of the section, we give a lemma that formally describes the properties of $\texttt{Needed}_{\mathcal{C}}$.

Problem with `Needed`

Strictly speaking, `Needed` was not defined to work with maintenance subexpressions with `exists` and `not exists` conditions. Nevertheless, the `Needed`$(T, \mathcal{E}_{\mathcal{C}})$ function can be adapted to apply to $\mathcal{E}_{\mathcal{C}}$ by modifying `Map`(E, T) to return the following query for each $E \in \mathcal{E}_{\mathcal{C}}$.

$$\pi_{\texttt{Attrs}(T)}\sigma_{\texttt{Ignore}(\texttt{Closure}(\mathcal{P}),(\mathcal{D}-\{T\}))}(\times_{R\in(\mathcal{R}-(\mathcal{D}-\{T\}))}R)\ltimes S\ldots\overline{\ltimes}T\ldots$$

`Map` must also ignore `exists` and `not exists` conditions involving views in $\mathcal{D} - \{T\}$. The above query still works but may deem more tuples as needed since `Closure` only takes into account the selection and join conditions but not the `exists` and `not exists` conditions.

Later in this section, we develop a new function `Closure`$_{\mathcal{C}}$, which takes into account `exists` and atomic `not exists` conditions. We then define `Map`$_{\mathcal{C}}$ similar to `Map` but using `Closure`$_{\mathcal{C}}$, and `Needed`$_{\mathcal{C}}$ similar to `Needed` but using `Map`$_{\mathcal{C}}$. Before we derive `Closure`$_{\mathcal{C}}$, we illustrate why taking into account the `exists` and `not exists` conditions is important in computing the closure.

EXAMPLE 5.4.3 In this example, we compare the tuples returned by `Map`$(E_{\mathcal{C}}, O)$ and `Map`$_{\mathcal{C}}(E_{\mathcal{C}}, O)$, where $E_{\mathcal{C}}$ is obtained by applying a set of constraints to

$$E = \pi_{O.clerk,\triangle C.custID,L.qty,L.cost}$$
$$\sigma_{L.cost>99\ \wedge\ \triangle C.custID<500\ \wedge\ O.ordID>1000\ \wedge\ O.ordID=L.ordID\ \wedge\ O.custID=\triangle C.custID}$$
$$(\triangle C \times O \times L).$$

Let us suppose that only the constraints expressing the following information are applied to E: (1) $custID$ is the key of C (Constraint (5.12)); and (2) a referential integrity holds from $O.custID$ to $C.custID$ (Constraint (5.13)). The modified maintenance subexpression $E_{\mathcal{C}}$ is as follows:

$$E_{\mathcal{C}} = \pi_{O.clerk,\triangle C.custID,L.qty,L.cost}$$
$$\sigma_{L.cost>99\ \wedge\ \triangle C.custID<500\ \wedge\ O.ordID>1000\ \wedge\ O.ordID=L.ordID\ \wedge\ O.custID=\triangle C.custID}$$
$$((\triangle C\ \overline{\ltimes}_{\triangle C.custID=C.custID} C)\times(O \ltimes_{O.custID=C.custID} C)\times L).$$

Notice that `Map`$(E_{\mathcal{C}}, O)$ returns

$$\pi_{\texttt{Attrs}(O)}\sigma_{L.cost>99\ \wedge\ O.custID<500\ \wedge\ O.ordID>1000\ \wedge\ O.ordID=L.ordID}$$
$$((O \ltimes_{O.custID=C.custID} C)\times L),$$

after computing the closure of the selection and join conditions, ignoring the conditions referring to $\triangle C$, and removing $\triangle C$ from the cross product.

On the other hand, let us suppose that $\texttt{Map}_{\mathcal{C}}$ uses the function $\texttt{Closure}_{\mathcal{C}}$ to "handle" `exists` and `not exists` conditions obtaining the following subexpression from $E_{\mathcal{C}}$.

$$\begin{aligned}
&\pi_{O.clerk, \triangle C.custID, L.qty, L.cost} \\
&\quad \sigma_{L.cost>99 \,\wedge\, \triangle C.custID<500 \,\wedge\, O.ordID>1000 \,\wedge\, O.ordID=L.ordID \,\wedge\, O.custID=\triangle C.custID} \\
&\qquad ((\triangle C \;\overline{\ltimes}_{\triangle C.custID=C.custID} C \ltimes_{\triangle C.custID=C.custID} C) \times \\
&\qquad (O \ltimes_{O.custID=C.custID \wedge O.custID \neq C.custID} C \;\overline{\ltimes}_{O.custID=C.custID} C) \times L)
\end{aligned}$$

Given the above subexpression, $\texttt{Map}_{\mathcal{C}}$ returns the following query

$$\begin{aligned}
&\pi_{\texttt{Attrs}(O)}\, \sigma_{L.cost>99 \,\wedge\, O.custID<500 \,\wedge\, O.ordID>1000 \,\wedge\, O.ordID=L.ordID} \\
&\quad ((O \ltimes_{O.custID=C.custID \wedge O.custID \neq C.custID} C \;\overline{\ltimes}_{O.custID=C.custID} C) \times L).
\end{aligned}$$

This query has an empty answer because the `exists` condition on O is contradictory! Hence, $\texttt{Map}_{\mathcal{C}}(E_{\mathcal{C}}, O)$ correctly states that no O tuple is needed in answering E, which makes sense because the new customers do not have any orders yet according to the constraints. On the other hand, $\texttt{Map}(E_{\mathcal{C}}, O)$ returns a possibly severe overestimate of the O tuples needed. $\square$

Alternative representation of $\ltimes$'s and $\overline{\ltimes}$'s

For convenience, we develop $\texttt{Closure}_{\mathcal{C}}$ to work on maintenance subexpressions that represent `exists` and `not exists` conditions differently. Instead of representing them using the $\ltimes$ and $\overline{\ltimes}$ operators, we represent them as conditions that are combined with the selection and join conditions. For instance, the query $R \ltimes_{R.a=S.a} S$ is represented as $\sigma_{\exists S_i \in S(R.a=S_i.a)}(R)$, where S_i is a tuple variable ([Ull89a]). The query $R \;\overline{\ltimes}_{R.a=S.a}\; S$ is represented as $\sigma_{\neg \exists S_i^{asj} \in S(R.a=S_i^{asj}.a)}(R)$, or alternatively $\sigma_{\forall S_i^{asj} \in S(R.a \neq S_i^{asj}.a)}(R)$. We call this new representation the *quantifier representation*, and the previous one, the *operator representation*.

In the quantifier representation, we make implicit tuple variables, like "R" in the `exists` condition $\exists S_i \in S(R.a = S_i.a)$, explicit. Hence, given the maintenance subexpression

$$\begin{aligned}
&\pi_{O.clerk, \triangle C.custID, L.qty, L.cost} \\
&\quad \sigma_{L.cost>99 \,\wedge\, \triangle C.custID<500 \,\wedge\, O.ordID>1000 \,\wedge\, O.ordID=L.ordID \,\wedge\, O.custID=\triangle C.custID} \\
&\qquad ((\triangle C \;\overline{\ltimes}_{\triangle C.custID=C.custID} C) \times (O \ltimes_{O.custID=C.custID} C) \times L),
\end{aligned}$$

its quantifier representation is $\pi_{O_0.clerk,\triangle C_0.custID,L_0.qty,L_0.cost}\sigma_{\mathcal{P}'}(\triangle C \times O \times L)$, where $\mathcal{P}'$ is

$$\begin{aligned}
&L_0.cost > 99 \wedge \triangle C_0.custID < 500 \wedge O_0.ordID > 1000 \wedge \\
&O_0.ordID = L_0.ordID \wedge O_0.custID = \triangle C_0.custID \wedge \\
&\forall C_2^{asj}(\triangle C_0.custID \neq C_2^{asj}.custID) \wedge \exists C_1(O_0.custID = C_1.custID). \qquad (5.17)
\end{aligned}$$

We assign the tuple variables mechanically as follows. For a view T appearing in the cross product (*e.g.*, $\triangle C$, O, L), we assign the tuple variable T_0 (*e.g.*, $\triangle C_0$, O_0, L_0). For a view T appearing in an `exists` condition $R \ltimes T$, we assign a unique tuple variable T_i (*e.g.*, C_1), where $i > 0$. For a view T appearing in a `not exists` condition $R \overline{\ltimes} T$, we assign a unique tuple variable T_j^{asj} (*e.g.*, C_2^{asj}), where $j > 0$. Henceforth, we use "T" to denote either a free variable T_0, or an existentially quantified variable T_i, or a universally quantified tuple variable T_j^{asj}.

Deriving $\texttt{Closure}_\mathcal{C}$, $\texttt{Map}_\mathcal{C}$, and $\texttt{Neeeded}_\mathcal{C}$

In general, given a maintenance subexpression $E = \pi_\mathcal{A}\sigma_\mathcal{P}(\times_{R \in \mathcal{R}} R)$ in quantifier representation, we can always obtain the *prenex normal form* (PNF) of $\mathcal{P}$, where all the quantifiers precede a quantifier-free condition expression ([PMW90]). That is $\mathcal{P}$ in PNF is of the form shown below where $\mathcal{P}'$ is a quantifier-free condition.

$$\exists R_i..\exists S_j..\forall T_k^{asj}..\forall U_l^{asj}(\mathcal{P}')$$

Assuming $\mathcal{P}'$ is conjunctive for now, $\texttt{Closure}_\mathcal{C}$ simply derives new atomic conditions from atomic conditions that use universally quantified tuple variables (*e.g.*, T_i^{asj}), and then uses the old `Closure` function to obtain the closure. More specifically, `Closure` uses standard axioms, such as the transitivity axiom, to derive atomic conditions ([Ull89a]). $\texttt{Closure}_\mathcal{C}$ adds the following two axioms to derive additional atomic conditions from ones that use universally quantified variables.

1. $S_i^{asj}.a\ \theta\ T.b \Rightarrow S.a\ \theta\ T.b$, where θ is either $=,\neq,\leq,<,\geq$, or $>$.

2. $S_i^{asj}.a = T_j.b \Rightarrow S_i^{asj}.a = S_k^{asj}.a$.

The first (additional) axiom states that if $S_i^{asj}.a\ \theta\ T.b$ holds, it means that the a attribute of all the S tuples are related to $T.b$ in the same way. Hence, an atomic condition $S.a\ \theta\ T.b$

holds regardless of whether S is existentially or universally quantified. The second axiom states that if $S_i^{asj}.a$ is equated to an attribute of an existentially quantified tuple variable, it must be the case that the a attributes of all the S tuples have the same value. Note that S_k^{asj} must be distinct from S_i^{asj}. If no such tuple variable exist, we introduce a new one for the purpose of applying the second axiom. We illustrate $\texttt{Closure}_{\mathcal{C}}$ in the next example.

EXAMPLE 5.4.4 Let us suppose we are given $E = \pi_{\mathcal{A}}\sigma_{\mathcal{P}}(\times_{R\in\mathcal{R}}R)$, where $\mathcal{P}$ is Expression (5.17). $\mathcal{P}$ in PNF is $\exists C_1 \forall C_2^{asj}\ (\mathcal{P}')$, where $\mathcal{P}'$ is

$$L_0.cost > 99 \wedge \triangle C_0.custID < 500 \wedge O_0.ordID > 1000 \wedge O_0.ordID = L_0.ordID \wedge$$
$$O_0.custID = \triangle C_0.custID \wedge \triangle C_0.custID \neq C_2^{asj}.custID \wedge O_0.custID = C_1.custID.$$

Since both C_1 and C_2^{asj} are tuple variables ranging over the domain of view C's tuples, and C_2^{asj} is a universally quantified tuple variable, any atomic condition that applies to C_2^{asj} must also apply to C_1 (i.e., the first axiom). That is, a condition that applies to all tuples must apply to a particular tuple. For instance, the atomic condition $\triangle C_0.custID \neq C_2^{asj}.custID$ implies the atomic condition $\triangle C_0.custID \neq C_1.custID$. Notice that when `Closure` is run on $(\mathcal{P}' \wedge (\triangle C_0.custID \neq C_1.custID))$, the contradictory atomic conditions $O_0.custID = C_1.custID$ and $O_0.custID \neq C_1.custID$ is derived from $\triangle C_0.custID \neq C_1.custID$, $O_0.custID = \triangle C_0.custID$ and $O_0.custID = C_1.custID$. Consequently, $\texttt{Map}(O, E)$ is guaranteed to return an empty answer which is consistent with Example 5.4.3. On the other hand, if `Closure` is run on $\mathcal{P}'$ alone, no contradictory atomic conditions are derived. □

Algorithm 5.4.3 $\texttt{Closure}_{\mathcal{C}}$
Input: conjunctive condition $\mathcal{P}$ possibly with `exists` and
(atomic) `not exists` conditions in quantifier representation
Output: closure of $\mathcal{P}$

1. Derive PNF of $\mathcal{P}$ of the form $\exists..\exists..\forall..\forall..(\mathcal{P}')$, where $\mathcal{P}'$ is quantifier-free
2. Derive $\mathcal{P}'$ from $\mathcal{P}$ by applying the two axioms
concerning universally quantified tuple variables.
3. Return $\exists..\exists..\forall..\forall..(\texttt{Closure}(\mathcal{P}'))$

◇

Figure 5.6: $\texttt{Closure}_{\mathcal{C}}$

The example illustrated $\texttt{Closure}_{\mathcal{C}}$ (Algorithm 5.4.3, Figure 5.6) which computes the closure of a conjunctive condition $\mathcal{P}$, possibly with `exists` and `not exists` conditions. $\texttt{Closure}_{\mathcal{C}}$ first converts $\mathcal{P}$ to its PNF, obtaining a quantifier-free condition $\mathcal{P}'$ (Step 1). To ensure that $\mathcal{P}'$ is still conjunctive, we assume that `not exists` conditions only have a single atomic condition. That is, they are of the form $\neg\exists T_i^{asj} p$ (or $\forall T_i^{asj} \neg p$), where p is a single atomic condition.[5] Any `not exists` conditions that do not conform to the previous restriction are ignored (replaced with `true`) when computing the closure. (The `not exists` condition added by Constraint (5.11) is an example of an ignored `not exists` condition.) $\texttt{Closure}_{\mathcal{C}}$ then derives new atomic conditions (Step 2) based on the two additional axioms introduced previously. Finally, the old `Closure` function is used to compute the closure of the quantifier-free conjunctive condition $\mathcal{P}'$ as if it was a conjunction of selection and join conditions.

$\texttt{Closure}_{\mathcal{C}}$ is reasonably efficient and can be done in $O(n^3+m^2\cdot a)$, where n is the number of distinct attributes, m is the number of distinct tuple variables, and a is the number of atomic conditions in $\mathcal{P}$. (Step 2 is done in $O(m^2\cdot a)$ time and Step 5 is done in $O(n^3)$ time.)

Using $\texttt{Closure}_{\mathcal{C}}$, we define $\texttt{Map}_{\mathcal{C}}$ to be the same as `Map` except that it uses $\texttt{Closure}_{\mathcal{C}}$, and $\texttt{Needed}_{\mathcal{C}}$ to be the same as `Needed` except that it uses $\texttt{Map}_{\mathcal{C}}$.

Lemma 5.4.1 *Given a view T and a set of maintenance subexpression $\mathcal{E}_{\mathcal{C}}$ obtained by applying the constraints $\mathcal{C}$ on a set of maintenance subexpression $\mathcal{E}$, the query*

$$\texttt{Needed}_{\mathcal{C}}(T,\mathcal{E}_{\mathcal{C}}) = \bigcup_{E_{\mathcal{C}}\in\mathcal{E}_{\mathcal{C}}} \texttt{Map}_{\mathcal{C}}(E_{\mathcal{C}},T),$$

returns all the tuples in T that are needed by the maintenance subexpressions in $\mathcal{E}_{\mathcal{C}}$. If all constraints in $\mathcal{C}$ using `not exists` *conditions are of the form*

$$\sigma_{\mathcal{P}_{LHS}}(\times_{R\in\mathcal{R}}R) \equiv \sigma_{\mathcal{P}_{RHS}}(\times_{R\in\mathcal{R}}R)\overline{\triangleright\!\!<}_{p}T$$

where p is a disjunction of atomic predicates, the query $T\triangleright\!\!<_{\texttt{Attrs}(T)}\texttt{Needed}(T,\mathcal{E})$ returns only the tuples in T that are needed by the maintenance subexpressions in $\mathcal{E}_{\mathcal{C}}$. Furthermore, for any set of constraints $\mathcal{C}$, it is guaranteed that $\texttt{Needed}_{\mathcal{C}}(T,\mathcal{E}_{\mathcal{C}}) \subseteq \texttt{Needed}(T,\mathcal{E}_{\mathcal{C}}) \subseteq \texttt{Needed}(T,\mathcal{E})$. □

[5] A `not exists` condition composed of a disjunction of atomic conditions is allowed but this can be expressed as separate `not exists` conditions with a single atomic condition.

The proof for Lemma 5.4.1, together with all the details of on the completeness of $\texttt{Closure}_{\mathcal{C}}$ and its impact on $\texttt{Needed}_{\mathcal{C}}$, can be found in Appendix C.

5.5 Discussion

Although Lemma 5.4.1 itself does not guarantee that $\texttt{Needed}_{\mathcal{C}}$ always returns strictly fewer tuples than `Needed`, we now illustrate that in practice, $\texttt{Needed}_{\mathcal{C}}$ often returns much fewer tuples.

ClerkCust View

The *ClerkCust* view has 27 maintenance subexpressions, which we assume to comprise $\mathcal{E}$. (The maintenance subexpressions are listed in report [GMLY98].) $\mathcal{C}$ are the append-only, key, referential integrity, weak minimality and ad hoc constraints in Example 5.4.1. Table 5.2 gives the queries returned by $\texttt{Needed}(T, \mathcal{E})$ and $\texttt{Needed}_{\mathcal{C}}(T, \mathcal{E}_{\mathcal{C}})$ for views L, O and C.

The second row of Table 5.2 shows that $\texttt{Needed}_{\mathcal{C}}(L, \mathcal{E}_{\mathcal{C}})$ identifies accurately that none of the L tuples are needed by $\mathcal{E}$, while $\texttt{Needed}(L, \mathcal{E})$ deems a large number of L tuples as needed. $\texttt{Needed}_{\mathcal{C}}$ is much better because it eliminates any maintenance subexpression E where $\texttt{Map}(L, E)$ is guaranteed to return an empty answer given the constraints. Of the 27 subexpressions in $\mathcal{E}$, 20 are eliminated. Of the 7 remaining subexpressions, none refer to L. ($\triangle L$ and $\triangledown L$ are used but not L.)

The third row of Table 5.2 shows that $\texttt{Needed}_{\mathcal{C}}(O, \mathcal{E}_{\mathcal{C}})$ identifies accurately (using a `not exists` condition) that only the *one* O tuple with the maximum *ordID* value is needed. On the other hand, $\texttt{Needed}(O, \mathcal{E})$ deems a large number of O tuples as needed.

The fourth row of Table 5.2 shows that $\texttt{Needed}_{\mathcal{C}}(C, \mathcal{E}_{\mathcal{C}})$ and $\texttt{Needed}(C, \mathcal{E})$ identify the same bag of needed tuples. This illustrates that using $\texttt{Needed}_{\mathcal{C}}$ does not always help in reducing the number of tuples that are deemed needed.

TPC-D Benchmark

We now investigate what TPC-D ([Com]) base view tuples are needed assuming certain TPC-D queries are used as views. In particular, we focus on 4 out of the 9 TPC-D base views: *LINEITEM* (L), *ORDER* (O), *CUSTOMER* (C) and *PART* (P). Fact views L and O contain 86% of the tuples in the benchmark. Hence, expiration requests will likely be issued on these two views. We consider two views, V_3 and V_5, whose view definitions are the TPC-D queries $Q3$ ("Shipping Priority Query") and $Q5$ ("Local Supplier Volume

Query"), respectively. We assume that either the maintenance subexpressions of V_3 or V_5 comprise $\mathcal{E}$. (Other queries that refer to the four views give similar results.) Finally, the set of constraints $\mathcal{C}$ we consider is based on the TPC-D "update model" specification (see [Com]).

view T	$\texttt{Needed}_{\mathcal{C}}(T, \mathcal{E}_{\mathcal{C}})$	$\texttt{Needed}(T, \mathcal{E})$
L	$\{\}$	$\pi_{\texttt{Attrs}(L)}\sigma_{L.cost>99 \wedge L.ordID>1000}(L)$
O	$\pi_{\texttt{Attrs}(O)}\sigma_{O.custID<500 \wedge O.ordID>1000}$ $(O \ltimes_{O.ordID<O'.ordID} \rho_{O'} O)$	$\pi_{\texttt{Attrs}(O)}\sigma_{O.custID<500 \wedge O.ordID>1000}(O)$
C	$\pi_{\texttt{Attrs}(C)}\sigma_{C.custID<500}(C)$	$\pi_{\texttt{Attrs}(C)}\sigma_{C.custID<500}(C)$

Table 5.2: Comparison of $\texttt{Needed}_{\mathcal{C}}$ and $\texttt{Needed}$ Using $ClerkCust$

view T	$\texttt{Needed}_{\mathcal{C}}(T, \mathcal{E}_{\mathcal{C}})$	$\texttt{Needed}(T, \mathcal{E})$
L	0%	100%
O	0%	100%
C	20%	20%
P	0%	0%

Table 5.3: Comparison of $\texttt{Needed}_{\mathcal{C}}$ and $\texttt{Needed}$ Using TPC-D Query $Q3$

view T	$\texttt{Needed}_{\mathcal{C}}(T, \mathcal{E}_{\mathcal{C}})$	$\texttt{Needed}(T, \mathcal{E})$
L	0%	100%
O	0%	100%
C	100%	100%
P	100%	100%

Table 5.4: Comparison of $\texttt{Needed}_{\mathcal{C}}$ and $\texttt{Needed}$ Using TPC-D Query $Q5$

To simplify the presentation, we do not give the queries returned by the functions but instead give the percentage of the base view tuples that are needed. We obtained this percentage for each view T (*i.e.*, L, O, C, and P) by running the queries returned by $\texttt{Needed}_{\mathcal{C}}(T, \mathcal{E}_{\mathcal{C}})$ and $\texttt{Needed}(T, \mathcal{E})$. We then counted the number of tuples in the result and divided it by the number of T tuples.

Table 5.3 gives the tuples that are needed by the maintenance subexpressions of V_3 assuming the constraints in $\mathcal{C}$. $\texttt{Needed}_{\mathcal{C}}$ identifies that none of the L and O tuples are needed, and 20% of the C tuples are needed. Since P is not referred to in V_3's view definition, none of its tuples are needed to maintain V_3. None of the L and O tuples are needed because of the append-only behavior of L and O specified in the benchmark, *i.e.*, $\triangle L$ tuples only join with $\triangle O$ tuples and vice versa. Only 20% of the C tuples are needed because $\texttt{Needed}_{\mathcal{C}}$ applies a selection condition on C with 20% selectivity. On the other hand, $\texttt{Needed}$ deems all of the L and O tuples as needed.

Table 5.4 shows similar results assuming the maintenance subexpressions of view V_5 comprise $\mathcal{E}$. The only difference is that both $\texttt{Needed}_{\mathcal{C}}$ and $\texttt{Needed}$ identify that all the tuples of C and P are needed. This is because V_5's view definition does not apply any selection conditions on C nor P. Had there been constraints that state that "some of the customers no longer place orders", or "some parts can no longer be ordered", then $\texttt{Needed}_{\mathcal{C}}$ would mark some C and P tuples as unneeded.

The previous study shows that using constraints allows greater flexibility for expiration and can significantly decrease storage requirements when data is no longer needed. Furthermore, it is likely that the efficiency of view maintenance is improved because the expired data is no longer processed by the maintenance subexpressions. Also, we illustrated that constraints can be used to eliminate some of the maintenance subexpressions altogether which definitely improves view maintenance.

5.6 Dynamic Setting

In the previous two sections, we focused on an initial static setting wherein we are given a set of views $\mathcal{T}$, a set of maintenance subexpressions $\mathcal{E}$, and a set of constraints $\mathcal{C}$. In this section, we explore how to cope with a dynamic setting wherein some of these parameters can be changed. Furthermore, we also drop the assumption that none of the tuples have been expired.

Before discussing the algorithms, it is important to note that even when parameters change, expiration requests are satisfied the same way. That is, given an expiration request $\sigma_{\mathcal{P}}(T)$ on T, it is satisfied by removing the tuples in $\sigma_{\mathcal{P} \wedge needed=\texttt{false}}(T)$.

Also, note that the queries returned by $\texttt{Needed}_{\mathcal{C}}$ (and $\texttt{Needed}$) still have complete answers even after some tuples have been expired. This is because any query returned by $\texttt{Needed}_{\mathcal{C}}$ takes the union of subexpressions derived from maintenance subexpressions using $\texttt{Map}_{\mathcal{C}}$. Since we guaranteed that all the tuples that are needed by maintenance subexpressions are not expired, the completeness of the queries returned by $\texttt{Needed}_{\mathcal{C}}$ follows. We now outline the algorithms for coping with various changes.

Changes to $\mathcal{T}$

Suppose *Def*(V) has a complete answer and V is added to $\mathcal{T}$. We must identify for each view T that V is defined on, which of the T tuples previously deemed as unneeded is now needed to maintain V. A reasonably efficient solution to the problem is to use the query $\sigma_{needed=\texttt{false}}(T) \ltimes_{\texttt{Attrs}(T)} \texttt{Needed}(T, \mathcal{E}_V)$, where $\mathcal{E}_V$ are the maintenance subexpressions of V. This query identifies the unneeded T tuples that now need to be marked as needed.

Changes To $\mathcal{C}$

We only allow changes to $\mathcal{C}$ that expire more tuples. There are two types of changes that satisfy this condition. First, a constraint may have been added to $\mathcal{C}$. Second, a constraint c previously in $\mathcal{C}$ may have been changed so that conditions are removed from $LHS(c)$ (*i.e.*, more opportunities for applying c) or added to $RHS(c)$ (*i.e.*, more conditions added whenever c is applied). To update the extension markings, for each view T, we use the query

$$\sigma_{needed=\texttt{true}}(T)\overline{\ltimes}_{\texttt{Attrs}(T)}\texttt{Needed}_{\mathcal{C}}(T,\mathcal{E}),$$

to identify the T tuples that were previously deemed needed (*i.e.*, $needed$ = $\texttt{true}$), but must now be marked as unneeded since they are not in $\texttt{Needed}_{\mathcal{C}}(T,\mathcal{E})$. Further, assuming the change to $\mathcal{C}$ is due to a change in *Constraint*(S), for some view S, we only need to modify the extension marking of a view T defined on S. This is valid under our assumption that the constraint of a view is not computed from the constraints of the underlying views (*i.e.*, the administrator inputs all constraints). Even without this assumption, we can still identify the views whose extension marking may be modified by defining a *view graph*. The nodes in a view graph represent base views or derived views. There is an edge $U \rightarrow V$ if V is defined on U. In general then, we only need to modify the extension marking of a view T if T is a node in the sub-graph "rooted" at S.

Insertions

Periodically, insertions $\triangle T$ and deletions $\bigtriangledown T$ are computed for each view T. While deleting the $\bigtriangledown T$ tuples from T does not pose any problem, inserting the $\triangle T$ tuples into T may. First, the inserted tuples need to be marked as needed or unneeded. Second, some of the unneeded tuples may need to be expired. The two problems are solved by performing the following procedure.

1. Insert $\triangle T$ and set *needed* attribute to `false` for all inserted tuples.

2. For the T tuples in $\sigma_{needed=\texttt{false}}(T)\ltimes_{\texttt{Attrs}(T)}\texttt{Needed}(T,\mathcal{E})$, set *needed* attribute to `true`.

3. Expire T tuples in $\sigma_{\mathcal{P}\wedge needed=\texttt{false}}(T)$, where $LastReq(T) = \sigma_{\mathcal{P}}(T)$.

The first step assumes all $\triangle T$ tuples are unneeded and do not need to be expired. The second step marks the $\triangle T$ tuples that are needed. The last step expires unneeded $\triangle T$ according to $LastReq(T)$. The most expensive step is clearly the second one. However, only the maintenance subexpressions of views $\mathcal{V}$ that are defined on T need to be considered. Hence, the step is reasonably efficient since it is (only) as expensive as computing the insertions to the views in $\mathcal{V}$ based on $\triangle T$.

5.7 Related Work

One of the problems that our framework tackles is how to maintain a view when only parts of the underlying views are accessible. Most work on view maintenance assumes that the complete underlying views are accessible, for example, [BLT86, CW91, GL95, GMS93, Han87, QW91]. However, there has also been work on view maintenance that assumes otherwise. [BT88] and [GJM96] identified *self-maintainable* views that can be maintained without accessing underlying views. [QGMW96], [HZ96] and [Qua97] tried to make a view self-maintainable by defining auxiliary views such that the view and the auxiliary views together are self-maintainable. The function $\texttt{Needed}(T, \mathcal{E})$ we introduce serves essentially the same purpose as an auxiliary view, although it does not have to be maintained as such. [HZ96] developed a framework wherein the attributes of the underlying views may be inaccessible. In our framework, the tuples of a view can be made inaccessible. It will be important in future work to combine both approaches.

Our framework also takes advantage of the available constraints in order to reduce the size of $\texttt{Needed}(T, \mathcal{E})$ and increase the effectiveness of expiration. This is different from, but related to, the use of constraints in the area of *semantic query optimization* [Min88, Kin81, SO89]. It is important to point out their connection since semantic query optimization has largely been ignored in view maintenance literature. Indeed, there has been some prior work in improving view maintenance using constraints; however, they all use special-case algorithms to take advantage of specific constraints. For instance, [QGMW96] used a specialized algorithm that exploits key and referential integrity constraints to eliminate maintenance subexpressions. [GJM96] used key constraints to rewrite maintenance subexpressions for a view to use itself. [JMS95] introduced *chronicles* that are updated in a special manner, and showed that views defined on chronicles can be maintained efficiently. [Vis98] uses key and referential integrity constraints to optimize view maintenance expressions. In our approach, we can describe chronicles using constraints and automatically infer that the entire chronicles can be safely expired. In summary, the techniques we introduce generalize many special-case algorithms developed in the previous work. Furthermore, since we exploit a broader class of constraints, we improve on many of the algorithms.

Our framework also introduces views whose real extensions are not complete when compared to their full extensions. There has been numerous work on incomplete databases. See [AHV95] for an overview. We are now investigating how previous work in the area can be used to solve some of the problems borne out of the framework. For instance, [Lev96]'s work on obtaining complete answers from an incomplete database is helpful in solving the fourth problem stated in Section 5.2.

Finally, the algorithms in [BCL89] for detecting irrelevant updates can be modified to detect unneeded tuples. This can be done by treating the maintenance subexpressions as views and treating a tuple $t \in T$ as if it were an insertion. However, the algorithms in [BCL89] do not work with constraints. Also, they require a satisfiability test for each tuple t. Our method is more "set-oriented" since it uses queries.

5.8 Chapter Summary

We have presented a framework for system-managed removal of warehouse data that avoids affecting user-defined materialized views over the data. Within our framework, the user or administrator can declaratively specify what he wants to expire and the system removes as much data as possible. The administrator can also input constraints (implied by the application) which the system uses to expire more data, as we illustrated using the TPC-D benchmark. We identified problems borne out of the framework and we solved the central problems by developing efficient algorithms. These problems included ones of a dynamic nature where the parameters of the framework may change.

Chapter 6

Recovery of the Load Process

6.1 Introduction

In Chapters 2, 3 and 4, we discussed techniques for improving the efficiency of the warehouse update. These techniques are important because there is a limited amount of time and resources that can be devoted to the warehouse update. A different problem that may arise is that the warehouse update may fail. Like database failures, warehouse update failures are not unlikely, due to the complexity of the warehouse update. For instance, according to the customers of a commercial data warehousing company ([Tec]), the data cleansing step of the warehouse update fails about once every thirty tries. Because of the limited amount of time and resources devoted to the warehouse update, restarting the warehouse update from scratch is very undesirable. Thus, in this chapter we develop algorithms for *resuming* a failed warehouse update.

As discussed in Chapter 1, the warehouse update conceptually has three steps.

1. Extraction of source data changes.

2. Cleansing of extracted changes.

3. Materialized view maintenance.

In this chapter, we focus on developing algorithms for resuming the second step. While developing resumption algorithms for the first step is also important, it is not as critical as developing resumption algorithms for the second and third steps. Typically, the warehouse update spends most of its time in the last two steps, especially if remote sources provide facilities for very efficiently detecting changes (*e.g.*, triggers). For instance, according to [Car97], the last two steps can take up to 24 hours to execute. On the other hand, as we saw in Chapter 2, even if snapshot differential algorithms are used, changes can be detected in a matter of seconds. However, we do not need to develop resumption algorithms for the third step because the recovery mechanism of the warehouse database can be used. For instance, if the VDAG strategy (Chapter 3) used for the third step can be partitioned into sub-transactions ([GR93]), then the warehouse database can easily resume failed VDAG strategies.

The resumption algorithms we develop in this chapter can also be used to resume failed cleansing processes for warehouse creation (as opposed to warehouse update). As discussed in Chapter 1, the cleansing process used for warehouse creation performs data integration and data cleansing to compute the initial contents of the base views. The cleansing process for warehouse update on the other hand computes a consistent set of changes to the base views. In this chapter, we develop the resumption algorithm for a *warehouse load*, which denotes the cleansing process for warehouse creation or the cleansing process for warehouse update.

Traditional recovery techniques as outlined below could be used to save partial load states, so that not all work is lost when a failure of the warehouse load occurs. However, these techniques are shunned in practice because they generate high overheads during normal processing and because they may require modification of the warehouse load processing. In this chapter we present a new, very low-overhead, technique for resuming failed loads. Our technique exploits some generic "properties" of the cleaning process used to load the warehouse, so that work is not repeated during a resumed load.

The cleaning process is typically implemented by a *workflow* of processes. There are three types of processes in the workflow. One type is an *extractor* (process) which is responsible for extracting data from a remote source, and performing data cleansing that can be done without accessing other remote sources. Another type is a *transform* which manipulates its input data to perform data cleansing and data integration. Note that a transform can perform data cleansing operations that involve multiple remote sources (*e.g.*, making the address values of two remote sources consistent). The third type is an *inserter* which puts its input data into the warehouse.

To illustrate the type of processing performed during a load, consider the load workflow of Figure 6.1. In this load workflow, extractors obtain data from the stock Trades and the

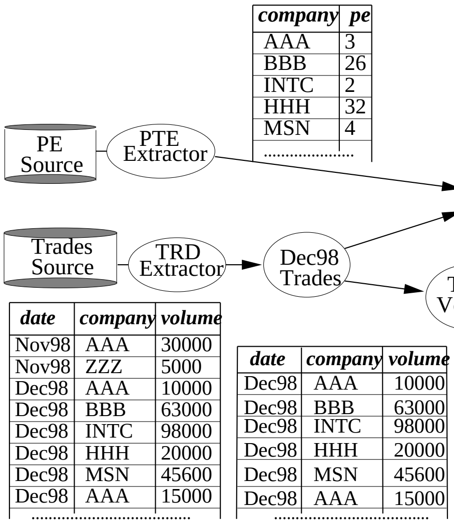

Figure 6.1: Load Workflow

price-to-earnings ratio (PE) sources. Figure 6.1 shows a prefix of the tuples extracted from each source. The stock trade data is first processed by the *Dec98Trades* transform, which only outputs trades from December 1998. Thus, the first two trades are removed since they happened in November 1998. The *TotalVolume* transform then computes the total volume of the December 1998 trades. The *PercentVolume* transform then groups the trades by company and finds the percent of the total trade volume contributed by companies whose *pe* is less than or equal to 4. For instance, companies *BBB* and *HHH* are discarded since they have high *pe*'s. An *AAA* tuple is output since its *pe* value is low: its *percentvol* value is the sum of the *AAA* volumes (25,000, assuming all *AAA* tuples are shown in the figure) divided by the *TotalVolume* output. The output of *PercentVolume* is then sent to the inserter, which stores the tuples in the warehouse.

In practice, load workflows can be much more complex than what we have illustrated, often having tens to hundreds of transforms [Tec]. Also, the transforms are not just conventional database operations (*e.g.*, join) but are often coded by application specialists to perform arbitrary processing (*e.g.*, data scrubbing, byte reordering). To load the data as fast as possible, the output tuples of each component are sent to the next as soon as they are generated to maximize pipelining.

There are many ways to recover a failed warehouse load. The fundamental features of

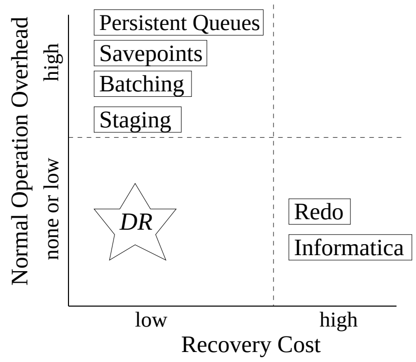

Figure 6.2: Applicability of Algorithms

various techniques are informally contrasted with our technique, called *DR*, in Figure 6.2. The vertical axis represents the *normal-operation overhead* of a technique while the horizontal axis indicates the *recovery cost* of a technique. Typically, loading the warehouse will take longer when a recovery technique is used than when no recovery technique is used. The additional time it takes to load the warehouse when a recovery technique is used is the normal-operation overhead. The reason the loading takes longer when a recovery technique is used is because additional processing is involved to save some information during normal operation to expedite the recovery.

In the lower right quadrant of Figure 6.2 are techniques that have very low normal-operation overhead. One such technique is to simply redo the entire load over again. Clearly, this technique can suffer from high recovery cost but it has no normal-operation overhead since it does not modify the load workflow. Informatica's solution [Inf] is similar: After a failure, Informatica reprocesses the data in its entirety, only filtering out the already stored tuples when they reach the warehouse for the second time (*i.e.*, just before the inserter). Sagas [GMS87, GR93] also incur high recovery cost because the load must be restarted from the beginning.

Other techniques, shown in the upper left quadrant of Figure 6.2, attempt to minimize the recovery cost by aggressively modifying the load workflow or load processing. One such technique is to divide the workflow into consecutive stages, and save intermediate results. All input data enters the first stage. All of the first stage's output is saved. The saved output then serves as input to the second stage, and so on. If a failure occurs while the second stage is active, it can be restarted, without having to redo the work performed by the first stage. Another technique in the same category is input batching wherein the input to the load workflow is divided into batches, and the batches are processed in sequence. Another technique is to take periodic savepoints [GR93] of the workflow state, or save tuples in transit in persistent queues [BHM90, BN97]. When a failure occurs, the *modified* transforms cooperate to revert to the latest savepoint, and proceed from there.

In general, techniques that require modification of the load workflow suffer from two disadvantages: (1) the normal-operation overhead is potentially high as confirmed by our experiments; and (2) the specific details of the load processing need to be known. These techniques are not straightforward to implement because careful selection of stages or batches is required to avoid high overhead. Furthermore, since the transforms are not just conventional operations, it may be hard to know their specific details.

With the *DR* technique we propose in this chapter, there is no normal-operation overhead, and the load workflow does not need to be modified. Yet, the recovery cost of *DR* can be much lower than Informatica's technique or redoing the entire load. Unlike redoing the entire load, *DR* avoids reprocessing input tuples and uses filters to intercept tuples much earlier than Informatica's technique. *DR* relies on simple and high-level transform properties (*e.g.*, are tuples processed in order?). These properties can either be declared by the transform writer or can be inferred from the basic semantics of the transform, without needing to know exactly how it is coded. After a failure, the load is restarted, except that portions that are no longer needed are "skipped." To illustrate, suppose that after a failure we discover that tuples AAA through MSN are found in the warehouse. If we know that tuples are processed in alphabetical order by the *PTE Extractor* and by the *AverageVolume* transform, the *PTE Extractor* can retrieve tuples starting with the one that follows MSN. If tuples are not processed in order, it may still be possible to generate a list of company names that are no longer needed, and that can be skipped. Our scheme is not always able to eliminate tuples during reprocessing; however, it does offer significant improvements in many cases, as in this example. During the reload, transforms operate as usual, except that they only receive the input tuples needed to generate what is missing in the warehouse. In summary, our strategy is to exploit some high-level semantics of the load workflow, and to be selective when resuming a failed load.

We note that there are previous techniques that are similar to *DR* in that they incur low normal-operation overhead but still have a low recovery cost. However, these techniques are applicable to very specific workflows for disk-based sorting [MN92], object database loading [WN95], and loading a flat file into the warehouse [RZ89, WCK93]. Our technique can handle more general workflows.

We do not claim that *DR* always recovers a load faster than other techniques. For instance, since some of the techniques modify the load processing to minimize recovery cost, these techniques can recover a failed load faster than *DR*. As mentioned, the downside of these techniques is the potential high normal-operation overhead and that the load workflow needs to be modified. However, our experiments show that *DR* is competitive if not better than these techniques for many workflows. In particular, *DR* is better for workflows that make heavy use of pipelining. Even if a workflow does not have a natural pipeline, our experiments show that a hybrid algorithm that combines *DR* and staging (or batching) can lower recovery cost.

We make the following contributions toward the efficient resumption of failed warehouse loads.

- We develop a framework for describing successful warehouse loads, and load failures. Within this framework, we identify basic properties that are useful in resuming loads.
- We develop algorithm *DR* that minimizes the recovery cost while imposing no overhead during normal operation. *DR* does not require knowing the specifics of a transform, but only its basic, high-level properties. *DR* is presented here in the context of data warehousing, but is really a generic solution for resuming any long-duration, process-intensive task.
- We develop *DR-Log* that selectively logs transform outputs to further improve on *DR*. *DR-Log* selects appropriate logging points, and uses the logs to avoid processing additional input tuples.
- We show experimentally that *DR* can significantly reduce recovery cost, as compared to traditional techniques. In our experiments we use Sagent's warehouse load package to load TPC-D tables and materialized views containing answers to TPC-D queries.

We define a warehouse loads in Section 6.2, and discuss warehouse load failure in Section 6.3. We develop the *DR* algorithm in Sections 6.4 and 6.5. We develop the *DR-Log* algorithm in Section 6.6. Experiments are presented in Section 6.7. The chapter is concluded in Section 6.8.

6.2 Normal Operation

When data is loaded into the warehouse, tuples are transferred from one component (extractor, transform, or inserter) to another. The order of the tuples is important to the resumption algorithm, so we define sequences as ordered lists of tuples with the same schema.

Definition 6.2.1 (**Sequence**) A sequence of tuples $\mathcal{T}$ is an ordered list of tuples $[t_1..t_n]$, and all the tuples in $\mathcal{T}$ have the attributes $[a_1..a_m]$. □

Before we describe a successful warehouse load, we discuss how a component directed acyclic graph (DAG) represents a load workflow, and how it is designed.

6.2.1 Component DAG Design

Figure 6.3 illustrates the same component DAG as Figure 6.1, with abbreviations for the transform names. Constructing a component DAG involves several important design decisions. First, the data obtained by the extractors is specified. Second, the transforms that process the extracted data are chosen. Moreover, if a desired transformation is not available, a user may construct a new custom-made transform. Finally, the warehouse tables(s) into which the inserter loads the data are specified. The extractors, transforms, and inserter comprise the nodes of the DAG.

Each transform and inserter expects certain *input parameter* sequences at load time. The components that supply these input parameters are also specified when the component DAG is designed. Similarly, each transform and extractor generates an output sequence to its *output parameter*. In commercial packages, the input and output parameters are specified by connecting the extractors, transforms, and the inserter together with edges in the component DAG.

In some cases, different components of a DAG may be assigned to different machines. Hence, during a load, data transfers between components may represent data transfers over the network.

As a component DAG is designed, the "properties" that hold for the various transforms and their input parameters are declared for use by our resumption algorithm. Commercial load packages already declare basic properties like tuple sequence keys. The properties that *DR* uses are explained in more detail in Section 6.4. We now illustrate a component DAG.

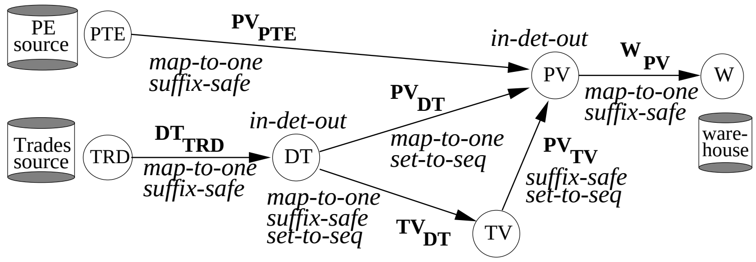

Figure 6.3: Component DAG with Properties

EXAMPLE 6.2.1 In Figure 6.3, the extractors are denoted PTE for the price-to-earnings (PE) source, and TRD for the Trades source. The transforms are denoted DT (for $Dec98Trades$), TV (for $TotalVolume$), and PV (for $PercentVolume$). The inserter is denoted W.

The input parameter(s) of each component are denoted by the component that produces the input. For instance, PV_{DT} is an input parameter of PV that is produced by DT. Each extractor and transform also has an output parameter although they are not shown in Figure 6.3. For instance, the output parameter of DT is denoted DT_O. In Figure 6.3, DT_O is used as input by PV and TV. That is, $PV_{DT} = DT_O$ and $TV_{DT} = DT_O$.

Figure 6.3 also shows the properties that hold for each input parameter and each transform. For instance, the properties *map-to-one*, *suffix-safe* and *set-to-seq* hold for input parameter TV_{DT}, and the property *in-det-out* holds for transform TV. In Section 6.4, we define these properties and justify why they hold in this example. We then use them in DR in Section 6.5. When the component DAG is designed, the attributes and keys of input parameters are also declared. For instance, the attributes of PV_{PTE} tuples are $[company, pe]$, while the keys are $[company]$. □

In summary, Y_X denotes the input parameter of component Y produced by component X, and Y_O is the output parameter of Y. We use Attrs(Y_X) to denote the attributes of the Y_X tuples. Similarly, KeyAttrs(Y_X) specifies their keys. W denotes the warehouse inserter.

We note that the component DAGs designed for warehouse creation and maintenance are different. Component DAGs for creation perform the initial population of tables, while component DAGs for maintenance typically populate “delta” tables with changes that are later applied to the tables. (Creating separate delta tables allows standard view maintenance algorithms to be applied to materialized views over these tables.) While a component DAG for creation only handles new tuples, a maintenance component DAG handles tuple inserts, deletes, and updates. Therefore, the components used are usually different. However, our resumption algorithm applies equally well to both creation and maintenance component DAGs. Thus, in the rest of the chapter, when we refer to a "load," it could be for initial warehouse creation or for warehouse maintenance.

6.2.2 Successful Warehouse Load

When a component DAG is used to load data, the extractors produce sequences that serve as inputs to the transforms. That is, each input parameter is "instantiated" with a tuple sequence. Each transform then produces an output sequence that is sent to subsequent components. Finally, the inserter receives a tuple sequence, inserts the tuples in batches, and periodically issues a commit command to ensure that the tuples are stored persistently. Note that each component's output sequence can be received as the next component's input as it is generated, to maximize pipelined parallelism. More specifically, at each point in time, a component Y has produced a prefix of its entire output sequence and shipped the prefix tuples to the next components. The next example illustrates a warehouse load during normal operation, *i.e.*, no failures occur.

EXAMPLE 6.2.2 Consider the component DAG in Figure 6.3. First, extractors fill their output parameters PTE_O and TRD_O with the sequences $\mathcal{PTE}_O$ and $\mathcal{TRD}_O$, respectively. (The calligraphy font denotes sequences.) Input parameter PV_{PTE} is instantiated with the sequence $\mathcal{PV}_{PTE} = \mathcal{PTE}_O$. Similarly, DT_{TRD} is instantiated with $\mathcal{DT}_{TRD} = \mathcal{TRD}_O$, and so on. Note that PTE does not need to produce $\mathcal{PTE}_O$ in its entirety before it can ship a prefix of $\mathcal{PTE}_O$ to PV. Finally, W_{PV} of the inserter is instantiated with $\mathcal{W}_{PV} = \mathcal{PV}_O$. W inserts the tuples in $\mathcal{W}_{PV}$ in order and issues a commit periodically. In the absence of failures, $\mathcal{W}_{PV}$ is eventually stored in the warehouse. □

To summarize our notation, $\mathcal{Y}_X$ and $\mathcal{Y}_O$ denote the sequences used for input parameter Y_X and output parameter Y_O during a warehouse load. When Y produces $\mathcal{Y}_O$ by processing $\mathcal{Y}_X$ (and possibly other input sequences), we say $Y(...\mathcal{Y}_X...) = \mathcal{Y}_O$. We also use $\mathcal{W}$ to denote the sequence that is loaded into the warehouse in the absence of failures.

6.3 Warehouse Load Failure

In general, there are two types of failures that can prevent a load from completing - logical failures (*e.g.*, invalid data) and system-level failures (*e.g.*, RDBMS or software crashes, hardware crashes, lack of disk space). If a load fails because of invalid data, the load will again fail if it is restarted to process the same invalid data. On the other hand, if a load fails because of system-level failures, it is not likely that the load will fail once it is restarted. This low likelihood assumes of course that the necessary actions were taken to fix the failure, *e.g.*, software was restarted, or the hardware was fixed/replaced, or disk space was allocated. In this chapter, we focus on system-level failures. Furthermore, we consider system-level failures that do not affect information stored in stable storage.

6.3.1 Component Failures

Even though various components may fail, the effect of any failure on the warehouse is the same. That is, only a prefix of the normal operation input sequence $\mathcal{W}$ is loaded into the warehouse.

Observation In the event of a failure, only a prefix of $\mathcal{W}$ is stored in the warehouse. $\square$

We now show why this observation holds for each type of component failure. When a source or its extractor E fails, only a prefix of E's normal operation output has been produced. Let transform Y take the output of E as its input. Y therefore receives and processes only part of its normal input and produces only a prefix of its output. Any transform Z that receives Y's output will then produce a prefix of its output, etc. This cascade of incomplete inputs eventually reaches the warehouse inserter W, causing it to insert only a prefix of $\mathcal{W}$.

Similarly, when a transform Y fails, only a prefix of Y's output has been produced. Again, a cascade of incomplete inputs leads to a prefix of $\mathcal{W}$ being stored in the warehouse. Finally, when the warehouse or the inserter W fails, it is clear that only a prefix of $\mathcal{W}$ is inserted and committed by W into the warehouse. (Note that the prefix may be empty.)

A network failure between components Y and Z results in only a prefix of Y's output reaching Z. Therefore, the effect of a network failure is the same as if component Y had failed. Henceforth, we ignore network failures since they can be modeled as failures of other components.

6.3.2 Data for Resumption

When a component Y fails, the warehouse load eventually halts due to lack of input. Once Y recovers, the load can be resumed. However, only limited data is available to the resumption algorithm. Limited data is available because the specific details (*e.g.*, state) of the transforms are not known. The resumption algorithm may use the prefix of the warehouse input $\mathcal{W}$ that is in the warehouse. In addition, the following procedures (and other slight variants) may be provided by each extractor E. We use $\mathcal{E}_O$ to denote the sequence that would have been extracted by E had there been no failures. More details on all of the re-extraction procedures are provided in Section 6.5.3.

- GetAll() extracts the same set of tuples as the set of tuples in $\mathcal{E}_O$. The order of the tuples may be different because many sources, such as commercial RDBMS, do not guarantee the order of the tuples. We assume that all extractors provide GetAll(), that is, that the original data is still available. If $\mathcal{E}_O$ cannot be reproduced, then $\mathcal{E}_O$ must be logged.

- GetAllInorder() extracts the same sequence $\mathcal{E}_O$. This procedure may be supported by an extractor of a commercial RDBMS that initially extracted tuples with an SQL ORDER BY clause. Thus, the same tuple order can be obtained by using the same clause during re-extraction.

- GetSubset(...) provides the $\mathcal{E}_O$ tuples that are not in the subset indicated by GetSubset's parameters. Sources that can selectively filter tuples typically provide GetSubset.

- GetSuffix(...) provides a suffix of $\mathcal{E}_O$ that excludes the prefix indicated by GetSuffix's parameters. Sources that can filter and order tuples typically provide GetSuffix.

In this chapter, we assume that the re-extraction procedures only produce tuples that were in the original sequence $\mathcal{E}_O$. However, our algorithms also work when additional tuples appear only in the suffix of $\mathcal{E}_O$ that was not processed before the failure.

6.3.3 Redoing the Warehouse Load

When the warehouse load fails, only a prefix $\mathcal{C}$ of $\mathcal{W}$ is in the warehouse. The goal of a resumption algorithm is to load the remaining tuples of $\mathcal{W}$, in any order since the warehouse is an RDBMS. The simplest resumption algorithm, called *Redo*, simply repeats the load.

First $\mathcal{C}$ is deleted, and then for each extractor in the component DAG, the re-extraction procedure GetAll() is invoked. *Redo* is shown in Figure 6.4.

Given component DAG, and $\mathcal{C}$ loaded in the warehouse 1. Delete $\mathcal{C}$. 2. For each extractor E in the component DAG   3. Call E.GetAll()

Figure 6.4: *Redo* Algorithm

Although *Redo* is very simple, it still requires that the entire workflow satisfies the property that if the same set of tuples are obtained by the extractors, the same set of tuples are inserted into the warehouse. Since this property pertains to an entire workflow, it can be hard to test. A *singular property* that pertains to a single transform is much easier to test. The following singular property, set-to-set, is sufficient to enable *Redo*. That is, if all extractors use GetAll or GetAllInorder, and all transforms are set-to-set, then *Redo* can be used. This condition is tested in Definition 6.3.1

Property 6.3.1 **(set-to-set(Y))** If (given the same set of input tuples, Y produces the same set of output tuples) then (set-to-set(Y) = true). Otherwise, set-to-set(Y) = false. □

Definition 6.3.1 **(Same-set(Y))** If (Y is an extractor and Y uses GetAllInorder or GetAll during resumption) then (Same-set(Y) = true). Otherwise, if ($\forall Y_X$: Same-set(X) and set-to-set(Y)) then (Same-set(Y) = true). Otherwise, Same-set(Y) = false. □

6.4 Properties for Resumption

Unlike *Redo*, *DR* does not need to reprocess all of the tuples originally extracted from the sources. In this section, we identify singular properties of transforms or input parameters that *DR* combines into "transitive properties" to avoid reprocessing some of the input tuples.

To illustrate, suppose that the sequence $\mathcal{W}_Y$ to be inserted into the warehouse is $[y_1 y_2 y_3]$ (see Figure 6.5) and $[x_1 x_2 x_3 x_4]$ is the $\mathcal{Y}_X$ input sequence that yields the warehouse tuples. An edge $x_i \rightarrow y_j$ in Figure 6.5 indicates that x_i "contributes" in the computation of y_j. (We define contributes formally in Definition 6.4.1.) Also suppose that after a failure, only y_1 is stored in the warehouse. Clearly, it is *safe* to filter $\mathcal{Y}_X$ tuples that contribute only to $\mathcal{W}_Y$ tuples already in the warehouse, in this case, y_1. Thus in Figure 6.5, x_1 and x_2 can be

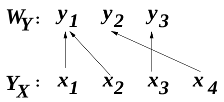

Figure 6.5: Safe Filtering of x_2

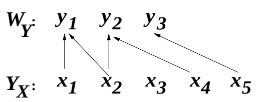

Figure 6.6: Unsafe Filtering of x_2

filtered out. We need to be careful with y_1 contributors that also contribute to other $\mathcal{W}_Y$ tuples. For example, in Figure 6.6, $\{x_1, x_2\}$ again contribute to y_1, but we cannot filter out x_2, since it is still needed to generate y_2.

In general, we need to answer the following questions to avoid reprocessing input tuples:

- *Question (1)*: For a given warehouse tuple, which tuples in $\mathcal{Y}_X$ contribute to it?
- *Question (2)*: When is it safe to filter those tuples from $\mathcal{Y}_X$?

The challenge is that we must answer these questions using limited information. In particular, we can only use the tuples stored in the warehouse before the failure, and the properties, attributes and key attributes declared when the component DAG was designed.

In Section 6.4.1, we identify four singular properties to answer Question (2). We then define three *transitive properties* that apply to sub-DAGs of the component DAG. *DR* will derive the transitive properties based on the declared singular properties. In Section 6.4.2, we define two more singular properties. Using these properties, we define *identifying attributes* of the tuples to answer Question (1). *DR* will derive the identifying attributes based on the declared singular properties and key attributes. In Section 6.7, we present a study that shows that the singular properties hold for many commercial transforms. Since singular properties pertain to a transform or an input parameter and not to a whole workflow, they are easy to grasp and can often be deduced easily from the transform manuals. Henceforth, we refer to singular properties as "properties" for conciseness.

Before proceeding, we formalize the notion of contributing input tuples. An input tuple x_i in an input sequence $\mathcal{Y}_X$ of transform Y *contributes* to a tuple y_j in a resulting output sequence $\mathcal{Y}_\mathcal{O}$ if y_j is only produced when x_i is in $\mathcal{Y}_X$. The definition of "contributes" uses the function IsSubsequence($\mathcal{S}$, $\mathcal{T}$), which returns true if $\mathcal{S}$ is a subsequence of $\mathcal{T}$, and false otherwise.[1]

[1]Given $\mathcal{T} = [t_1..t_n]$ and $\mathcal{S} = [s_1..s_k]$, $\mathcal{S}$ is a subsequence of $\mathcal{T}$ if there exists a strictly increasing sequence $[i_1..i_k]$ of indices of $\mathcal{T}$ such that for all $j = 1, 2, \ldots, k$, $t_{i_j} = s_j$ ([CLR92]).

Definition 6.4.1 (Contributes, Contributors) Given transform Y, let $Y(...\mathcal{Y}_X...) = \mathcal{Y}_O$ and $Y(...\mathcal{Y}'_X...) = \mathcal{Y}'_O$. Also let $\mathcal{Y}_X = [x_1..x_{i-1}x_ix_{i+1}..x_n]$ and $\mathcal{Y}'_X = [x_1..x_{i-1}x_{i+1}..x_n]$.

Contributes(x_i,y_j) = true, if $y_j \in \mathcal{Y}_O$ and $y_j \notin \mathcal{Y}'_O$. Otherwise, Contributes(x_i,y_j) = false.

Contributors($\mathcal{Y}_X$, y_j) = $\mathcal{T}$, where IsSubsequence($\mathcal{T}$, $\mathcal{Y}_X$) and ($\forall x_i \in \mathcal{T}$: Contributes(x_i, y_j)) and ($\forall x_i \in \mathcal{Y}_X$: Contributes(x_i,y_j) $\Rightarrow x_i \in \mathcal{T}$). □

We can extend Definition 6.4.1 in a transitive fashion to define when a tuple contributes to a warehouse tuple. For instance, if a x_i contributes to y_j, which in turn contributes to a warehouse tuple z_k, then x_i contributes to z_k.

Definition 6.4.1 does not consider transforms with non-monotonic input parameters. Informally, Y_X is non-monotonic if the number of output tuples of Y grows when the number of input tuples to Y_X is decreased. For instance, if Y is the difference transform $Y_{X1} - Y_{X2}$, Y_{X2} is non-monotonic. In this chapter, we do not filter input tuples of a non-monotonic input parameter.

Notice that there may be tuples that do not contribute to any output tuple. For instance, if transform Y computes the sum of its input tuples and an input tuple t is $\langle\ 0\ \rangle$, then according to Definition 6.4.1, t does not contribute to the sum unless t is the only input tuple. Tuples like t that do not affect the output are called *inconsequential input tuples*, and are candidates for filtering.

6.4.1 Safe Filtering

During resumption, a transform Y may not be required to produce all of its normal operation output $\mathcal{Y}_O$. Therefore, Y may not need to reprocess some of its input tuples, either. In this section, we identify properties that ensure safe filtering of input tuples.

The *map-to-one* property holds for Y_X whenever every input tuple x_i contributes to at most one Y_O output tuple y_j (as in Figure 6.5). A study presented in Section 6.7 confirms that input parameters of many transforms are map-to-one. For instance, the input parameters of selection, projection, union, aggregation and some join transforms are map-to-one.

Property 6.4.1 (map-to-one(Y_X)) Given transform Y with input parameter Y_X, Y_X is *map-to-one* if $\forall\mathcal{Y}_X, \forall\mathcal{Y}_O, \forall x_i \in \mathcal{Y}_X$: $(Y(...\mathcal{Y}_X...) = \mathcal{Y}_O) \Rightarrow (\neg\exists y_j, y_k \in \mathcal{Y}_O$ such that Contributes(x_i,y_j) and Contributes(x_i,y_k) and $j \neq k$). □

If Y_X is map-to-one, and some of the tuples in $\mathcal{Y}_O$ are not needed, then the corresponding tuples in $\mathcal{Y}_X$ that contribute to them can be safely filtered at resumption time. For example, in Figure 6.5, if the $\mathcal{Y}_O$ output tuples are the tuples being loaded into the warehouse, and tuples y_1 and y_2 are already committed in the warehouse, then the subset $\{x_1, x_2, x_4\}$ of the input tuples does not need to be processed and can be filtered from the Y_X input.

Subset-feasible(Y_X) is a transitive property that states that it is feasible to filter some subset of the Y_X input tuples. If there is a single path[2] from Y_X to the warehouse, Subset-feasible holds when all of the input parameters in the path are map-to-one. In this case, we can safely filter the Y_X tuples that contribute to some warehouse tuple for these Y_X tuples contribute to no other. Similarly, if there are multiple paths from Y_X to the warehouse, each input parameter along any path from Y_X to the warehouse must be map-to-one. If even one of the input parameters in the path(s) is not map-to-one, then we cannot filter any Y_X tuples because each Y_X tuple may contribute to tuples that are not yet in the warehouse.

Definition 6.4.2 (Subset-feasible(Y_X)) Given transform Y with input parameter Y_X, Subset-feasible(Y_X) = true if Y is the warehouse inserter. Otherwise, Subset-feasible(Y_X) = true if Y_X is map-to-one and $\forall Z_Y$: Subset-feasible(Z_Y). Otherwise, Subset-feasible(Y_X) = false. □

While the map-to-one and Subset-feasible properties allow a subset of the input sequence to be filtered, the *suffix-safe* property allows a prefix of the input sequence to be filtered. The suffix-safe property holds when any prefix of the output can be produced by some prefix of the input sequence. Moreover, any suffix of the output can be produced from some suffix of the input sequence. For instance, the input parameters of transforms that perform selection, projection, union, and aggregation over sorted input are likely to be suffix-safe (see Section 6.7).

Property 6.4.2 (suffix-safe(Y_X)) Given $\mathcal{T} = [t_1..t_n]$, let First($\mathcal{T}$) = t_1, Last($\mathcal{T}$) = t_n, and $t_i \leq_{\mathcal{T}} t_j$ if t_i is before t_j in $\mathcal{T}$ or $i = j$. Given transform Y with input parameter Y_X, Y_X is *suffix-safe* if $\forall \mathcal{Y}_X, \forall \mathcal{Y}_O, \forall y_j, y_{j+1} \in \mathcal{Y}_O$: $(Y(...\mathcal{Y}_X...) = \mathcal{Y}_O) \Rightarrow$ (Last(Contributors($\mathcal{Y}_X$, y_j)) $\leq_{\mathcal{Y}_X}$ First(Contributors($\mathcal{Y}_X$, y_{j+1}))). □

[2]Formally, a path P in a component DAG is a sequence of edges where each pair of consecutive edges E_iE_j represents the input and output parameters Y_X and $Y_O = Z_Y$ of a transform Y. If P is composed of one edge, the edge must represent W_X, where X is the extractor that feeds the inserter W.

Figure 6.6 illustrates conceptually how suffix-safe can be used. If only $[y_3]$ of $\mathcal{Y}_O$ in Figure 6.6 needs to be produced, processing the suffix $[x_5]$ of $\mathcal{Y}_X$ will produce $[y_3]$. Conversely, if $[y_1 y_2]$ does not need to be produced, the prefix $[x_1 x_2 x_3 x_4]$ can be filtered from Y_X at resumption time. Notice that when the suffix-safe property is used, tuples like x_3 that do not contribute to any output tuple can be filtered. Filtering such tuples is not possible using the map-to-one property.

Prefix-feasible(Y_X) is a transitive property that states that it is feasible to filter some prefix of the Y_X input sequence. This property is true if all of the input parameters from Y_X to the warehouse are suffix-safe. (The reasoning is similar to that for Subset-feasible(Y_X) and map-to-one.)

Definition 6.4.3 (Prefix-feasible(Y_X)) Given transform Y with input parameter Y_X, Prefix-feasible(Y_X) = true if Y is the warehouse inserter. Otherwise, Prefix-feasible(Y_X) = true if Y_X is suffix-safe and $\forall Z_Y$: Prefix-feasible(Z_Y). Otherwise, Prefix-feasible(Y_X) = false. □

Filtering a prefix of the Y_X input sequence is possible only if Y_X receives the same sequence during load resumption as it did during normal operation. For instance, in Figure 6.6, even if Prefix-feasible(Y_X) holds we cannot filter out any prefix of the Y_X input if the input sequence is $[x_5 x_4 x_3 x_2 x_1]$ during resumption. We now define some properties that guarantee that an input parameter Y_X receives the same sequence at resumption time.

We say that a transform Y is *in-det-out* if Y produces the same output sequence $\mathcal{Y}_O$ whenever it processes the same input sequences. We expect most transforms to satisfy this property.

Property 6.4.3 (in-det-out(Y)) Transform Y is *in-det-out* if Y produces the same output sequence whenever it processes the same input sequences. □

The in-det-out property guarantees that if a transform X and all of the transforms preceding X are in-det-out, and the data extractors produce the same sequences at resumption time, then X will produce the same sequence, too. Hence, Y_X receives the same sequence.

The requirement that all of the preceding transforms are in-det-out can be relaxed if some of the input parameters are *set-to-seq*. That is, if the order of the tuples in Y_X does not affect the order of the output tuples in Y_O, then Y_X is set-to-seq. For example, if the sequence $[z_1 z_2 z_3 z_4]$ is produced by a sorting transform Z, then as long as Z processes the same set of tuples, $[z_1 z_2 z_3 z_4]$ is produced as output.

Property 6.4.4 (set-to-seq(Y_X)) Given transform Y with input parameter Y_X, Y_X is *set-to-seq* if (Y is in-det-out) and ($\forall \mathcal{Y}_X, \mathcal{Y}'_X$: (($\mathcal{Y}_X$ and $\mathcal{Y}'_X$ have the same set of tuples) and (all other input parameters of Y receive the same sequence)) $\Rightarrow Y(...\mathcal{Y}_X...) = Y(...\mathcal{Y}'_X...)$). □

Same-seq(Y_X) is a transitive property that holds if Y_X is guaranteed to receive the same sequence at resumption time. Same-seq(Y_X) is true if the transforms and input parameters that precede Y_X satisfy the in-det-out or set-to-seq property, respectively. Same-seq(Y_X) guarantees that Y_X receives the same input sequence. A weaker guarantee that sometimes allows for prefix filtering is that Y_X receives a suffix of the normal operation input $\mathcal{Y}_X$. We do not develop this weaker guarantee here.

Definition 6.4.4 (Same-seq(Y_X)) If X is an extractor then Same-seq(Y_X) = true if X uses the GetAllInorder re-extraction procedure. Otherwise, Same-seq(Y_X) = true if X is in-det-out and $\forall X_V$: (Same-seq(X_V) or (X_V is set-to-seq and Same-set(V))). Otherwise, Same-seq(Y_X) = false. □

6.4.2 Identifying Contributors

To determine which $\mathcal{Y}_X$ tuples contribute to a warehouse tuple w_k, we are only provided with the value of w_k after the failure. Since transforms are black boxes, the only way to identify the contributors to w_k is to match the attributes that the $\mathcal{Y}_X$ tuples and w_k have in common. (If a transform changes an attribute value, *e.g.*, reorders the bytes of a key attribute, we assume that it also changes the attribute name.)

We now define properties that, when satisfied, guarantee that we can identify exactly the $\mathcal{Y}_X$ contributors to w_k by matching certain *identifying attributes*, denoted IdAttrs(Y_X). In practice, some inconsequential $\mathcal{Y}_X$ input tuples may also match w_k on IdAttrs(Y_X). However, these tuples can be safely filtered since they do not contribute to the output. If the contributors cannot be identified by matching attributes, IdAttrs(Y_X) is set to [ ].

We define the *no-hidden-contributor* property to hold for Y_X if all of the $\mathcal{Y}_X$ tuples that contribute to some output tuple y_j match y_j on Attrs(Y_X) $\cap$ Attrs(Y_O). Selection, projection, aggregation, and union transforms have input parameters with no hidden contributors.

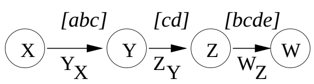

Figure 6.7: Example Component DAG

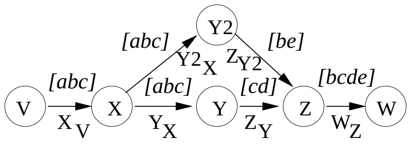

Figure 6.8: Component DAG with Replicated Outputs

The input parameters of many join transforms also do not have hidden contributors. We show later in Section 6.7 that many commercial transforms have input parameters with no hidden contributors.

Property 6.4.5 (**no-hidden-contributor**(Y_X)) Given transform Y with input parameter Y_X, *no-hidden-contributors*(Y_X) if $\forall \mathcal{Y}_X$, $\forall \mathcal{Y}_O$, $\forall y_j \in \mathcal{Y}_O$, $\forall x_i \in$ Contributors($\mathcal{Y}_X$, y_j), $\forall a \in$ (Attrs(Y_X) $\cap$ Attrs(Y_O)): $(Y(...\mathcal{Y}_X...) = \mathcal{Y}_O) \Rightarrow (x_i.a = y_j.a)$. □

If Y_X has no hidden contributors, we can identify a set of input tuples that contains all of the contributors to an output tuple y_j. This set is called the *potential contributors* of y_j. Shortly, we will use keys and other properties to verify that the set of potential contributors of y_j contains only tuples that do contribute to y_j. For now, we illustrate how the potential contributors are found.

EXAMPLE 6.4.1 Consider the component DAG shown in Figure 6.7. The labels below the edges, *e.g.*, Z_Y, identify the input parameter, and the labels above the edges give the attributes of the input tuples, *e.g.*, Attrs(Z_Y) = [cd]. If Z_Y has no hidden contributors, then all of the $\mathcal{Z}_Y$ contributors to a warehouse tuple w_k, denoted S_k, match w_k on [cd] (*i.e.*, Attrs(Z_Y) $\cap$ Attrs(Z_O)). If Y_X has no hidden contributors, then all of the $\mathcal{Y}_X$ contributors to $z_i \in S_k$ match z_i on [c] (*i.e.*, Attrs(Y_X) $\cap$ Attrs(Y_O)). Since all of the tuples in S_k have the same c attribute (*i.e.*, the c attribute of w_k), all of the $\mathcal{Y}_X$ tuples that contribute to w_k match w_k on [c]. Hence, all of the potential contributors of w_k in $\mathcal{Y}_X$ are the ones that match w_k on [c]. □

We call attributes that identify the $\mathcal{Y}_X$ potential contributors, the *candidate identifying attributes* or candidate attributes (CandAttrs) of Y_X. The formal definition of CandAttrs applies to an input parameter Y_X and a path P from Y_X to the warehouse.

Definition 6.4.5 (**CandAttrs(Y_X,P)**) Let P be a path from input parameter Y_X to the warehouse. There are three possibilities for CandAttrs(Y_X,P):

1. If Y is the warehouse inserter, then CandAttrs(Y_X, P) = Attrs(Y_X).
2. If Y_X has hidden contributors then CandAttrs(Y_X, P) = [ ].
3. Else CandAttrs(Y_X, P) = CandAttrs(Z_Y, P') $\cap$ Attrs(Y_X), where $P = [Y_X Z_Y..W_I]$, and P' is P excluding Y_X.

$\square$

In summary, CandAttrs(Y_X, P) is just the attributes that are present throughout the path P starting from Y_X, unless one of the input parameters in P has hidden contributors. If so, then CandAttrs(Y_X, P) is set to [ ] implying that all $\mathcal{Y}_X$ tuples are potential contributors.

Since the potential contributors identified by CandAttrs(Y_X, P) may include tuples that do not contribute to w_k, we would like to verify that all the potential contributors do contribute to w_k. To do so, we need to use key attributes. The *no-spurious-output* property may also be used to verify contributors. We define the no-spurious-output property to hold for transform Y if each output tuple y_j has at least one contributor from each input parameter Y_X. While this property holds for many transforms (see Section 6.7), union transforms do not satisfy it.

Property 6.4.6 (**no-spurious-output(Y)**) A transform Y produces *no spurious output* if $\forall$ input parameters Y_X, $\forall \mathcal{Y}_X$, $\forall \mathcal{Y}_O$, $\forall y_j \in \mathcal{Y}_O$: $(Y(...\mathcal{Y}_X...) = \mathcal{Y}_O) \Rightarrow$ (Contributors($\mathcal{Y}_X$,y_j) $\neq$ [ ]). $\square$

We now illustrate in the next example how key attributes, candidate attributes, and the no-spurious-output property combine to determine the identifying attributes.

EXAMPLE 6.4.2 Consider the component DAG shown in Figure 6.7. Note that CandAttrs(Y_X, P) = $[c]$ where $P = [Y_X Z_Y W_Z]$, assuming that Y_X, Z_Y, and W_Z have no hidden contributors. Now consider which attributes can be used as IdAttrs(Y_X). There are three possibilities.

1. IdAttrs(Y_X) = KeyAttrs(Y_X) if KeyAttrs(Y_X) $\subseteq$ CandAttrs(Y_X, P) and both Y and Z satisfy the no-spurious-output property.

2. IdAttrs(Y_X) = KeyAttrs(W_Z) if KeyAttrs(W_Z) $\subseteq$ CandAttrs(Y_X, P).

3. IdAttrs(Y_X) = IdAttrs(Z_Y) if IdAttrs(Z_Y) $\subseteq$ CandAttrs(Y_X, P).

To illustrate the first possibility, suppose KeyAttrs(Y_X) is $[c]$. If $w_k.c = 1$, any $\mathcal{Y}_X$ tuple that contributes to w_k must have $c = 1$ since CandAttrs(Y_X, P) = $[c]$. Since neither Y nor Z has spurious output tuples, there is at least one $\mathcal{Y}_X$ tuple that contributes to w_k. Because c is the key for Y_X, the $\mathcal{Y}_X$ tuple with $c = 1$ must be the contributor.

To illustrate the second possibility, suppose KeyAttrs(W_Z) = $[c]$. If $w_k.c = 1$, any $\mathcal{Y}_X$ tuple that contributes to w_k must have $c = 1$ since CandAttrs(Y_X, P) = $[c]$. All $\mathcal{Y}_X$ tuples with $c = 1$ must contribute to either w_k or to no warehouse tuples since c is the key of W_Z.

To illustrate the third possibility, suppose IdAttrs(Z_Y) = $[c]$. Then given a warehouse tuple w_k with $w_k.c = 1$, we can identify the $\mathcal{Z}_Y$ contributors to w_k, denoted S_k, by matching their c attribute with 1. Since Y_X has no hidden contributors (because CandAttrs(Y_X,P) $\neq$ [ ]), a $\mathcal{Y}_X$ tuple with $c = 1$ must contribute to a tuple $z_j \in S_k$ or to no tuple in $\mathcal{Z}_Y$. Hence, we can identify exactly the $\mathcal{Y}_X$ contributors to w_k by matching their c attribute values.

In summary, the key attributes of Y_X, Z_Y (or any other input parameter in the path from Y_X to W_Z), or W_Z can serve as IdAttrs(Y_X). These key attributes must be a subset of CandAttrs(Y_X, P) to ensure that the matching can be performed between the warehouse tuples and the $\mathcal{Y}_X$ tuples. □

The previous example provides the intuition behind our definition of the identifying attributes of Y_X. The following definition gives the identifying attributes of Y_X along path P. If there is a single path P from Y_X to the warehouse, IdAttrs(Y_X) = IdAttrsPath(Y_X,P).

Definition 6.4.6 (IdAttrsPath(Y_X,P), IdAttrs(Y_X)) Let P be the only path from Y_X to the warehouse. There are three possibilities for IdAttrsPath(Y_X,P) (*i.e.*, IdAttrs(Y_X)).

1. If (KeyAttrs(Y_X) $\subseteq$ CandAttrs(Y_X, P) and $\forall Z_V \in P : Z_V$ has no spurious output tuples),
   then (IdAttrsPath(Y_X, P) = KeyAttrs(Y_X)).

2. Otherwise, let $Z_V \in P$ but $Z_V \neq Y_X$. Let P' be the path from Z_V to the warehouse.
   If (IdAttrsPath(Z_V,P') $\neq$ [ ] and IdAttrsPath(Z_V,P') $\subseteq$ CandAttrs(Y_X, P)),
   then (IdAttrsPath(Y_X,P) = IdAttrsPath(Z_V,P')).

3. Otherwise IdAttrsPath(Y_X,P) = [ ].

□

Case (1) in Definition 6.4.6 uses the key attributes of Y_X as IdAttrs(Y_X). Case (2) in Definition 6.4.6 encompasses the second and third possibilities illustrated in Example 6.4.2. That is, for each input parameter in P, it checks if the IdAttrs of that input parameter can be used as IdAttrs(Y_X). Notice that there may be more than one input parameter in P whose identifying attributes can be used for IdAttrs(Y_X). We revisit this issue shortly.

We now modify IdAttrs to handle the general case where there are multiple paths from Y_X to the warehouse. The next example provides the intuition behind the generalization of IdAttrs.

EXAMPLE 6.4.3 Consider the component DAG shown in Figure 6.8, where there are two paths from X_V to the warehouse: $P_1 = [X_V Y_X Z_Y W_Z]$ and $P_2 = [X_V Y2_X Z_{Y2} W_Z]$. We want to determine IdAttrs(X_V). Suppose IdAttrs(X_V, P_1) is $[b]$, while IdAttrs(X_V,P_2) is $[c]$. Then, given a warehouse tuple w_k, the $\mathcal{X}_V$ contributors along P_1 match w_k on $[b]$, while the $\mathcal{X}_V$ contributors along P_2 match w_k on $[c]$. Hence, the $\mathcal{X}_V$ contributors match w_k on $[b]$ or $[c]$, denoted $[b] \vee [c]$. □

Example 6.4.3 shows that if there are multiple paths, IdAttrs(Y_X) is the disjunction of the identifying attributes of the individual paths.

Definition 6.4.7 (IdAttrs(Y_X)) Let $\{P\}$ be the set of all paths from Y_X to the warehouse input parameter.
If $\exists P \in \{P\}$ such that IdAttrsPath(Y_X, P) $= [\,]$, then IdAttrs(Y_X) $= [\,]$.
Otherwise, IdAttrs(Y_X) $= \bigvee_{P \in \{P\}}$ IdAttrsPath(Y_X, P).
□

Although we provide a general definition for IdAttrs(Y_X), in most cases there is a single path from Y_X to the warehouse. Even when there are multiple paths from Y_X to the warehouse, we can simplify IdAttrs(Y_X) as follows: Given IdAttrs(Y_X) $= A_1 \vee \ldots \vee A_n$, we eliminate A_i if $\exists A_j \subseteq A_i$, because any contributor identified by A_i is also identified by A_j.

This method for simplifying IdAttrs(Y_X) is also a guide for choosing the identifying attributes. When we developed IdAttrs(Y_X) for a single path P (Definition 6.4.6), we did not specify how to choose the input parameter along P whose IdAttrs will be used for IdAttrs(Y_X). It is best to choose an input parameter that appears on many paths from

Y_X to the warehouse. The input parameter of the inserter W is a prime candidate since it appears on all paths from Y_X.

Transform	*Properties*	*Function computed by transform*
DT	in-det-out no-spurious-output	`select` $*$ `from` DT_{TRD} `where` $date \geq 12/1/98$ `and` $date \leq 12/31/98$
TV	in-det-out no-spurious-output	`select sum(`$volume$`) as` $totalvol$ `from` TV_{DT}
PV	in-det-out no-spurious-output	`select` $PV_{PTE}.company, PV_{PTE}.pe,$ `sum(`$PV_{DT}.volume) * 100/PV_{TV}.totalvol$ `as` $percentvol$ `from` $PV_{PTE}, PV_{DT}, PV_{TV}$ `where` $PV_{PTE}.company = PV_{DT}.company$ `and` $PV_{PTE}.pe \leq 4$ `group by` $PV_{PTE}.company, PV_{PTE}.pe$

Table 6.1: Properties and Functions of Transforms.

Input Y_X	*Attrs(*Y_X*)*	*KeyAttrs(*Y_X*)*	Y_X *Properties*	*IdAttrs(*Y_X*)*	Y_X *Transitive Properties*
DT_{TRD}	[date,company, volume]	[date,company]	map-to-one suffix-safe	[ ]	-
TV_{DT}	[date,company, volume]	[date,company]	map-to-one suffix-safe set-to-seq	[ ]	Prefix-feasible
PV_{PTE}	[company,pe]	[company]	map-to-one suffix-safe	[company]	Subset-feasible, Prefix-feasible
PV_{DT}	[date,company, volume]	[date,company]	map-to-one set-to-seq	[company]	Subset-feasible
PV_{TV}	[totalvol]	[totalvol]	suffix-safe set-to-seq	[ ]	Prefix-feasible
W_{PV}	[company,pe, percentvol]	[company]	map-to-one suffix-safe	[company]	Subset-feasible, Prefix-feasible

Table 6.2: Declared and Inferred Properties of Input Parameters.

6.4.3 The Trades Example Revisited

We now return to our main example, shown in Figure 6.3, and illustrate the properties satisfied by the input parameters and transforms.

Table 6.1 shows the functions computed by the transforms. Although these function definitions cannot be used by the resumption algorithms, we include them here to help explain why the properties hold. We show SQL functions for simplicity even though transforms often perform functions that cannot be written in SQL. Table 6.1 also shows that all three transforms are declared to be in-det-out since they produce the same output sequence given the same input sequences.

The first four columns of Table 6.2 show the attributes, keys, and properties declared for each input parameter when the component DAG is designed. We now explain why the properties hold. DT reads each tuple in DT_{TRD} and only outputs the tuple if it has a date in December 1998. Therefore, DT_{TRD} is suffix-safe, since DT outputs tuples in the input tuple order. It is map-to-one, since each input tuple contributes to zero or one output tuple. It is not set-to-seq, since a different order of input tuples will produce a different order of output tuples.

Transform TV reads all of its input before producing one output tuple. TV_{DT} is trivially map-to-one, suffix-safe, and set-to-seq.

Transform PV reads each tuple in PV_{PTE} and if its pe attribute is ≤ 4, it finds all of the trade tuples for the same company in PV_{DT}, which are probably not in order by company. It computes the percent of the total trade volume using the trade tuples and PV_{TV} and outputs a tuple. Then it processes the next tuple in PV_{PTE}. PV_{PTE} is map-to-one since each tuple contributes to zero or one output tuple, depending on its value for the attribute pe. It is not set-to-seq for the same reason it is suffix-safe: PV processes tuples from PV_{PTE} one at a time, in order. PV_{DT} is map-to-one since each trade tuple contributes to the percent volume tuple of only one company. However, PV_{DT} is not suffix-safe, *e.g.*, the trade tuple needed to join with the first tuple in PV_{PTE} may be the last tuple in PV_{DT}. Similarly, it is set-to-seq because the order of trades tuples is not relevant to PV. PV_{TV} is not map-to-one since the lone PV_{TV} input tuple containing the total volume contributes to all of the output tuples. PV_{TV} is trivially suffix-safe and set-to-seq.

Finally, since the warehouse inserter simply stores its input tuples in order, W_{PV} is map-to-one and suffix-safe but not set-to-seq.

The last two columns of Table 6.2 show the identifying attributes and the transitive properties. We assume that none of the input parameters have hidden contributors. The identifying attribute of W_{PV}, PV_{DT}, and PV_{PTE} is [*company*] because it is the key of W_{PV}. Since none of the attributes of PV_{TV} are preserved in the warehouse, we cannot possibly identify the contributing $\mathcal{PV}_{TV}$ tuples, and IdAttrs(PV_{TV}) is set to [ ]. As a result, IdAttrs(TV_{DT}) = IdAttrs(DT_{TRD}) = [ ]. The transitive properties (*e.g.*, Subset-feasible) are computed using Definitions 6.4.2 and 6.4.3. Note that Same-seq and Same-set are not computed since the re-extraction procedures have not been determined.

6.4.4 Practical Issues

The properties that we have introduced hold in many cases. In Section 6.7, we present a thorough study of a commercial load package to support this claim. The properties are also fairly simple. In fact, some commercial load packages [Sag98] already declare whether some of the properties (*e.g.*, suffix-safe) hold for their transforms. Even if the properties are not declared, they can often be deduced easily from the transform specifications or manuals. Moreover, the properties focus on a single transform and not the whole component DAG, which makes them easy to grasp. (The transitive properties are derived by the *DR* algorithm.)

6.5 The *DR* Resumption Algorithm

We now present the *DR* resumption algorithm, which uses the properties developed in the previous section. *DR* is actually composed of two algorithms, *Design* and *Resume*, hence the name. After a component DAG G is designed, *Design* constructs a component DAG G' that *Resume* will employ to resume any failed warehouse load that used G. The component DAG G' is the same as G except for the following differences.

1. Re-extraction procedures are assigned to the extractors in G'.

2. Filters are assigned to some of the input parameters in G'.

The component DAG G' is constructed by *Design* based solely on the attributes, keys, and properties declared for G. When a warehouse load that uses G fails, *Resume* initializes the filters and re-extraction procedures in G' based on the tuples that were stored in the warehouse. *Resume* then uses G' to resume the warehouse load. Since neither *Design* nor *Resume* runs during normal operation, *DR* does not incur any normal operation overhead!

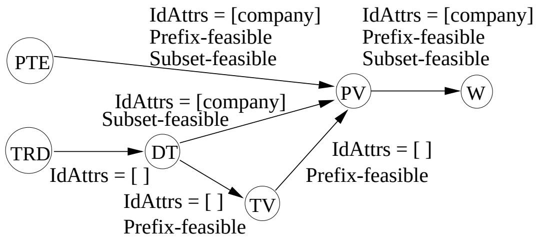

Figure 6.9: Identifying Attributes and Transitive Properties

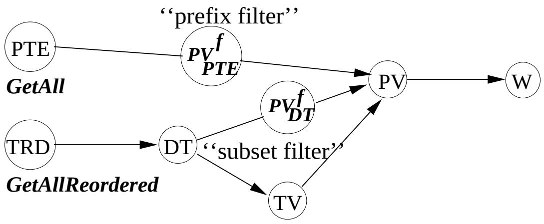

Figure 6.10: Re-extraction Procedures and Filters Assigned

6.5.1 Example using *DR*

To illustrate the overall operation of *DR*, we return to our running example (Figure 6.3). After this illustration, we cover *DR* in more detail. Algorithm *Design* of *DR* first computes the Subset-feasible and Prefix-feasible transitive properties, as well as the IdAttrs of each input parameter. We computed these transitive properties and identifying attributes in Section 6.4.3, and the results are shown in Figure 6.9.

Design then constructs G' by first assigning re-extraction procedures to extractors based on the computed properties and identifying attributes. Since IdAttrs(PV_{PTE}) = [$company$], it is possible to identify source PE tuples that contribute to tuples in the warehouse based on the *company* attribute. Since Prefix-feasible(PV_{PTE}) holds, *DR* can assign GetSuffix to PTE to avoid re-extracting all the PE tuples over again. Also, since Subset-feasible(PV_{PTE}) holds, *DR* can alternatively assign GetSubset to PTE to avoid re-extracting all the PE tuples. However, suppose PTE supports neither GetSuffix nor GetSubset. GetAllInorder is assigned to PTE instead.

IdAttrs(DT_{TRD}) is empty, implying that it is not possible to identify the Trades tuples that contribute to warehouse tuples. Hence, assuming TRD does not support GetAllInorder, only the re-extraction procedure GetAll can be assigned to TRD.

For each input parameter, *Design* then chooses whether to discard a prefix of the input ("prefix filter"), or to discard a subset of the input ("subset filter"). Since discarding a prefix requires the Same-seq property, *Design* computes the Same-seq property as it assigns filters to input parameters. As a result, the input parameters are processed in topological order because the Same-seq property of an input parameter depends on the Same-seq properties of previous input parameters.

1. Same-seq(DT_{TRD}) does not hold because TRD is assigned GetAll, so it is not possible to filter a prefix of the DT_{TRD} input sequence. Furthermore, since DT_{TRD} is not Subset-feasible, a subset filter cannot be assigned.

2. Same-seq(PV_{PTE}) holds because PTE is assigned GetAllInorder. Therefore, PV_{PTE} is both Prefix-feasible and Same-seq, so it is possible to filter a prefix of the PV_{PTE} input sequence. Furthermore, we can identify the contributors to the warehouse tuples based on IdAttrs(PV_{PTE}) = [*company*]. Thus, a filter, denoted PV^f_{PTE}, that removes a prefix of the PV_{PTE} input sequence is assigned to PV_{PTE}. When a failed load is resumed, PV^f_{PTE} removes the prefix of the PV_{PTE} input sequence that ends with the tuple whose *company* attribute matches the last warehouse tuple.

3. TV_{DT} is Prefix-feasible but we cannot identify the contributors of the warehouse tuples since IdAttrs(TV_{DT}) = [ ]. Furthermore Same-seq(TV_{DT}) does not hold since Same-seq(DT_{TRD}) does not hold. No filter is assigned to TV_{DT}.

4. PV_{DT} is Subset-feasible and IdAttrs(PV_{DT}) = [*company*], so a subset filter PV^f_{DT} is assigned to PV_{DT}. Same-seq(PV_{DT}) does not hold, but the subset filter PV^f_{DT} does not require it. When a failed load is resumed, PV^f_{DT} removes all tuples in the PV_{DT} sequence whose *company* attribute value matches some warehouse tuple.

5. IdAttrs(PV_{TV}) is [ ], so no filter is assigned to PV_{TV}. Note that Same-seq(PV_{TV}) holds since TV_{DT} is set-to-seq.

6. Finally, Same-seq(W_{PV}) cannot hold since the filters assigned to PV_{PTE} and PV_{DT} make it impossible for W_{PV} to receive the same sequence. A subset filter can be assigned to W_{PV} since W_{PV} is Subset-feasible. However, *Design* determines that this filter is redundant with the previous filters. Therefore, no filter is assigned to W_{PV}.

The component DAG G' constructed from G is shown in Figure 6.10. Note that G' is constructed using just two "passes" over G: a backward pass to compute IdAttrs, Prefix-feasible, Subset-feasible, and a forward pass to compute Same-seq. Hence, the time to construct G', which is in the order of seconds or minutes, is negligible compared to the time to design and debug G, which is in the order of days or weeks ([Tec]). Algorithm *Design* is now done. Until a failed load by G is resumed, G' is not used.

Suppose that a load using G fails, and the tuple sequence that made it into the warehouse is

$$\mathcal{C} \;=\; [\; \langle AAA, 3, 0.25\rangle \; \langle INTC, 2, 0.98\rangle \; \langle MSN, 4, 0.456\rangle \;],$$

where the three attributes are *company*, *pe*, and *percentvol*, respectively. Based on $\mathcal{C}$, *Resume* instantiates the filters and re-extraction procedures (*i.e.*, GetSuffix, GetSubset) that are sensitive to $\mathcal{C}$. Since only GetAllInorder and GetAll are assigned in our example, only the filters are affected by $\mathcal{C}$.

Subset filter PV^f_{DT} is instantiated to remove any PV_{DT} input tuple whose company is either AAA, $INTC$ or MSN. It is safe to filter these tuples since it is guaranteed that they contribute to at most one warehouse tuple (*i.e.*, Subset-feasible(PV_{DT})), and that tuple is in $\mathcal{C}$. Similarly, prefix filter PV^f_{PTE} is instantiated to remove the prefix of its PV_{PTE} input that ends with the tuple whose *company* attribute is MSN. It is safe to filter these tuples since it is guaranteed that PV has processed through the MSN tuple of PV_{PTE} (*i.e.*, Prefix-feasible(PV_{PTE})). Note that the tuples before the MSN tuple may include ones that do not contribute to any warehouse tuple (*i.e.*, because their *pe* attribute is too high). Once the filters and re-extraction procedures are instantiated, the warehouse load is resumed by calling the re-extraction procedures of G'. Because of the filters, the input tuples that contribute to the tuples in $\mathcal{C}$ are filtered and are not processed again by PV and W. Had the load failed with a longer warehouse tuple sequence $\mathcal{C}$, the filters would have been instantiated appropriately by DR to filter more input tuples.

We conclude the example by contrasting the recovery performed by DR with other methods.

- Unlike *Redo*, DR avoids re-processing many of the input tuples using filters PV^f_{DT} and PV^f_{PVE}. Also, had the extractors PTE and TRD supported GetSubset or GetSuffix, DR could have even avoided re-extracting tuples from the sources.

- *DR* avoids re-processing many input tuples without having to identify batches. Recall that for our example component DAG (Figure 6.9), batches cannot be formed due to the *TotalVolume* (TV) transform. Since batches cannot be formed, a recovery algorithm based on batching input tuples would redo the entire warehouse load.

- During normal operation, the designed component DAG G (Figure 6.9) is used. No normal operation overhead is incurred unlike recovery algorithms based on savepoints or snapshots. Again, the time it takes to construct G' from G is very small compared to the time it takes to design and debug G. Furthermore, this overhead occurs when G is designed, and does not occur during the normal operation of the load.

6.5.2 Filters

In the previous example, we mentioned subset filters and prefix filters. More specifically, there are two types of subset filters and two types of prefix filters that may be assigned to Y_X. In each case, the filter receives X's output sequence as input, and the filter sends its output to Y as the Y_X input sequence.

Clean-Prefix Filter

The clean-prefix filter, $CP[s, A]$, is instantiated with a tuple s and a set of attributes A. CP discards tuples from its input sequence until it finds a tuple t that matches s on A. CP discards t, and continues discarding until an input tuple t' does *not* match s on A. All tuples starting with t' are output by CP. We use CP on Y_X when Y_X is Subset-feasible, Prefix-feasible, and Same-seq, and IdAttrs(Y_X) is not empty. In this case, all input tuples up to and including the contributors of the last $\mathcal{C}$ tuple, denoted Last($\mathcal{C}$), can be safely filtered. So CP is instantiated as $CP[\text{Last}(\mathcal{C}), \text{IdAttrs}(Y_X)]$, where $\mathcal{C}$ is the tuple sequence in the warehouse after the crash. We call CP a clean filter because no $\mathcal{C}$ contributors emerge from it.

Dirty-Prefix Filter

The dirty-prefix filter, $DP[s, A]$, is a slight modification to the clean-prefix filter. DP discards tuples from its input sequence until it finds a tuple t that matches s on A. All tuples starting with t are output by DP. We use DP on Y_X when Y_X is Prefix-feasible, and Same-seq, and IdAttrs(Y_X) is not empty. In this case, all input tuples preceding the contributors of Last($\mathcal{C}$) can be safely filtered. So CP is instantiated as $CP[\text{Last}(\mathcal{C}), \text{IdAttrs}(Y_X)]$.

Clean-Subset Filter

The clean-subset filter, $CS[\mathcal{S}, A]$, is instantiated with a tuple sequence $\mathcal{S}$ and a set of attributes A. For each tuple t in its input sequence $\mathcal{I}$, if t does not match any $\mathcal{S}$ tuple on the A attributes, then t is output. Otherwise, t is discarded. In other words, CS performs an anti-semijoin between $\mathcal{I}$ and $\mathcal{S}$ ($\mathcal{I}\overline{\ltimes}_A\mathcal{S}$). We use CS on Y_X when Y_X is Subset-feasible and IdAttrs(Y_X) is not empty. CS is instantiated as $CS[\mathcal{C}, \text{IdAttrs}(Y_X)]$.

Dirty-Subset Filter

The dirty-subset filter, $DS[\mathcal{C}, \text{IdAttrs}(Y_X)]$, is a slight modification to the clean-subset filter. DP is assigned to Y_X when Y_X is Prefix-feasible and IdAttrs(Y_X) is not empty. Unlike CS, DS removes a suffix $\mathcal{C}_s$ of $\mathcal{C}$ before performing the anti-semijoin. $\mathcal{C}_s$ contains all the tuples that share Y_X contributors with Last($\mathcal{C}$). This suffix can be obtained easily by matching all the $\mathcal{C}$ tuples with the Last($\mathcal{C}$) tuple on IdAttrs(Y_X). After $\mathcal{C}_s$ is obtained, a prefix of $\mathcal{C}$, denoted $\mathcal{C}_p$, is obtained by removing $\mathcal{C}_s$ from $\mathcal{C}$. $\mathcal{C}_s$ is removed since we cannot filter the contributors to $\mathcal{C}_s$ because Y_X is not required to be Subset-feasible. DP then acts like the clean-subset filter $CS[\mathcal{C}_p, \text{IdAttrs}(Y_X)]$.

Assigning the Filters

In summary, the properties that hold for an input parameter Y_X determine the types of filters that can be assigned to Y_X. When more than one filter type can be assigned, we assign the filter that removes the most input tuples. When filter type f removes more tuples than g, we say $f \succ g$. The relationships among the filter types we have introduced are as follows.

$$CP \succ DP \succ DS, \quad CP \succ CS \succ DS$$

Hence, we try to assign the clean-prefix filter first, and the dirty-subset filter last. In *DR*, we assign the dirty-prefix filter before the clean-subset filter for two reasons. First, it is much cheaper to match each input tuple to a single filter tuple s than to a sequence of tuple filters $\mathcal{S}$. Second, the prefix filters can remove tuples that do not contribute to any warehouse tuple, simply because they precede a contributing tuple. The subset filters can only remove contributors. The second advantage is especially apparent in our experimental results in Section 6.7.

The procedure *AssignFilter* is shown in Figure 6.11. Observe that *AssignFilter* assigns a filter to Y_X whenever possible. Since some of these filters may be redundant with previous filters, *Design* uses a subsequent procedure to remove redundant filters.

Algorithm 6.5.1 *AssignFilter*
Input: Component DAG G'; input parameter Y_X
Output: Input parameter Y_X in G' is assigned a filter whenever possible
1. If Prefix-feasible(Y_X) and Subset-feasible(Y_X) and Same-seq(Y_X) and IdAttrs(Y_X) $\neq$ [ ]
   2. Insert $Y_X^f = CP[\text{Last}(\mathcal{C}),\text{IdAttrs}(Y_X)]$ between Y and X in G'
3. Else If Prefix-feasible(Y_X) and Same-seq(Y_X) and IdAttrs(Y_X) $\neq$ [ ]
   4. Insert $Y_X^f = DP[\text{Last}(\mathcal{C}), \text{IdAttrs}(Y_X)]$ between Y and X in G'
5. Else if Subset-feasible(Y_X) and IdAttrs(Y_X) $\neq$ [ ]
   6. Insert $Y_X^f = CS[\mathcal{C}, \text{IdAttrs}(Y_X)]$ between Y and X in G'
7. Else if Prefix-feasible(Y_X) and IdAttrs(Y_X) $\neq$ [ ]
   8. Insert $Y_X^f = DS[\mathcal{C}, \text{IdAttrs}(Y_X)]$ between Y and X in G'

$\diamond$

Figure 6.11: Assigning Input Parameter Filters

So far, we have implicitly assumed in our discussion that IdAttrs is a single attribute set, when in general it could be a disjunction of attribute sets. While it is usually the case that IdAttrs is a single attribute set (as in our working Trades example), there may be cases where it is not.

We now illustrate how the filters are implemented when IdAttrs is a disjunction of attribute sets. Let us suppose that a clean-subset filter Y_X^f is assigned to Y_X. Recall that in the single attribute set case where IdAttrs(Y_X) $= A$, Y_X^f is simply $CS[\mathcal{C}, A]$, and the filter identifies a subset S of the sequence that Y_X receives during resumption time, denoted $\mathcal{Y}'_X$, that can be discarded.

If IdAttrs(Y_X) is a disjunction of attribute sets $A_1 \vee \ldots \vee A_n$, each attribute set A_i identifies a subset S_i of the $\mathcal{Y}'_X$ tuples that can be safely discarded considering one or more paths from Y_X to the warehouse. The problem is that there may be tuples in S_i that cannot be safely filtered when other paths are considered. The solution is to discard only the tuples that can be safely filtered along all paths. That is, only the tuples in $S_1 \cap \ldots \cap S_n$ are filtered.

To implement this solution, each A_i in IdAttrs(Y_X) results in a "sub-filter" denoted Y_X^i $= CS[\mathcal{C}, A_i]$. The overall CS filter Y_X^f then works as follows. For each tuple $x \in \mathcal{Y}'_X$, Y_X^f passes x to each sub-filter Y_X^i. If all sub-filters discard x, then x is discarded. Otherwise, x passes through.

The implementation of other filter types are altered in a similar fashion. For instance, if Y_X is assigned a clean-prefix filter, then the sub-filter Y_X^i is $CP[\text{Last}(\mathcal{C}), A_i]$.

6.5.3 Re-extraction Procedures

We now define the re-extraction procedures formally. From these definitions, it is clear that the re-extraction procedures are very similar to the filters. In particular, the re-extraction procedures GetSuffix and GetSubset perform the same processing as the CP and CS filters, respectively. Furthermore, we introduce the re-extraction procedures GetDirtySuffix and GetDirtySubset that correspond to the DP and DS filters.

Definition 6.5.1 (Re-extraction procedures for resumption) GetAllInorder() $= \mathcal{E}_O$, where $\mathcal{E}_O$ was the the output of E during normal operation.
GetAll() $= \mathcal{T}$: $\mathcal{T}$ and $\mathcal{E}_O$ have the same set of tuples.
GetSuffix(s,A) $= \mathcal{T}$: $CP[s,A] = \mathcal{T}$.
GetDirtySuffix(s,A) $= \mathcal{T}$: $DP[s,\ A] = \mathcal{T}$.
GetSubset($\mathcal{S}$,A) $= \mathcal{T}$: $CS[\mathcal{S},\ A] = \mathcal{T}$.
GetDirtySubset($\mathcal{S}$,A) $= \mathcal{T}$: $DS[\mathcal{S},\ A] = \mathcal{T}$.
□

The definition assumes that A is a single set of attributes, and not a disjunction of attributes. However, the extraction procedures can be easily altered just like the filters (in Section 6.5.2) to accommodate a disjunction of attributes. We do not show the extension here.

Since the re-extraction procedures and filters perform similar processing, it is not surprising that the procedure *AssignReextraction* is similar to *AssignFilter*. To illustrate, consider an extractor E and a component Y that receives E's output. If Prefix- feasible(Y_E), Subset-feasible(Y_E) and IdAttrs(Y_E) $\neq$ [ ], then we can assign a clean-prefix filter CP to Y_E. However, this filter can be "pushed" to E if E supports GetSuffix. Similarly, the other parts of *AssignReextraction* tries to push the remaining filter types from Y_E to E. In Section 6.7, we show experimentally the benefits of pushing the filtering to the extractors. If no filter can be pushed to an extractor E, either GetAllInorder or GetAll is assigned to it.

The full listing of the *AssignReextraction* algorithm is given in Figure 6.12. Lines 1–2 try to push a clean-prefix filter to the extractor using GetSuffix. Lines 3–4 try to push a dirty-prefix filter to the extractor using GetDSuffix. Lines 5–6 try to push a clean-subset filter to the extractor using GetSubset. Lines 7–8 try to push a dirty-subset filter to the extractor using GetSubset. If no filter can be pushed, *AssignReextraction* tries to assign GetAllInorder. Otherwise, GetAll, which is assumed to be supported, is assigned.

Algorithm 6.5.2 *AssignReextraction*
Input: Component DAG G'; extractor E
Side effect: Extractor E in G' is assigned a re-extraction procedure

1. If Prefix-feasible(Y_E) and Subset-feasible(Y_E) and IdAttrs(Y_E) $\neq$ [ ] and E supports GetSuffix
   2. Assign GetSuffix(Last($\mathcal{C}$),IdAttrs(Y_E)) to E in G'
3. Else If Prefix-feasible(Y_E) and IdAttrs(Y_E) $\neq$ [ ] and E supports GetDirtySuffix
   4. Assign GetDirtySuffix(Last($\mathcal{C}$), IdAttrs(Y_E)) to E in G'
5. Else if Subset-feasible(Y_E) and IdAttrs(Y_E) $\neq$ [ ] and E supports GetSubset
   6. Assign GetSubset($\mathcal{C}$, IdAttrs(Y_E)) to E in G'
7. Else if Prefix-feasible(Y_E) and IdAttrs(Y_E) $\neq$ [ ] and E supports GetDirtySubset
   8. Assign GetDirtySubset($\mathcal{C}$, IdAttrs(Y_E)) to E in G'
9. Else if E supports GetAllInorder
   10. Assign GetAllInorder() to E in G'
11. Else Assign GetAll() to E in G'

$\diamond$

Figure 6.12: Assigning Re-extraction Procedures

6.5.4 The *Design* and *Resume* Algorithms

Algorithm *Design* of *DR* (Algorithm 6.5.3, Figure 6.13) starts by computing the IdAttrs and the Prefix-feasible and Subset-feasible transitive properties of each input parameter Y_X in the given component DAG G. The input parameters are processed in reverse topological order because all of the above properties of Y_X depend on the properties of subsequent input parameters (*e.g.*, Z_Y).

Then *Design* calls *AssignReextraction* to assign re-extraction procedures to each extractor in G'. Next, *Design* computes the Same-seq property and calls *AssignFilter* (Figure 6.11) to assign filters to each input parameter in G'. Since the Same-seq property of Y_X depends on the Same-seq properties of previous input parameters, the input parameters are processed in topological order. Note that Same-seq(Y_X) is set to false if a filter is assigned to Y_X, because the filter ensures that Y_X does not receive the same input sequence as it did during normal operation. Redundant filters are removed and then G' is saved persistently.

In case of failure, *Resume* of *DR* (Algorithm 6.5.4, Figure 6.13) simply instantiates the re-extraction procedures and filters in G' with the actual value of the warehouse tuple sequence $\mathcal{C}$. The warehouse load is then resumed by invoking the re-extraction procedures. Note that *Resume* can be invoked multiple times on the same G', while *Design* only needs to be called once, at design time, regardless of the number of failures.

We now discuss how redundant filters are detected by *Design*. We say a filter Y_X^f is

Algorithm 6.5.3 *Design*
Input: Component DAG G
Output: Component DAG G'
1. $G' \leftarrow G$ // copy G
2. Compute IdAttrs(Y_X), Subset-feasible(Y_X), Prefix-feasible(Y_X) for each input parameter Y_X in reverse topological order.
3. For each extractor E
   4. *AssignReextraction*(G',E)
5. For each input parameter Y_X in topological order
   6. Compute Same-seq(Y_X)
   7. *AssignFilter*(G',Y_X)
   8. If Y_X is assigned a filter, set Same-seq(Y_X) to false.
9. *RemoveRedundantFilters*(G, G')
10. Save G' persistently and return G'

$\diamond$

Algorithm 6.5.4 *Resume*
Input: Component DAG G'
Side Effect: Resumes failed warehouse load using G
Let $\mathcal{C}$ be the tuples in the warehouse
1. Instantiate each re-extraction procedure in G', and each filter in G' with actual value of $\mathcal{C}$
2. For each extractor E in G'
   3. Invoke re-extraction procedure assigned to E

$\diamond$

Figure 6.13: *DR* Algorithm

redundant if Y_X^f is guaranteed not to discard any tuples. Given a path P in G, with V_U preceding Y_X in P, Y_X^f in G' is redundant if there is a filter V_U^f in G' and the following two conditions hold:

1. V_U^f is of filter type f (*e.g.*, CP) and Y_X^f is of filter type g (*e.g.*, CS) and $f \succ g$ or $f = g$.

2. IdAttrs(V_U) $\subseteq$ IdAttrs(Y_X).

Once Y_X^f is detected as redundant, it is removed from G'. A brute force way to detect redundant filters is to consider each path in G' and check the above conditions.

We now discuss how redundant filters can be efficiently detected. Recall that any filter

Y_X^f assigned to Y_X can be one of four filter types: CP, DP, CS, and DS. Since the re-extraction procedures perform the same processing as the input parameter filters, we say that GetSuffix is of filter type CP, GetDirtySuffix is of filter type DP, GetSubset is of filter type CS, and GetDirtySubset is of filter type DS. The GetAllInorder and GetAll re-extraction procedures do not filter any tuples and have no filter types.

The key in removing a redundant filter for Y_X is deducing the filters that are already "in effect" for Y_X due to previous filters or re-extraction procedures. For instance, if a CP filter is assigned to X_V, then Y_X will only receive a suffix of its normal operation input. Thus, even if there is no filter assigned to Y_X, a CP filter is "in effect". If a CP filter is already "in effect", any Y_X^f filter would be redundant since CP filters discard the most tuples. Similarly, if X is an extractor that is assigned GetSuffix, a CP filter is already in effect for Y_X, and any Y_X^f filter would be redundant.

To capture the filters that are in effect, we introduce a field $Y_X.inEffect$ that contains a set of filter types for each input parameter Y_X. (Actually, $inEffect$ also records the attribute sets used by re-extraction procedures and input parameter filters so that redundant filters can be compared appropriately.) Initially, the $inEffect$ field of each input parameter is set to { } by *RemoveRedundantFilters* in Lines 1–2 (Figure 6.14). The algorithm then computes the filter types in effect due to the re-extraction procedures in Lines 3–5. The algorithm then processes the input parameters in topological order to ensure that the filter types "in effect" are computed correctly. In Lines 9–10, it checks if the filter Y_X^f is redundant because of previous filters or re-extraction procedures. The effect of previous filters or re-extraction procedures is conveniently recorded in $Y_X.inEffect$. If Y_X^f is redundant, it is removed from G'. Otherwise, Y_X^f stays and the type of filtering it provides is recorded in $Y_X.inEffect$ (Lines 11–14).

We now analyze the complexity of *DR*. Let n be the number of nodes in G. Steps 2–8 of *Design* produce a topological ordering of the nodes in G and then traverse it. They take $O(n^2)$ time. Detecting redundant filters in step 9 also takes $O(n^3)$ time (see Appendix B). *Resume* instantiates at most $O(n^2)$ filters. Usually many fewer than $O(n^2)$ filters are created. Furthermore, we show in our experiments (Section 6.7) that even adding a single filter can dramatically improve performance. Subset filters can be instantiated in $O(|\mathcal{C}|)$ time, where $|\mathcal{C}|$ is the number of warehouse tuples. Prefix filters are instantiated in $O(1)$ time (with appropriate indices on warehouse tables). Therefore, *DR* runs in $O(n^2 \cdot |\mathcal{C}| + n^3)$ time.

Algorithm 6.5.5 *RemoveRedundantFilters*
Input: Component DAGs G, G'
Side effect: G' with any redundant filters removed
1. For each Y_X in G'
    2. $Y_X.inEffect \leftarrow \{\ \}$
3. For each extractor E in G'
    4. For each Y_E in G'
        Let g be the filter type of E in G'
        Let E use the attribute set A in its re-extraction procedure
        5. $Y_E.inEffect \leftarrow Y_E.inEffect \cup \{\langle g, A\rangle\}$
6. For each Y_X in G in topological order
    7. If Y_X^f is in G' Then
        8. Let g be the filter type of Y_X^f. Let Y_X^f use the attribute set A
        9. If there is a filter type $\langle f, A'\rangle \in Y_X.inEffect$ and $f \succ g$ and $A' \subseteq A$ Then
            10. Remove Y_X^f from G' and connect X to Y in G' // redundant filter removed
        11. Else
            12. $Y_X.inEffect \leftarrow Y_X.inEffect \cup \{\langle g, A\rangle\}$
    13. For each Z_Y in G
        14. $Z_Y.inEffect \leftarrow Z_Y.inEffect \cup Y_X.inEffect$

$\diamond$

Figure 6.14: Removing Redundant Filters

6.5.5 Correctness of *DR*

A correct load resumption algorithm produces the same set of tuples in the warehouse as the original load would have, had there been no failures. By this definition, *DR* is correct. *DR* only filters tuples that are not needed to produce subsequent warehouse tuples. Furthermore, no warehouse tuple in $\mathcal{C}$ is reproduced.

DR only filters unneeded tuples because it relies on the properties defined in Section 6.4. For instance, if Subset-feasible(Y_X) holds, then *DR* can safely filter some Y_X tuples, knowing that those tuples only contribute to a warehouse tuple already in $\mathcal{C}$. *DR* ensures that none of the $\mathcal{C}$ tuples is reproduced by guaranteeing that a CP or CS filter is assigned. Since a clean filter removes all of the contributors to $\mathcal{C}$ tuples, none of the $\mathcal{C}$ tuples are reproduced. Since the input parameter of W is guaranteed to be Subset-feasible and have non-empty IdAttrs, *DR* can always assign it a CS filter (if no other filter assignment is possible).

6.6 *DR* and Logging

The *DR* algorithm does not log any transform output. However, it may be beneficial in many cases to log the output of some transforms. In our example, logging the one-tuple output of transform TV is most likely beneficial because we can avoid reprocessing the input tuples to produce the total volume. In this section, we develop *DR-Log* that builds on *DR* but may log some transform outputs. *DR-Log* uses the contents of the logs to possibly assign additional filters and assign more efficient re-extraction procedures (than what *DR* would have assigned). *DR-Log* needs to solve the following problems at design time:

1. How to use the logs in the filters and re-extraction procedures; and

2. Where to put the logs.

We present *DR-Log* by answering the above questions in Sections 6.6.1 and 6.6.2. The *DR-Log* we present here only handles component DAGs that do not have any components that replicate their outputs. These component DAGs have the special property that there is a unique path from any component to the warehouse inserter. A *DR-Log* that handles general component DAGs is important future work.

In addition to the extractor, transform and warehouse inserter components, a new component called a *log inserter* is used by *DR-Log*. A log inserter L functions similarly to a warehouse inserter. That is, a log inserter L takes as input the output of either an extractor or a transform, and inserts the input tuples in batches into a log. In *DR-Log*, the log inserters log all of the input tuples, and all of the attributes of each input tuple. That is, if log inserter L is placed after the component X, then L will log all of the tuples in $\mathcal{X}_O$ in order, and all of the attributes $\text{Attrs}(X_O)$ of each tuple. Investigating the use of log inserters that log only summary information about their input is important future work.

A log inserter can be placed anywhere. However, a log inserter that is placed after the penultimate component (*i.e.*, the component that feeds the warehouse inserter) will log tuples that are also stored in the warehouse. Thus, we do not consider placing a log after the penultimate component. As a result, a log inserter is always placed after a component X whose output is fed to some transform Y, and never before the warehouse inserter. So that the processing of the component DAG is not disturbed, the log inserter placed after X needs to forward the tuples it is logging to Y.

6.6.1 Using The Logs

To illustrate how *DR-Log* uses the various log contents, let us suppose that a log inserter L is placed in between transforms X and Y (*i.e.*, $L_X = Y_X = X_O$). At resumption time, *DR-Log* starts by reading the contents $\mathcal{C}_L$ of L's log and sending all of the $\mathcal{C}_L$ tuples along input Y_X. Based on the tuples in $\mathcal{C}_L$, a filter placed by *DR-Log* can filter any tuples coming *into* X that contributed (only) to the $\mathcal{C}_L$ tuples. This is because the tuples in $\mathcal{C}_L$ do not need to be reproduced. This filtering is analogous to how *DR* filtered input tuples based on the warehouse tuples, except that now the tuples in the log are used as the basis of the filtering.

Adding the log inserter L between X and Y is especially useful if for some input parameter X_V of X, IdAttrs(X_V) has been computed as [ ] *and* X_V is either Prefix-feasible or Subset-feasible. In this case, *DR* would have been unable to place a filter at X_V based on warehouse tuples, or at any edge before X_V in the load workflow! However, in this case, *DR-Log* is able to add filters to X_V based on the contents $\mathcal{C}_L$ of L's log. These filters remove tuples based on what is in $\mathcal{C}_L$, and not on what is in the warehouse.

We now present the *DR-Log* algorithm in detail. We first discuss how the definitions of Prefix-feasible and Subset-feasible are modified. Next, we present the definitions of CandLogAttrs and IdAttrs which are analogous to the definitions of CandAttrs (Definition 6.4.5) and IdAttrs (Definition 6.4.6), respectively. We then present the modifications to the *AssignFilter* (Figure 6.11) and the *AssignReextraction* (Figure 6.12) procedures. Finally, we present the overall *DR-Log* algorithm.

Modifications to Prefix-feasible and Subset-feasible

Recall that Subset-feasible(Y_X) states that the Y_X tuples contribute to at most one warehouse tuple. If the contributors along Y_X can be identified (*i.e.*, IdAttrs(Y_X) $\neq$ [ ]), then because Y_X is Subset-feasible, some of the Y_X tuples can be filtered based on the tuples in the warehouse. The definition of Subset-feasible needs to be altered because it is possible to filter Y_X tuples if they contribute to at most one warehouse tuple, *or* to at most one tuple in the nearest log. Note that because we focus on component DAGs that do not have transforms that replicate their outputs, there is a unique path from Y_X to the warehouse. Because there is a unique path, there can only be one log that is nearest to Y_X.

Subset-feasible is modified as shown in Definition 6.6.1. The definition of Prefix-feasible is modified in a similar fashion (Definition 6.6.2).

Definition 6.6.1 (**Subset-feasible(Y_X)**) Given transform Y with input parameter Y_X, Subset-feasible(Y_X) = true if Y is the warehouse inserter. Otherwise, Subset-feasible(Y_X) = true if there is a log inserter L that records X_O (*i.e.*, Y_X). Otherwise, Subset-feasible(Y_X) = true if Y_X is map-to-one and $\forall Z_Y$: Subset-feasible(Z_Y). Otherwise, Subset-feasible(Y_X) = false. □

Definition 6.6.2 (**Prefix-feasible(Y_X)**) Given transform Y with input parameter Y_X, Prefix-feasible(Y_X) = true if Y is the warehouse inserter. Otherwise, Prefix-feasible(Y_X) = true if there is a log inserter L that records X_O (*i.e.*, Y_X). Otherwise, Prefix-feasible(Y_X) = true if Y_X is suffix-safe and $\forall Z_Y$: Prefix-feasible(Z_Y). Otherwise, Prefix-feasible(Y_X) = false. □

CandLogAttrs and IdLogAttrs

Recall that IdAttrs(Y_X) gives the attributes of Y_X that can be used to find the Y_X contributors to warehouse tuples. We now define IdLogAttrs(Y_X) which gives the attributes of Y_X that can be used to find the Y_X contributors to the log tuples of the nearest log inserter. Again, it is important to note that either there are no log inserters in the path from Y_X to the warehouse, or there is a unique log inserter that is nearest to Y_X. The definition of CandLogAttrs (Definition 6.6.3) is very similar to CandAttrs (Definition 6.4.5), except that the logs are taken into account. Similarly, the definition of IdLogAttrs (Definition 6.6.4) is very similar to IdAttrsPerPath (Definition 6.4.6).

Definition 6.6.3 (**CandLogAttrs(Y_X)**) There are four possibilities for CandLogAttrs(Y_X).

1. If Y is the warehouse inserter, then CandLogAttrs(Y_X) = [ ].
2. Else if there is a log inserter L that takes as input X's output ($X_O = L_X = Y_X$), then CandLogAttrs(Y_X) = Attrs(Y_X).
3. Else if Y_X has hidden contributors, then CandLogAttrs(Y_X) = [ ].
4. Else, CandLogAttrs(Y_X) = CandLogAttrs(Z_Y) $\cap$Attrs(Y_X).

□

Definition 6.6.4 (IdLogAttrs(Y_X)) There are five possibilities for IdLogAttrs(Y_X).

1. If Y is the warehouse inserter, then IdLogAttrs(Y_X) = [ ].
2. Else if there is a log inserter L that takes as input X's output ($X_O = L_X = Y_X$), then IdLogAttrs(Y_X) = KeyAttrs(Y_X).
3. Otherwise, if there is a log inserter in the path from Y_X to the warehouse, let P be the unique path from Y_X to the nearest log inserter L.
   If (KeyAttrs(Y_X) $\subseteq$ CandLogAttrs(Y_X) and $\forall Z_V \in P$: Z_V has no spurious output tuples),
   then (IdLogAttrs(Y_X) = KeyAttrs(Y_X)).
4. Otherwise, if there is a log inserter in the path from Y_X to the warehouse, let P be the unique path from Y_X to the nearest log inserter L. Let $Z_V \in P$ but $Z_V \neq Y_X$.
   If (IdLogAttrs(Z_V) $\neq$ [ ] and IdLogAttrs(Z_V) $\subseteq$ CandLogAttrs(Y_X)),
   then (IdLogAttrs(Y_X) = IdLogAttrs(Z_V)).
5. Otherwise IdLogAttrs(Y_X) = [ ].

□

AssignFilter **and** *AssignReextraction*

Recall that *DR* assigned a filter to Y_X if the contributors to the warehouse tuples $\mathcal{C}$ can be identified using IdAttrs(Y_X) (*i.e.*, IdAttrs(Y_X) $\neq$ [ ]). Furthermore, the contributors are filtered only if *DR* knew that they were safe to filter out (*i.e.*, Subset-feasible(Y_X) or Prefix-feasible(Y_X)).

In the case of *DR-Log*, not only can it filter Y_X tuples based on $\mathcal{C}$, but it may also filter tuples based on the log contents $\mathcal{C}_L$ of the log inserter L nearest to Y_X. That is, *DR-Log* assigns a filter to Y_X if the contributors to the log tuples $\mathcal{C}_L$ can be identified using IdLogAttrs(Y_X) (*i.e.*, IdLogAttrs(Y_X) $\neq$ [ ]). Furthermore, the contributors are filtered only if *DR-Log* knows that they are safe to filter out (*i.e.*, Subset-feasible(Y_X) or Prefix-feasible(Y_X)).

A filter can conceivably discard tuples based on a combination of log tuples and warehouse tuples. Here, we present a simple filter assignment algorithm that chooses to use log tuples over warehouse tuples whenever possible. This is because the log is likely to contain

Algorithm 6.6.1 *AssignFilter*
Input: Component DAG G'; input parameter Y_X
Output: Input parameter Y_X in G' is assigned a filter whenever possible
Let $\mathcal{C}_L$ be the contents of the log inserter nearest to Y_X
1. If Prefix-feasible(Y_X) and Subset-feasible(Y_X) and Same-seq(Y_X) and IdLogAttrs(Y_X) $\neq$ [ ]
   2. Insert $Y_X^f = CP[\text{Last}(\mathcal{C}_L), \text{IdLogAttrs}(Y_X)]$ between Y and X in G'
3. Else If Prefix-feasible(Y_X) and Same-seq(Y_X) and IdLogAttrs(Y_X) $\neq$ [ ]
   4. Insert $Y_X^f = DP[\text{Last}(\mathcal{C}_L), \text{IdLogAttrs}(Y_X)]$ between Y and X in G'
5. Else if Subset-feasible(Y_X) and IdLogAttrs(Y_X) $\neq$ [ ]
   6. Insert $Y_X^f = CS[\mathcal{C}_L, \text{IdLogAttrs}(Y_X)]$ between Y and X in G'
7. Else if Prefix-feasible(Y_X) and IdLogAttrs(Y_X) $\neq$ [ ]
   8. Insert $Y_X^f = DS[\mathcal{C}, \text{IdLogAttrs}(Y_X)]$ between Y and X in G'

9–16. Lines 1–8 in Figure 6.15

$\diamond$

Figure 6.15: Assigning Input Parameter Filters

more tuples and attributes since the log is "closer" to the edge Y_X than the warehouse. The modified *AssignFilter* and *AssignReextraction* procedures are shown in Figures 6.15 and 6.16, respectively.

DR-Log Details

The algorithm *DR-Log* is shown in Figure 6.17. *DR-Log* returns two component DAGs. The first one, denoted G_{log}, is the same as G except that it may have log inserters assigned to record the output of some of the transforms. G_{log} is used during normal operation. *DR-Log* also returns G', which is the same as G except it has filters and reextraction procedures assigned. G' is used during recovery.

DR-Log uses a procedure *AssignLogs* discussed in the next section to assign the logs. Once the logs are assigned, *DR-Log* proceeds very much like *DR* proceeds. That is, it first computes IdAttrs, Subset-feasible, Prefix-feasible and IdLogAttrs it the backward pass (Line 2). Extractors are assigned using the modified *AssignReextraction* procedure (Lines 3–4). The forward pass (Lines 5–10) of *DR-Log* is the same as *DR* except that the modified *AssignFilter* procedure is used.

At resumption time, *DR-Log* uses the contents of the logs and the warehouse to instantiate the filters and the extraction procedures. The failed warehouse load is re-strated by firing the extraction procedures.

Algorithm 6.6.2 *AssignReextraction*
Input: Component DAG G'; extractor E
Side effect: Extractor E in G' is assigned a re-extraction procedure
Let $\mathcal{C}_L$ be the contents of the log inserter nearest to Y_X
1. If Prefix-feasible(Y_E) and Subset-feasible(Y_E) and IdLogAttrs(Y_E) $\neq$ [ ] and E supports GetSuffix
   2. Assign GetSuffix(Last($\mathcal{C}_L$),IdLogAttrs(Y_E)) to E in G'
3. Else If Prefix-feasible(Y_E) and IdLogAttrs(Y_E) $\neq$ [ ] and E supports GetDirtySuffix
   4. Assign GetDirtySuffix(Last($\mathcal{C}_L$), IdLogAttrs(Y_E)) to E in G'
5. Else if Subset-feasible(Y_E) and IdLogAttrs(Y_E) $\neq$ [ ] and E supports GetSubset
   6. Assign GetSubset($\mathcal{C}_L$, IdLogAttrs(Y_E)) to E in G'
7. Else if Prefix-feasible(Y_E) and IdLogAttrs(Y_E) $\neq$ [ ] and E supports GetDirtySubset
   8. Assign GetDirtySubset($\mathcal{C}_L$, IdLogAttrs(Y_E)) to E in G'

9–19. Lines 1–11 of Figure 6.12

$\diamond$

Figure 6.16: Assigning Re-extraction Procedures

6.6.2 Log Placement

While using logs benefits load resumption, it incurs overhead during normal operation. In deciding where to put the logs, the following factors need to be taken into account: the amount of space available for logging tuples, the amount of normal operation overhead allowed, and the time to complete a resumed load using the logs. Where to add logs to a component DAG depends on which of these factors is most important. We do not give a detailed cost model here. Instead, we present a cost-model independent log placement algorithm that puts the least number of logs while achieving the most filtering possible. However, even though the number of logs is minimized, the normal operation overhead as well as the space for logs may still be excessive. Thus, a cost-based log placement algorithm is an important future work. The algorithm we present here, called *AssignLogs*, is run by *DR-Log* (Line 1a, Figure 6.17) during design time. We also present some heuristics to guide log placement.

The log placement algorithm is shown in Figure 6.18. It first computes IdAttrs(Y_X), Prefix-feasible(Y_X) and Subset-feasible(Y_X) for each edge Y_X in the component DAG. The IdLogAttrs of each edge is also computed, but they are all [ ] since there are no logs initially. The algorithm puts a log whenever there is an opportunity for filtering (*i.e.*, either Subset-feasible(Y_X) or Prefix-feasible(Y_X)) and neither the warehouse contents nor the contents of logs already present can be used for identifying contributors (*i.e.*, both IdAttrs(Y_X) and

Algorithm 6.6.3 *Design* and *Log*
Input: Component DAG G
Output: Component DAG G_{log}, G'
1a. $G_{log} \leftarrow AssignLogs(G)$
1b. $G' \leftarrow G$ // copy G
2. Compute IdAttrs(Y_X), IdLogAttrs(Y_X), Subset-feasible(Y_X), Prefix-feasible(Y_X) for each input parameter Y_X in reverse topological order based on G_{log}.
3. For each extractor E
   4. *AssignReextraction*(G',E)
5. For each input parameter Y_X in topological order
   6. Compute Same-seq(Y_X) based on G_{log}
   7. *AssignFilter*(G',Y_X)
   8. If Y_X is assigned a filter, set Same-seq(Y_X) to false on G_{log}.
9. *RemoveRedundantFilters*(G, G')
10. Save G_{log} and G' persistently and return G' and G_{log}

◇

Algorithm 6.6.4 *Resume*
Input: Component DAGs G_{log}, G'
Side Effect: Resumes failed warehouse load using G_{log}
Let $\mathcal{C}$ be the tuples in the warehouse
1. Instantiate each re-extraction procedure in G', and each filter in G' with actual value of the log tuples closest to that filter or procedure. If there is no log that can be used, instantiate based on $\mathcal{C}$
2. For each extractor E in G'
   3. Invoke re-extraction procedure assigned to E

◇

Figure 6.17: *DR-Log* Algorithm

IdLogAttrs(Y_X) are [ ]). This way, the amount of filtering is maximized. By processing the transforms in reverse topological order, the logs are placed as close to the warehouse as possible. This way, the contents of a log can potentially be used for filtering in as many edges of the component DAG as possible, minimizing the logs added to the component DAG.

Since this simple log placement algorithm is oblivious of the complexity of the transforms and the extractors, the following heuristics can be used to decide where to place additional logs.

- If a transform X performs a complex computation that takes a long time to complete,

Algorithm 6.6.5 *AssignLogs*
Input: Component DAG G
Output: Component DAG G_{log}

1. Compute IdAttrs(Y_X), IdLogAttrs(Y_X), Subset-feasible(Y_X), Prefix-feasible(Y_X) for each input parameter Y_X in the component DAG in reverse topological order. Also compute IdLogAttrs(E_O) and IdAttrs(E_O) for each extractor E in the component DAG.
2. For each transform Y in reverse topological order
   3. For each Y_X such that ((Subset-feasible(Y_X) or Prefix-feasible(Y_X))
      and (IdAttrs(Y_X) = [ ] and IdLogAttrs(Y_X) = [ ])
      and (Y_X has no hidden contributors))
      4. Create log L to log X_O
      5. Recompute IdLogAttrs for all edges of the component DAG

$\diamond$

Figure 6.18: Log Placement Algorithm

add a log $L_X = X_O$. If the log contains the results of X, $\mathcal{X}_O$, at resumption time, then the load can resume based on the contents of L alone. This implies that X does not need to produce any output, and also does not require any input from the the transforms or extractors that feed into it. Even if the log contains only part of $\mathcal{X}_O$, it may still be possible to filter inputs to X based on $\mathcal{L}_X$ and only compute the remaining portion of $\mathcal{X}_O$.

- If it takes a lot of effort to extract E_O from a source and data extractor E, add a log inserter $L_E = E_O$. Remote sources, legacy sources, and transient (*e.g.*, newsfeed) sources are good candidates to have their output logged. If the load fails but all of $\mathcal{E}_O$ is in the log at resumption time, then no re-extraction procedure is necessary for E.

6.7 Experiments

In this section, we present our experiments that compares *DR* to other recovery algorithms in terms of normal operation overhead and recovery cost. Although we did not measure the overhead and recovery cost of *DR-Log*, the performance of *DR-Log* should be similar to *DR-Save*, which we introduce shortly. We also show that the properties on which *DR* relies are quite common.

6.7.1 Study of Transform Properties

Sagent's Data Mart 3.0 is commercial software for constructing component DAGs for warehouse creation and maintenance. It provides 5 types of warehouse inserters, 3 types of extractors, and 19 transforms. The software also allows users to create their own transforms.

All three extractors support GetAllInorder and GetAll, but only the "SQL" Extractor supports GetSuffix, GetDirtySuffix, GetSubset and GetDirtySubset. Of the 19 transforms, 15 have one input parameter, and the other 4 have two input parameters, for a total of 23 input parameters. Figure 6.19 shows a summary of the properties that hold for the 19 transforms and the 23 input parameters. The 19 transforms include Sagent's implementations of conventional operations used in databases, such as selection, projection, union, aggregation, and join.

- 100% (19 out of 19) of the transforms are in-det-out.
- 95% (18 out of 19) of the transforms have no spurious output.
- 91% (21 out of 23) of the input parameters are map-to-one.
- 78% (18 out of 23) of the input parameters are suffix-safe.
- 17% (4 out of 23) of the input parameters are set-to-seq (*i.e.*, perform sorting).
- 100% (23 out of 23) of the input parameters have no hidden contributors.

Figure 6.19: Properties of Sagent Transforms and Input Parameters

Some of these properties, like suffix-safe, are actually declared by Sagent. Other properties were deduced easily from the Sagent manuals that specify the transforms. The statistics in Figure 6.19 imply that the transitive properties Subset-feasible (due to map-to-one), Prefix-feasible (due to suffix-safe) and Same-seq (due to in-det-out and set-to-seq) hold for many component DAG scenarios.

6.7.2 Resumption Time Comparison

We performed experiments using Sagent's Data Mart 3.0 to construct various component DAGs. The software ran on a Dell XPS D300 with a Pentium II 300 MHz processor and

64 MB of RAM.

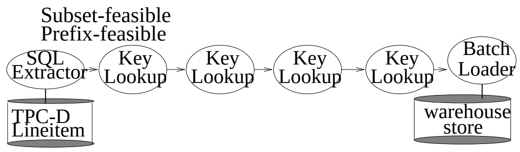

Figure 6.20: Fact Table Creation DAG

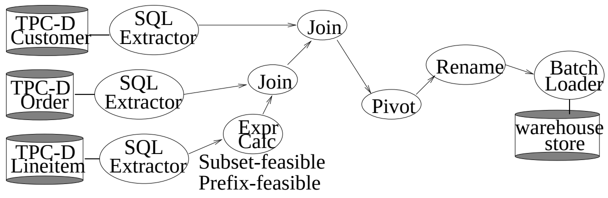

Figure 6.21: TPC-D View Creation DAG

We designed three types of component DAGs. One type of component DAG loads *dimension tables*, *e.g.*, the *Customer* and *Supplier* TPC-D tables [Com]. Dimension tables typically store data about entities like customers. Another type of component DAG loads *fact tables*, *e.g.*, the *Order* and *Lineitem* TPC-D tables. Fact tables typically store transactional data. The last type of component DAG loads materialized views that contain the answers to queries, *e.g.*, TPC-D queries. Since the results of the dimension and fact table scenarios were very similar, we only present results for the fact table and the TPC-D materialized view scenarios. The component DAGs for loading the TPC-D fact table *Lineitem*, and the materialized view for the TPC-D query *Q3* are shown in Figures 6.20 and 6.21 respectively. Query *Q3*, the "shipping priority query," joins 3 tables and performs a GROUP BY and a SUM of revenue estimates.

Experiment 1

In the first experiment, we compared the resumption times of *DR*, *Redo*, and the algorithm used by Informatica ([Inf]), denoted *Inf*, for the *Lineitem* DAG (Figure 6.20). Recall that *Inf* filters the input to the inserter "*BatchLoader*" based on the warehouse tuples. No other filters are employed by *Inf*. The three algorithms compared impose no overhead during normal operation but can handle complex workflows. That is, all the algorithms are in the lower right quadrant of Figure 6.2 (Section 6.1). Furthermore, we studied "variants" of *DR* by assuming different properties for the component DAG.

- **Variant** DR_{src}: DR_{src} pushes filtering to the re-extraction procedure at the source. In Figure 6.20, the transform properties show that $KeyLookup_{SQLExtractor}$ is both Prefix-feasible and Subset-feasible, and the extractor for *Lineitem* supports GetSuffix. Therefore, DR_{src} assigns GetSuffix to the *Lineitem* extractor.

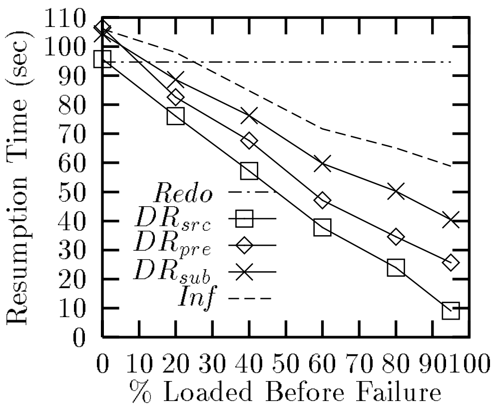

Figure 6.22: Resumption Time (*Lineitem*)

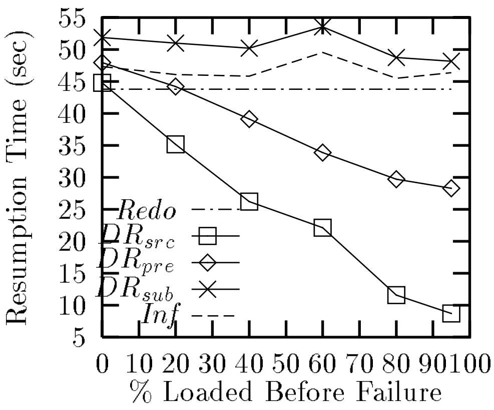

Figure 6.23: Resumption Time (*Q3*)

- **Variant DR_{pre}:** DR_{pre} assigns a prefix filter immediately after the *Lineitem* source. In Figure 6.20, DR_{pre} places a clean-prefix filter between the *Lineitem* extractor and *KeyLookup*. This component DAG will be constructed when the *Lineitem* extractor does not support GetSuffix.

- **Variant DR_{sub}:** DR_{sub} assigns a subset filter immediately after the *Lineitem* source. In Figure 6.20, DR_{sub} assigns a clean-subset filter to $KeyLookup_{SQLExtractor}$.

We compared *Redo*, *Inf* and the variants of *DR* under various failure scenarios. More specifically, we investigated scenarios where 0%, 20%, 40%, 60%, 80% and 95% of the warehouse table is loaded when the failure occurs. For example, since *Lineitem* has 60,000 tuples (*i.e.*, 0.01 TPC-D scaling), we investigated failures that occurred after loading 0 to 57,000 tuples. A low scaling factor was used so that the experiment can be repeated numerous times.

The results are shown in Figure 6.22, which plots the resumption time of *Inf*, *Redo*, DR_{src}, DR_{pre} and DR_{sub} as more tuples are loaded into the warehouse before the failure. As expected, DR_{src}, DR_{pre} and DR_{sub} all perform better than *Redo* once 20% (or more) of the *Lineitem* tuples reach the warehouse. For instance, when *Lineitem* is 95% loaded, DR_{src} resumes the load 10.4 times faster than *Redo*, DR_{pre} resumes the load 3.68 times faster, and DR_{sub} resumes the load 2.35 times faster. The variants of *DR* also resume the load significantly faster than *Inf*. For instance, when *Lineitem* is 95% loaded, DR_{src} resumes the load 6.46 times faster than *Inf*, DR_{pre} resumes the load 2.28 times faster, and

DR_{sub} resumes the load 1.45 times faster. On the other hand, when none of the *Lineitem* tuples reach the warehouse before the failure, *Inf*, DR_{sub}, and DR_{pre} perform worse than *Redo* because of the overhead of the filters they use. More specifically, when *Lineitem* is 0% loaded, *Redo* is 1.12 times faster than DR_{pre}, 1.10 times faster than DR_{sub}, and 1.12 times faster than *Inf*. The overhead of the filters can be minimized by improving their implementation. DR_{src} which pushes the filtering to the *Lineitem* source, is almost as fast as *Redo* when the warehouse table is 0% loaded. Preliminary experiments using 1.0 TPC-D scaling show very similar relative improvements by the *DR* variants over *Redo* when enough *Lineitem* tuples are loaded.

Among the three *DR* variants, DR_{src} performs the best since it filters the tuples the earliest. DR_{sub} performs worse than DR_{pre} because of the overhead of the anti-semijoin operation employed by DR_{sub}'s subset filters. Furthermore, the next experiment will show that DR_{pre} filters more tuples than DR_{sub}.

Experiment 2

The second experiment is similar to the first but considers the *Q3* DAG (Figure 6.21). The results are shown in Figure 6.23. As in the first experiment, DR_{pre} and DR_{src} perform better than *Redo* once 20% (or more) of the warehouse table tuples is loaded. For instance, when the warehouse table is 95% loaded, DR_{src} is 5.03 times faster than *Redo*, and DR_{pre} is 1.55 times faster than *Redo*. However, DR_{sub} and *Inf* perform worse than *Redo* regardless of how many tuples are loaded. For instance, *Redo* is 1.22 times faster than DR_{sub} when the warehouse table is 60% loaded. The reason why DR_{sub} and *Inf* do not perform well is that query *Q3* is very selective, and many of the source tuples extracted do not contribute to any warehouse tuple. Since subset filters can only remove tuples that contribute to a warehouse tuple, the filters used by DR_{sub} do not remove enough tuples to compensate for the cost of the filter. Similarly, the filter used by *Inf* removes tuples based only on the warehouse tuples. Just like DR_{sub}, *Inf* does not filter many tuples.

Experiment 3

In the third experiment, we examined the normal operation overhead of a recovery algorithm that is based on savepoints. Such an algorithm is a representative of the algorithms in the upper right quadrant of Figure 6.2 (Section 6.1). We again considered the *Lineitem* and *Q3* component DAGs. For the former component DAG, we introduced 1 to 3 savepoints. For instance, the first savepoint records the result of the first "*KeyLookup*" transform. The results are shown in Figure 6.24. Without savepoints, the *Lineitem* table is loaded in 94.7 seconds. As the table shows, one savepoint makes the normal operation load 1.76 times slower, two savepoints make the normal load 2.6 times slower, and three savepoints make the normal load 3.3 times slower. On the other hand, the algorithms compared in the first two experiments (*e.g.*, *DR*) have no normal operation overhead, and do not increase the load time.

For the *Q3* DAG, we also introduced 1 to 3 savepoints. The first savepoint records the result of the first "*Join*" transform, the second records the result of the second "*Join*" transform, and the third records the result of the "*Pivot*" and "*Rename*" transform. The normal operation overhead of the savepoints is tolerable for this component DAG. Even with three savepoints, the normal operation load is only about 1.08 times slower. The reason why the savepoints do not incur much overhead is that the "*Join*" transforms are very selective. Hence, only few tuples are recorded in the savepoints. More specifically, the first savepoint records 1344 tuples, the second records 285 tuples, and the third records 103 tuples.

# Savepoints	Load Time (s)	% Increase Load Time
0	94.7	0%
1	166.4	75.7%
2	245.9	159.7%
3	314.0	231.6%

Figure 6.24: Savepoint Overhead (*Lineitem*)

# Savepoints	Load Time (s)	% Increase Load Time
0	43.8	0%
1	46.1	5.3%
2	46.9	7.1%
3	47.2	7.8%

Figure 6.25: Savepoint Overhead (*Q3*)

Experiment 4

In the fourth experiment, we compared the resumption time of *DR* against an algorithm based on savepoints, denoted *Save*. We compared the two algorithms under various failure scenarios. For instance, for *DR* we would load the warehouse using the *Lineitem* DAG, and stop the load after t_{fail} seconds. To simulate various failure scenarios, we would vary t_{fail}. We then resumed the load using *DR* and recorded the resumption time. For *Save*, we would load the warehouse using the same *Lineitem* DAG, but with savepoints. We also stop the load after t_{fail} seconds. We then resumed the load using any completed savepoints. We again considered the *Lineitem* and *Q3* DAGs. In the case of *Save*, we used two savepoints for each component DAG.

The result for the *Lineitem* DAG is shown in Figure 6.26 which plots the resumption time of *DR* and *Save* as t_{fail} is increased. The graph shows that *Save*'s resumption time improves in discrete steps. For instance, when $t_{fail} < 79$ seconds, the first savepoint has not completed and cannot be used. Once $t_{fail} > 79$ seconds, the first savepoint can be used to make resumption more efficient. For the *Lineitem* DAG, *DR* is more efficient than *Save* in resuming the load. This is because the warehouse table is populated early in the load, and *DR* can use the warehouse table tuples to make resumption efficient.

The result for the *Q3* DAG is shown in Figure 6.27. Again, *Save*'s resumption time improves in discrete steps based on the completion of the savepoints. For this DAG, *DR*'s resumption time does not improve until t_{fail} is near 43 seconds (when the load completes). This is because the second "*Join*" transform takes in excess of 30 seconds to produce its first output tuples. As a result, the warehouse table is not populated until the load time is near 43 seconds. For this DAG, *Save* is slightly more efficient than *DR* in resuming the load for many values of t_{fail}. Unfortunately, both *Save* and *DR* do not perform well.

To improve the resumption performance, a hybrid algorithm that combines the features of *Save* and *DR* can be employed. The two savepoints employed by *Save* essentially partition the *Q3* DAG into three "sub-DAGs." However, *Save* does not make use of incomplete savepoints to improve resumption. On the other hand, *DR* can be used to treat an incomplete savepoint and the "sub-DAG" that produced it as if it was a warehouse table being loaded by a component DAG. The performance of the hybrid algorithm, denoted *DR-Save*, is plotted in Figure 6.27. For most values of t_{fail}, *DR-Save* is better than either *Save* or *DR*.

Experiment 5

In the fifth experiment, we examined the normal operation overhead of a recovery algorithm that is based on batching. Such an algorithm is a representative of the algorithms in the lower left quadrant of Figure 6.2 (Section 6.1). We again considered the *Lineitem* and *Q3* component DAGs. For the former component DAG, we loaded *Lineitem* in three input batches. The results are shown in Table 6.3. The table shows that batching results in a significant overhead especially when 4 or more batches are used.

For the *Q3* DAG, we also loaded the target table using three input batches. The results are shown in Table 6.4. Again, the table shows that batching results in a significant overhead

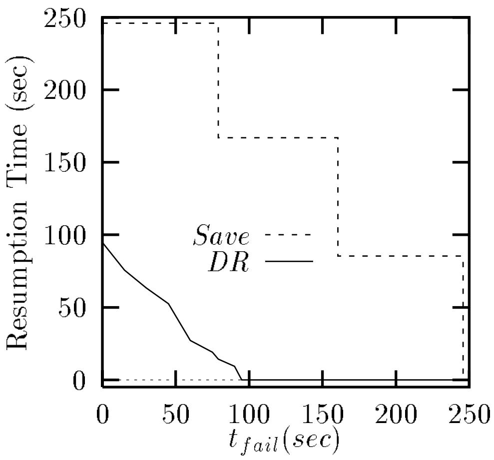

Figure 6.26: *Save* vs. *DR* (*Lineitem*)

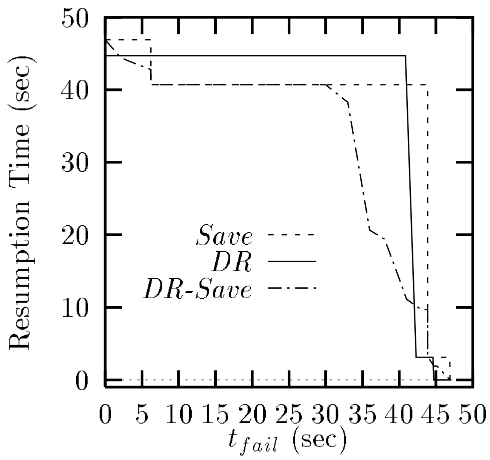

Figure 6.27: *Save* vs. *DR* (*Q3*)

especially when 4 or more batches are used. Hence, when it is possible to divide the input into batches, one must be careful as to how many batches should be formed. A high number of batches results in significant normal operation overhead. On the other hand, a low number of batches results in a longer (average) resumption time

# Batches	Load Time (s)	% Increase Load Time
1	94.7	0%
2	97.6	3.1%
3	104.8	7.4%
4	107.0	13.0%
5	113.0	19.3%
10	150.6	59.0%

Table 6.3: Batching Overhead (*Lineitem*)

# Batches	Load Time (s)	% Increase Load Time
1	43.8	0%
2	44.6	1.8%
3	44.9	2.5%
4	49.1	12.1%
5	54.2	23.7%
10	76.2	74.0%

Table 6.4: Batching Overhead (*Q3*)

Experiment 6

In the sixth experiment, we compared the resumption time of *DR* against an algorithm based on batching, denoted *Batch*. The setup of this experiment is similar to the setup in Experiment 4. That is, for *DR* we would load the warehouse using the designed DAG and stop the load after t_{fail} seconds. We then measure the resumption time of *DR*. For *Batch*, we would load the warehouse by processing the input batches in sequence. For *Batch*,

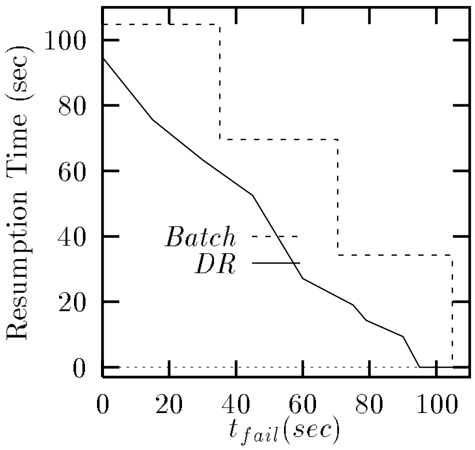

Figure 6.28: *Batch* vs. *DR* (*Lineitem*)

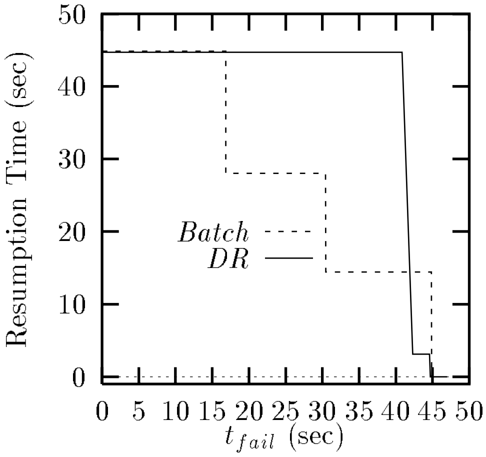

Figure 6.29: *Batch* vs. *DR* (*Q3*)

we used three input batches so that the normal operation overhead is tolerable. We then measure the resumption time of *Batch* based on the input batches that have been processed completely.

The result for the *Lineitem* DAG is shown in Figure 6.28 which plots the resumption time of *DR* and *Batch* as t_{fail} is increased. The graph shows that *Batch*'s resumption time improves in discrete steps. For instance, when $t_{fail} < 36$ seconds, the first input batch has not been processed completely. During resumption, the output based on the first input batch is discarded, and the first input batched is reprocessed. Once $t_{fail} > 36$ seconds, the first input batch has been processed completely and does not need to be reprocessed during resumption. For the *Lineitem* DAG, *DR* is surprisingly more efficient than *Batch* in resuming the load given that *DR* does not impose any normal operation overhead.

The result for the *Q3* DAG is shown in Figure 6.29. Again, *Batch*'s resumption time improves in discrete steps based on the input batches that have been processed completely. The performance of *DR* was already explained in Experiment 4 for this DAG. As Figure 6.29 shows, *Batch* performs better than *DR* for this DAG. The resumption time can also be improved by combining *DR* and *Batch* together (as in *DR-Save*). However, for the *Q3* DAG, the improvement is negligible.

Summary

We can draw a number of conclusions from the previous experiments.

- *DR* resumes a failed load much more efficiently than *Redo* and *Inf*. *DR* is also flexible in that the more properties exist, the more choices *DR* has and the better *DR* performs.
- There is a need for a "cost-based" analysis of when to use *DR*. For instance, if the warehouse table is empty, *Redo* is better than both *DR* and *Inf*. However, as more tuples are loaded, using *DR* becomes more and more beneficial. Another reason why a "cost-based" analysis is needed is that in some cases, subset filters may not remove enough tuples to justify the cost that the subset filters impose when a load is resumed (*e.g.*, cost of performing an anti-semijoin).
- In many cases, savepoints (or snapshots) result in a significant normal operation overhead. When a batching algorithm is used, a careful selection of the number of input batches is required because a batching algorithm can result in a significant normal operation overhead. However, if certain transforms of a component DAG are very selective (*i.e.*, few output tuples compared to input tuples), the overhead of savepoints may be tolerable.
- For component DAGs that load dimension and fact tables, *DR*, despite having no normal operation overhead, resumes the load more efficiently than algorithms that employ savepoints or batching. On the other hand, for component DAGs that do not produce warehouse tuples immediately, using savepoints after very selective transforms may be beneficial. In this case, a hybrid algorithm that combines *DR* and the savepoint-based algorithm can be used. For component DAGs that are simple enough (so that input batches can be formed) but do not produce warehouse tuples immediately, a batching algorithm may be best.

6.8 Chapter Summary

We developed a warehouse load resumption algorithm *DR* that performs most of its actions during "design time," and imposes no overhead during normal operation. The *Design* portion of *DR* only needs to be invoked once, when the warehouse load component DAG is designed, no matter how many times the *Resume* portion is called to resume from a failure. *DR* is novel because it uses only properties that describe how complex transforms process their input at a high level (*e.g.*, Are the tuples processed in order?). These properties usually can be deduced easily from the transform specifications, and some of them (*e.g.*, keys, ordering) are already declared in current warehouse load packages. By performing experiments under various TPC-D scenarios using Sagent's load facility, we showed that DR leads to very efficient resumption.

DR can also be used to identify "problem spots" in a component DAG, and suggest modifications to make resumption more efficient. For instance, in our example component DAG, transform TV needs to reprocess all of its input because DR finds that there are no identifying attributes. Further, TV's output is a single tuple, suggesting that saving the result of TV is useful.

Although we have developed DR to resume warehouse loads, DR is useful for many applications. In particular, if an application performs complex and distributed processing, DR is a prime recovery algorithm candidate when minimal overhead is required. Since previous algorithms either require heavy overhead during normal operation, or incur high recovery cost, DR fills the need for an efficient lightweight recovery algorithm.

Chapter 7

WHIPS: A Data Warehouse System Prototype

7.1 Introduction

In this chapter, we discuss the WHIPS (WareHouse Information Processing System) prototype we have developed at Stanford. The goal of WHIPS is to serve as a framework for prototyping and experimenting with our techniques for efficiently updating the data warehouse. WHIPS is not a new database management system. It is a distributed application that runs on top of the database, and that manages view maintenance. WHIPS uses a commercial RDBMS as its back-end database. Because of the flexibility of the back-end RDBMS, there are many possible alternatives for representing and maintaining the warehouse views.

In this chapter, we identify specific areas in which the back-end RDBMS provides WHIPS flexibility in terms of view maintenance. These decision areas are often encountered by developers of view maintenance software. Furthermore, any developer that is writing software that needs to do bulk-updates on RDBMS tables will encounter the same decision areas.

An example of such a decision area is how to install the changes to a view once the changes are computed. Recall that in Chapter 3, we introduced the notion of a VDAG "strategy," which is a sequence of *Comp* expressions (for computing changes) and *Inst* expressions (for installing changes). In building WHIPS, we realized that there are many ways of implementing the *Inst* expression. In Chapter 3, we focused on developing the

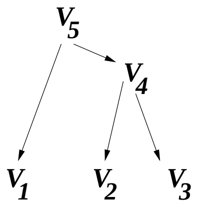

Figure 7.1: Conceptual Representation

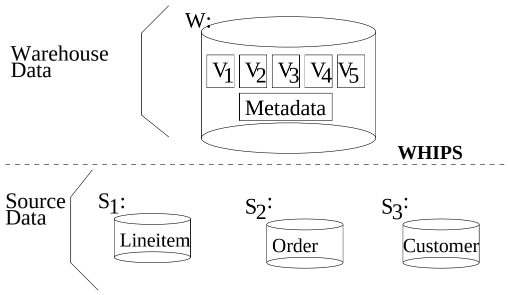

Figure 7.2: Physical Representation

higher-level algorithms for choosing efficient VDAG strategies, but we did not focus on the details of executing the *Inst* expression. As we will see, there are many implementation possibilities.

In this chapter, we discuss the decisions that were made when developing WHIPS, and show experimentally why all of the decisions were reasonable ones. We present experiments that show that by making the right decisions, WHIPS updates the warehouse much faster than if we had made the wrong decisions. We begin the chapter with an overview of the WHIPS architecture.

7.2 WHIPS Architecture

WHIPS is a data warehousing system that incrementally maintains the warehouse data. Before we describe the components of WHIPS, we first discuss how the warehouse data is conceptually modeled and physically stored by WHIPS.

7.2.1 Data Representation

The warehouse data in WHIPS is conceptually modeled using a VDAG. Figure 7.1 shows a simple example of a VDAG with three base views (*i.e.*, V_1,V_2,V_3) and two derived views (*i.e.*, V_4, V_5). The source data from which the base views V_1, V_2, and V_3 are derived from are not in the VDAG. Figure 7.2 shows how WHIPS physically stores the data represented by the VDAG in Figure 7.1. WHIPS stores the views V_1 through V_5 as tables in its back-end RDBMS. WHIPS also keeps metadata in the RDBMS that records each view's definition. All the information in the VDAG is captured in the metadata.

The metadata also stores information about the source data from which the base views are derived. In WHIPS, each base view is defined over source data using a single `SELECT-FROM-WHERE` (`SFW`) SQL statement. This simple base view definition language allows the warehouse designer to filter and combine source data by using appropriate selection and join conditions in the `WHERE` clause. Aggregations are not permitted in base view definitions.

Each derived view is defined over other warehouse views using one or more `SELECT-FROM-WHERE-GROUPBY` (`SFWG`) SQL statements. Multiple `SFWG` statements may be combined using a `UNION ALL` SQL operator. Aggregations can be used in derived view definitions.

EXAMPLE 7.2.1 Let us suppose that there are three remote information sources S_1, S_2 and S_3 as shown in Figure 7.2. Let the TPC-D tables *Lineitem*, *Order*, and *Customer* reside in S_1, S_2 and S_3 respectively. Base views V_1, V_2, and V_3 at the warehouse can be defined as projections over $S_1.Lineitem$, $S_2.Order$ and $S_3.Customer$ as follows.

`CREATE VIEW` V_1 `AS`
`SELECT` $orderID$, $partID$, qty, $cost$ `FROM` $S_1.Lineitem$

`CREATE VIEW` V_2 `AS`
`SELECT` $orderID$, $custID$, $date$ `FROM` $S_2.Order$

`CREATE VIEW` V_3 `AS`
`SELECT` $custID$, $name$, $address$ `FROM` $S_3.Customer$

Of course, selection and join operations can be specified as well in the definitions of V_1, V_2 and V_3. The derived view V_4 may be defined to count the number of orders each customer has made in 1998.

`CREATE VIEW` V_4 `AS`
`SELECT` $custID$, `COUNT`$(*)$
`FROM` V_2, V_3
`WHERE` $V_2.custID = V_3.custID$ `AND` $V_2.date >= 1/1/98$ `AND` $V_2.date < 1/1/99$
`GROUPBY` $custID$

□

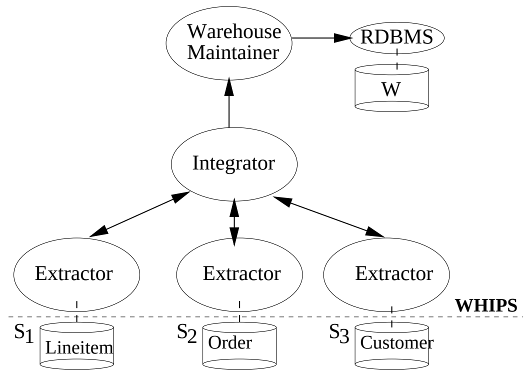

Figure 7.3: WHIPS Components

7.2.2 Overview of WHIPS Components

Three types of components comprise the WHIPS system — the Extractor, the Integrator and the Warehouse Maintainer. As mentioned previously, the WHIPS system also relies on a back-end RDBMS to store the warehouse data. The WHIPS components, along with the RDBMS, are shown in Figure 7.3. We now discuss the WHIPS components by giving an overview of how the warehouse data is maintained when source data changes.

The Extractor component periodically detects the changes to the various source data. In WHIPS, a Extractor component is constructed for each remote information source. Each table or each file that is referred to in the `FROM` clauses of the base view definitions is monitored. Hence, in Figure 7.3, there are three Extractor components, and each one is assigned to one of the remote information sources (S_1, S_2 or S_3) in the working example. For instance, the Extractor assigned to S_1 detects the changes to the *Lineitem* table that resides in S_1.

One option for the Extractor component is to use the sort-merge outerjoin or window algorithms developed in Chapter 2. In this case, WHIPS takes as input a source specification that includes the schema, which algorithm to use, and the period of change detection.

Periodically, the Integrator receives the deltas detected by the Extractor components, and computes deltas to the base views stored in the warehouse. WHIPS assumes that the sources are autonomous. Hence, unlike view maintenance in a centralized environment, *update anomalies* may take place. The Integrator component uses the algorithms developed in [ZGMHW95] to ensure that the deltas computed for the base views are consistent with each other. The notion of consistency guaranteed by the WHIPS Integrator is defined in [ZGMHW95]. The WHIPS Integrator does not perform any data cleansing that can often be done by commercial cleansing tools. The WHIPS Integrator may need to send queries back to the Extractor components to compute the base view deltas. Hence, in Figure 7.3, the bidirectional edges between the Integrator and Extractor components indicate that messages and data are exchanged between the two components.

The Warehouse Maintainer component receives the base view deltas from the Integrator, and computes the deltas to the derived views. The Warehouse Maintainer then updates the materialized views based on the computed deltas. The Warehouse Maintainer component uses the dual-stage VDAG strategies discussed in Chapter 3 for updating the warehouse VDAG. To compute the derived view deltas and update the materialized views, the Warehouse Maintainer sends a sequence of queries and other DML (Data Manipulation Language) commands to the RDBMS. The queries are used for computing the deltas while the DML commands (SQL `INSERT` and `DELETE` commands, cursor updates) are used for updating the materialized views.

The interaction among the WHIPS components is similar when the warehouse data is first initialized. First, the Extractor component identifies the source data that is needed in populating the warehouse based on the `SFW` view definitions of the base views. The Integrator then computes a consistent set of initial base view data based on the source data extracted by the Extractor components. The Warehouse Maintainer will then send a sequence of DDL (Data Definition Language) and DML commands to the RDBMS to create the materialized views and populate them. The DDL commands (*e.g.*, the `CREATE TABLE` SQL command) are used for creating the materialized views. The DML commands are used to populate the materialized views.

In the next section, we discuss the Warehouse Maintainer component in more detail. We discussed the change detection algorithms employed by the Extractor component in Chapter 2, as well as other methods of change detection. We refer the reader to [ZGMHW95] for further discussion of the Integrator component.

7.3 Warehouse Maintainer

The Warehouse Maintainer is the software component that is responsible for initializing and maintaining the warehouse data. Recall that in WHIPS, a back-end RDBMS is used for

orderID	*partID*	*qty*	*cost*
1	*a*	1	20
1	*b*	2	250
1	*a*	1	20

Figure 7.4: DUP Representation (V_1^{dup})

orderID	*partID*	*qty*	*cost*	*dupcnt*
1	*a*	1	20	2
1	*b*	2	250	1

Figure 7.5: COUNT Representation (V_1^{count})

storing the warehouse data. Because of the flexibility of the back-end RDBMS, there are many possible ways for representing and maintaining the views. In Sections 7.3.1 through 7.3.3, we identify specific areas in which the back-end RDBMS provides the Warehouse Maintainer flexibility in terms of view creation and maintenance. For each area, we discuss the approach taken by the WHIPS Warehouse Maintainer. We summarize in Section 7.3.4.

7.3.1 View Representation

Since the views in WHIPS are defined using SQL `SFWG` statements and SQL supports bag semantics, each view can contain a bag of tuples. There are two ways to represent a bag of tuples. One representation, which we call the *DUP* representation, simply keeps the duplicate tuples as shown in Figure 7.4. Another representation, which we call the *COUNT* representation, keeps one copy of each unique tuple but keeps track of the number of duplicates in a special *dupcnt* field as in Figure 7.5. Let us denote a view V's COUNT representation as V^{count} and its DUP representation as V^{dup}.

Clearly, the COUNT representation has lower storage costs if there are a lot of duplicates and if the rows in V are large enough so that the storage overhead of having a *dupcnt* field is not significant. The reduction in storage achieved by using V^{count} instead of V^{dup} may speed up selection, join and aggregation operations on V. Thus, any maintenance expression that uses these operations is potentially faster when the underlying views are in the COUNT representation. However, projections may be slowed down when the COUNT representation is used. To see this, let us consider the following operation that lists the *orderID*'s in V_1^{count}.

`SELECT` *orderID* `FROM` V_1^{count}

Clearly, the answer to the above query is not in COUNT representation. For the answer to be in COUNT representation, we need to group the tuples with matching *orderID* attribute values and sum up their *dupcnt* values as follows:

SELECT $orderID$, SUM($dupcnt$) AS $dupcnt$ FROM V_1^{count} GROUPBY $orderID$

Thus, whenever a projection operation is used, an aggregation operation is necessary to produce an answer in COUNT representation.

Using the COUNT representation can also significantly slow down the installation of the insertions to a view V. Under the DUP representation, the tuples in $\triangle V^{dup}$ are inserted into V^{dup} using a single SQL `INSERT` DML command. Under the COUNT representation, each inserted tuple t_{ins} in $\triangle V^{count}$ results in either an update or an insertion to V^{count}. If there is a tuple in V^{count} that matches t_{ins}, we increment the matching V^{count} tuple's $dupcnt$ by $t_{ins}.dupcnt$. Otherwise, we insert t_{ins} into V^{count}. The entire $\triangle V^{count}$ can be processed using one `UPDATE` statement and one `INSERT` statement, both with expensive correlated subqueries. There are other ways to install $\triangle V^{count}$, but installing $\triangle V^{dup}$ is always much simpler.

On the other hand, installing the deletions of a view V, denoted $\bigtriangledown V$, basically requires performing a join between V and $\bigtriangledown V$ under both the COUNT and DUP representations. (There are other complications in installing deletions under the DUP representation, which we return to in Section 7.3.2.) If the COUNT representations of V and $\bigtriangledown V$ are much smaller than their DUP representations, then installing deletions under the COUNT representation may be faster because less data is processed.

WHIPS Approach: Although it seems that the COUNT representation has many benefits, we use the DUP representation in WHIPS for the following reasons.

- It is often the case that the views have keys and do not have duplicates. For instance, dimension tables and fact tables, which are modeled as base views, often have keys. Summary tables (or derived views) often perform group-by operations and the group-by attributes are the keys of the summary tables. Thus in many cases, the views will not have duplicates. In these cases, the DUP representation will incur lower storage costs than the COUNT representation because the DUP representation does not use $dupcnt$ fields.

- As we will show experimentally in Section 7.4, installing insertions under the COUNT representation is significantly more expensive than installing insertions under the DUP representation. Since insertions are the most common changes in data warehousing applications, it is important in WHIPS to install insertions efficiently.

- As we will show experimentally in Section 7.4, even when there are duplicates, computing the deltas of the views under the COUNT representation does not significantly outperform delta-computation under the DUP representation. In fact, if the average number of duplicates is low (*i.e.*, 2 or 3), using the DUP representation may be better.

Henceforth, V refers to the DUP representation V^{dup} unless specified otherwise.

7.3.2 Deletion Installation Under DUP Representation

If V does not have duplicates, the deletions to V, denoted $\bigtriangledown V$, can be installed using a single SQL `DELETE` DML command. We illustrate the command required by installing deletions to V_1 in our working example.

`DELETE FROM` V_1
`WHERE` ($V_1.orderID$, $V_1.partID$) `IN` (`SELECT` $orderID$, $partID$ `FROM` $\bigtriangledown V_1$)

The above statement assumes $\langle orderID, partID\rangle$ comprise V_1's key. Clearly, the `WHERE` clause can be changed appropriately to handle keys with arbitrary number of attributes. For conciseness, we denote the method of installing deletions using a single SQL `DELETE` statement as *SQL-delete*.

Unfortunately, SQL-delete is incorrect when V_1 has duplicates. Even assuming V_1 has only the two attributes $orderID$ and $partID$, the above example `DELETE` statement may delete more tuples than necessary. That is, for each tuple t_{del} in $\bigtriangledown V_1$, *all* (instead of just one) of the V_1 tuples that match t_{del} on $orderID$ and $partID$ are deleted. This is the semantics of the `DELETE` statement under the SQL standard.

Hence in general, to install $\bigtriangledown V$, a cursor on $\bigtriangledown V$ is required. For each tuple t_{del} examined by the cursor, a cursor on V is instantiated to find the first V tuple t that matches t_{del}.[1] Only tuple t is deleted from V. For conciseness, we call this method *cursor-delete*.

WHIPS Approach: While cursor-delete is necessary when V has duplicates, it seems like overkill when V has no duplicates. Thus, in WHIPS, we use SQL-delete in cases where V is guaranteed to have no duplicates. For instance, if V is defined by performing group-by operations, then the group-by attributes are guaranteed to be V's key. If V can have duplicates, a cursor-delete is used to install V's deletions. We will compare the performance of cursor-delete and SQL-delete in Section 7.4.

[1]The cursor on V can be avoided if the back-end RDBMS supports queries on row ID's, or provides some mechanism of restricting the number of rows processed by queries.

orderID	*partID*	*qty*	*cost*
1	*a*	1	20
1	*b*	2	250
2	*a*	1	20
3	*c*	1	500

Table 7.1: V_1

partID	*revenue*	*cnt*
a	40	2
b	500	1
c	500	1

Table 7.2: *ByParts*

7.3.3 Maintenance Expressions

The maintenance expression of any view defined using an SQL `SFW` statement (without sub-queries) is well known ([GL95]) and we do not discuss it here. For views defined using SQL `SFWG` statements (*i.e.*, views with group-by operations and aggregations), they can be maintained using the *summary-delta* algorithm [MQM97]. We now illustrate how summary-deltas are computed and installed.

EXAMPLE 7.3.1 In this example, let us suppose that view V_1 contains the tuples shown in Table 7.1. View *ByParts* is defined over V_1 to group the V_1 tuples by *partID*. (View *ByParts* was not in the previous examples.) The revenue of each part is reported in *ByParts* by summing up the product of *qty* and *cost* for each order for that particular part. Also, *ByParts* counts the number of V_1 tuples that are used to derive each *ByParts* tuple. This *cnt* field is useful in determining when a *ByParts* tuple t needs to be deleted because all of the V_1 tuples that derive t were deleted from V_1. If the *cnt* field is not included in *ByParts*'s definition, WHIPS automatically modifies the view definition to include the *cnt* field to ensure that *ByParts* can be incrementally maintained. The view definition of *ByParts* is as follows.

CREATE VIEW *ByParts* AS
SELECT *partID*, SUM($qty * price$) AS *revenue*, COUNT(*) AS *cnt*
FROM V_1
GROUPBY *partID*

The tuples in *ByParts* are shown in Table 7.2.

Let us suppose that the tuples shown in Table 7.3 are to be inserted into V_1, and the ones shown in Table 7.4 are to be deleted. Note that tuple $\langle 1, a, 1, 20\rangle$ in $\bigtriangledown V_1$ and the tuple $\langle 1, a, 2, 20\rangle$ in $\triangle V_1$ together represent an update wherein the *qty* of *a* parts purchased in the first order ($orderID = 1$) is increased from 1 to 2.

In the summary-delta algorithm, a *compute phase* is used to determine the "effect" of $\triangle V_1$ and $\triangledown V_1$ on $ByParts$. The effect is captured in a summary-delta denoted $ByParts_{SD}$ and computed as follows.

```
SELECT partID, SUM(revenue) AS revenue, SUM(cnt) AS cnt
FROM (
      (SELECT partID, SUM(qty * price) AS revenue, COUNT(*) AS cnt
      FROM △V1
      GROUPBY partID)
   UNION ALL
      (SELECT partID, -SUM(qty * price) AS revenue, -COUNT(*) AS cnt
      FROM ▽V1
      GROUPBY partID))
```

The summary-delta basically applies the group-by and aggregation operations specified in the definition of $ByParts$ on $\triangle V_1$ and $\triangledown V_1$. Note that the aggregate values computed from $\triangledown V_1$ are negated to reflect the effect of deletions on the SUM, COUNT and AVG functions. (Functions MAX and MIN cannot be incrementally maintained in general.)

Given the $\triangle V_1$ and $\triangledown V_1$ shown in Tables 7.3 and 7.4, the summary-delta is shown in Table 7.5. Tuple $\langle a, 20, 0\rangle$ affects $ByParts$ by changing the $ByParts$ tuple for part a, and increasing the tuple's *revenue* by 20 and the tuple's *cnt* by 0. This procedure makes sense because as mentioned earlier, the number of a parts in the first order was increased from 1 to 2. Hence, the *revenue* for a parts must increase by 20. The *cnt* is unchanged because there is still the same number of V_1 tuples that derive the $ByParts$ tuple for part a.

Tuple $\langle b, -500, -1\rangle$ of the summary-delta states that for the tuple with a $partID$ of b, its *revenue* must be decreased by 500, and its *cnt* is decreased by 1. Intuitively, this is due to the deletion of tuple $\langle 1, b, 2, 250\rangle$ from V_1 (see $\triangledown V_1$). Also, the $ByParts$ tuple for part b needs to be deleted if its *cnt* attribute becomes zero after it is decreased by 1. This makes sense because all of the V_1 tuples that the $ByParts$ tuple for part b was derived from were deleted.

Tuple $\langle c, 500, 1\rangle$ of the summary-delta states that for the tuple with a $partID$ of c, its *revenue* must be increased by 500, and its *cnt* is increased by 1. Intuitively, this is due to the insertion of tuple $\langle 4, c, 1, 500\rangle$ into V_1 (see $\triangle V_1$).

Finally, tuple $\langle d, 30, 1\rangle$ of the summary-delta needs to be inserted into $ByParts$ since there is no $ByParts$ tuple with a $partID$ of d so far. The *install phase* of the summary-delta

orderID	*partID*	*qty*	*cost*
1	*a*	2	20
4	*c*	1	500
4	*d*	1	30

Table 7.3: $\triangle V_1$

orderID	*partID*	*qty*	*cost*
1	*a*	1	20
1	*b*	2	250

Table 7.4: $\triangledown V_1$

partID	*revenue*	*cnt*
a	20	0
b	-500	-1
c	500	1
d	30	1

Table 7.5: $ByParts_{SD}$

algorithm changes *ByParts* by instantiating a cursor on the summary-delta, and applying the appropriate changes to *ByParts* based on the current tuple examined by the cursor. □

There are two problems in using the summary-delta algorithm. First, as illustrated in the previous example, the install phase of the summary-delta algorithm is tuple-oriented. Since the compute phase processes delta tables (*e.g.*, $\triangle V_1$ and $\triangledown V_1$) which are relatively small, computing the summary-delta is potentially fast, and the install can become the bottleneck. Reference [Qua97] developed a method for alleviating this first problem. (The method developed is discussed in Section 7.5.) The second problem is that when the algorithm (as presented in [MQM97]) is used to maintain a view V, the algorithm does not explicitly compute the insertions and deletions to V. In the example, delta tables $\triangle ByParts$ and $\triangledown ByParts$ are not explicitly computed by the summary-delta algorithm. Delta tables $\triangle ByParts$ and $\triangledown ByParts$ would be useful if there are views defined on *ByParts* that need to be maintained due to the changes to *ByParts*.

WHIPS Approach: For a view V defined using a `SFW` statement, WHIPS uses standard maintenance expressions ([GL95]) to incrementally maintain V. For `SFWG` views, WHIPS does not use the summary-delta algorithm. Instead, WHIPS uses an algorithm that has a more efficient install phase than the summary-delta algorithm. However, the algorithm we use may process more tuples during the compute phase. Another advantage of our algorithm is that for each `SFWG` view V, it computes a $\triangle V$ and a $\triangledown V$ that can easily be used to compute changes of the views defined on V. The next example illustrates the algorithm employed by WHIPS.

EXAMPLE 7.3.2 In this example, we show how $ByParts$ is maintained in WHIPS. First, the summary-delta is computed as before (see Table 7.5). Using the summary-delta (*i.e.*, $ByParts_{SD}$), $\bigtriangledown ByParts$ is computed as follows.

```
INSERT INTO ▽ByParts
SELECT *
FROM ByParts
WHERE partID IN (SELECT partID FROM ByParts_SD)
```

That is, any $ByParts$ tuple that is affected by the summary-delta is inserted into $\bigtriangledown ByParts$, and deleted later from $ByParts$ in the install phase. Given the summary-delta shown in Table 7.5, the resulting $\bigtriangledown ByParts$ is shown in Table 7.6. Since any `SFWG` aggregate view like $ByParts$ has no duplicates, $\bigtriangledown ByParts$ can be installed efficiently using SQL-delete (Section 7.3.2).

Note that the SQL `INSERT` statement above used a join operation to compute $\bigtriangledown ByParts$. The delta table $\bigtriangledown ByParts$ is then installed using an SQL-delete that also uses a join operation. The join operations employed by the WHIPS method are specified declaratively and can be processed in a set-oriented fashion. On the other hand, recall that the join operation used by the summary-delta algorithm is specified using a cursor over one of the join operands. This results in a tuple-oriented processing of the join operation.

The insertions $\bigtriangleup ByParts$ are computed by applying $ByParts_{SD}$ to $\bigtriangledown ByParts$. First, we take the union of $ByParts_{SD}$ and $\bigtriangledown ByParts$. The result of this union is shown in Table 7.7. Then, we group the tuples by $partID$, sum up their $revenue$ values and cnt values, and filter out those groups with cnt less than one. The $\bigtriangleup ByParts$ computed is shown in Table 7.8. For instance, the tuple $\langle a, 40, 1\rangle$ in $\bigtriangleup ByParts$ is obtained by combining the two tuples for part a in Table 7.7. On the other hand, there is no tuple for part b in $\bigtriangleup ByParts$ since the sum of the cnt values in Table 7.7 for part b is zero. Again, a cnt of zero implies that the $ByParts$ tuple for part b is to be deleted. The SQL statement below can be used to compute $\bigtriangleup ByParts$.

```
SELECT partID, SUM(revenue) AS revenue, SUM(cnt) AS cnt
FROM (
```

$partID$	$revenue$	cnt
a	20	1
b	500	1
c	500	1

Table 7.6: $\bigtriangledown ByParts$

$partID$	$revenue$	cnt
a	20	0
b	-500	-1
c	500	1
d	30	1
a	20	1
b	500	1
c	500	1

Table 7.7: $ByParts_{SD} \cup \bigtriangledown ByParts$

$partID$	$revenue$	cnt
a	40	1
c	1000	2
d	30	1

Table 7.8: $\bigtriangleup ByParts$

(SELECT *
FROM $ByParts_{SD}$)
UNION ALL
(SELECT *
FROM $\bigtriangledown ByParts$))
GROUPBY $partID$
HAVING SUM(cnt) > 0

The tuples in $\bigtriangleup ByParts$ can then be inserted using a single SQL `INSERT` statement; no cursors are needed. □

Although the example illustrated how WHIPS maintains a specific aggregate view V, it is not hard to show that the WHIPS strategy can handle views with arbitrary combinations of `SUM`, `COUNT` and `AVG` aggregate functions. Just like the summary-delta algorithm, WHIPS will in general need to recompute aggregate views with `MAX` or `MIN`.

To summarize, given an aggregate view V, WHIPS computes $\bigtriangleup V$ and $\bigtriangledown V$ explicitly, which enable WHIPS to perform the install phase using SQL-delete and a single SQL `INSERT` statement. In comparison, the summary-delta algorithm uses a cursor to apply custom changes to V depending on whether the tuple in the summary-delta of V is an insertion, an update, or a deletion.

In Section 7.4, we compare the performance of the summary-delta algorithm against that of the algorithm used by WHIPS. We also compare these incremental maintenance algorithms against fully and partially recomputing the `SFWG` view from scratch.

7.3.4 Summary

In the previous sections, we have identified three areas in which for WHIPS we needed to make critical decisions on view maintenance.

1. Choice of view and delta table representation (Section 7.3.1)
2. Choice of installation algorithm (Sections 7.3.1 and 7.3.2)
3. Choice of maintenance algorithm and expressions (Section 7.3.3)

We believe that in WHIPS we have made sound choices in the areas listed above. Our maintenance algorithm (*i.e.*, the third area) could be improved further by considering key and referential integrity constraints. Constraints were not taken into account in WHIPS because the rewriting of the maintenance expressions in the presence of constraints can be done by the underlying query optimizer, as demonstrated in [Vis98].

7.4 Experiments

In Section 7.3, we discussed the three areas in which we needed to make critical decisions for the WHIPS prototype. We now evaluate the various decisions that were made in WHIPS through experiments.

Experiments outline: The outline of the experiments is as follows.

1. *View representation.* In Section 7.4.1, we evaluate the performance of computing and installing deltas under the COUNT and DUP representations.
2. *Deletion installation.* In Section 7.4.2, we compare the performance of SQL-delete and cursor-delete.
3. *Aggregate maintenance expressions.* In Section 7.4.3, we compare the performance of the summary-delta and the WHIPS aggregate-view maintenance algorithm. We also compare these algorithms against fully and partially recomputing aggregate views from scratch.

Performance measurement strategy: Recall that WHIPS sends a sequence of queries, DML statements, and cursor-fetches to the back-end RDBMS to maintain the data warehouse. In the experiments, we measured the time it took the back-end RDBMS to run the commands sent by WHIPS. It is reasonable to focus on this time because it represents the bulk of the time spent in updating the data warehouse. For instance, WHIPS needs to traverse the data structure representing the VDAG to determine the order in which the views should be maintained. However, the time it takes for WHIPS to traverse the VDAG is negligible and is ignored in the experiments. Methods that are not used by WHIPS (*e.g.*, installation of deltas under COUNT representation) were simulated. That is, we created a script containing the sequence of commands that would have been sent to the back-end RDBMS had those methods been used by WHIPS. The back-end RDBMS used was Oracle 8.0 running on a Windows NT machine with a Pentium II processor. The size of the buffer cache assigned to Oracle was 234 MB.

Base views used: The base views used in the experiments are copies of the TPC-D tables $Order$ and $Lineitem$. For conciseness, we call the base views O for $Order$ and L for $Lineitem$. The derived views vary from one experiment to the next. A TPC-D scaling of 0.1 was used. Hence, L is about 75 MB, and O is about 17 MB. More specifically, there are 600,000 L tuples, and each tuple is 124 bytes large on average. There are 150,000 O tuples, and each tuple is 113 bytes large on average.

7.4.1 View Representation

In the first experiment, we compared the installation time of the deltas of base view L under both the COUNT and DUP representations. Recall that in WHIPS, we use the DUP representation. In the experiment, we varied the update percentage of L from 1% to 10%. An update percentage of k% implies that $(k/100) \cdot |L|$ tuples are inserted and $(k/100) \cdot |L|$ tuples are deleted, where $|L|$ is the number of tuples in L. The inserted tuples were produced using a program that is supplied with the TPC-D benchmark. The deleted tuples are chosen randomly.

Figure 7.6 shows that installing $\triangle L^{count}$ (insertions of L under the COUNT representation) is significantly more expensive than installing $\triangle L^{dup}$ (insertions of L under the DUP representation). For instance, when the update rate is 10%, installing $\triangle L^{count}$ takes almost 10000 sec, which is two orders of magnitude times longer than the time to install $\triangle L^{dup}$. This is because the tuples in $\triangle L^{dup}$ are simply inserted into L^{dup}. On the other hand, an anti-semijoin (*i.e.*, an SQL `NOT EXISTS` condition) is needed to check if a tuple in $\triangle L^{count}$ is in L^{count} or not, before that tuple can be inserted into L^{count}.

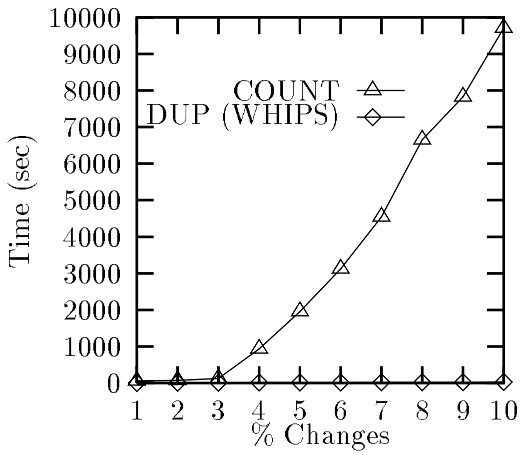

Figure 7.6: Installing $\triangle L$ Without Duplicates

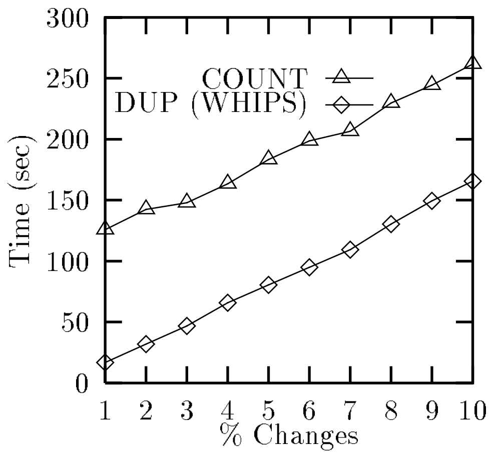

Figure 7.7: Installing $\bigtriangledown L$ Without Duplicates

Figure 7.7 shows that installing $\bigtriangledown L^{count}$ (deletions of L under the COUNT representation) is also more expensive than installing $\bigtriangledown L^{dup}$. However, under both representations, a join between $\bigtriangledown L$ and L is required so that the disparity is not that significant. That is, deletion installation under the DUP representation is "only" about 3 times slower on average.

Although installing deletions requires performing a join just like when $\triangle L^{count}$ is installed, the deletions can be installed more swiftly. This is because the join required in installing deletions can be done faster than the anti-semijoin required in installing $\triangle L^{count}$. The reason is that the anti-semijoin needs to be done on a per-tuple basis – each tuple in $\triangle L^{count}$ is checked to see if there is a matching tuple in L^{count}. On the other hand, the join between $\bigtriangledown L$ and L (under both representations) can be done in a set-oriented fashion.

In the next experiment, we artificially introduced duplicates in L. Note that normally in TPC-D, L has a key and has no duplicates. In this experiment, each L tuple has 3 copies, *i.e.*, L has a *multiplicity* of 3. It is easy to see that as the multiplicity is increased, the COUNT representation is more storage-efficient than the DUP representation in representing duplicates. Thus, one may expect the COUNT representation to outperform the DUP representation in installing $\triangle L$ and $\bigtriangledown L$.

However, installing $\triangle L^{count}$ is significantly slower than installing $\triangle L^{dup}$ even when the multiplicity is increased (Figure 7.8). This is because the benefit of the more concise

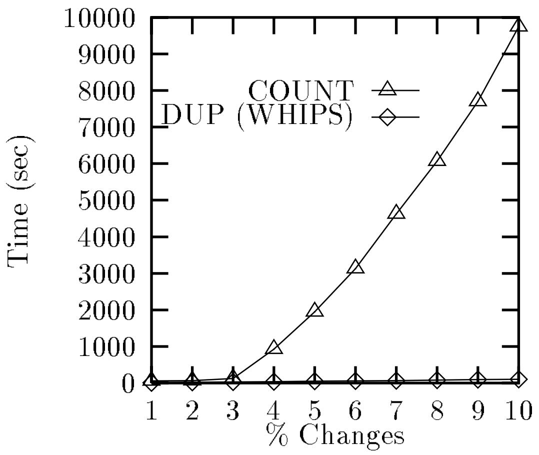

Figure 7.8: Installing $\triangle L$ With Duplicates

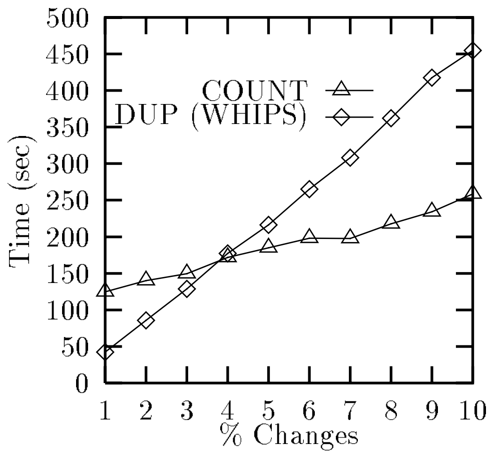

Figure 7.9: Installing $\bigtriangledown L$ With Duplicates

representation afforded by $\triangle L^{count}$ cannot overcome the overhead of having to perform an anti-semijoin between $\triangle L^{count}$ and L^{count}.

Figure 7.9 shows that installing $\bigtriangledown L^{count}$ is faster than installing $\bigtriangledown L^{dup}$ when the update percentage is over 4%. This is because increasing the multiplicity increases the size of $\bigtriangledown L^{dup}$ proportionately, but the size of $\bigtriangledown L^{count}$ is unaffected. Hence, the time to install $\bigtriangledown L^{dup}$ increases proportionately with multiplicity. On the other hand, the time to install $\bigtriangledown L^{count}$ is not affected by increases in multiplicity. (Note that the lines for COUNT are very similar in Figures 7.7 and 7.9.)

The previous experiments focused on the installation of L's deltas. In the next experiment, we compared the performance of computing deltas under the COUNT and DUP representations. In the experiment, we defined a derived view LO that performs a join between L and O. We then measured the time it took to compute $\triangle LO$ and $\bigtriangledown LO$ given $\triangle L$ and $\bigtriangledown L$ under the two representations.

In the normal case where L and O have no duplicates (*i.e.*, multiplicity is equal to 1), computing the deltas of LO under the DUP representation is comparable to computing the deltas of LO under the COUNT representation as shown in Figure 7.10. (The update percentage is 5% for this experiment.) As we artificially increase the multiplicity of L and O, computing the deltas under the COUNT representation becomes faster relative to delta-computation under the DUP representation. This is because the COUNT representations

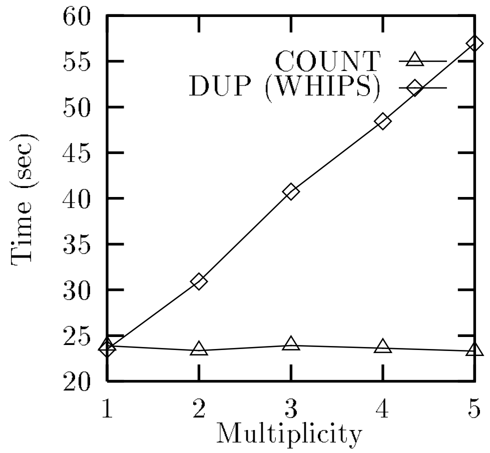

Figure 7.10: Computing $\triangle LO$ and ∇LO

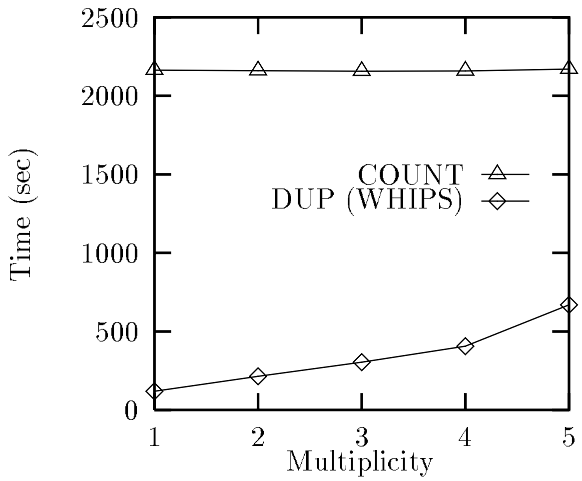

Figure 7.11: Delta-computation and installation

of L and O are about m times smaller than their DUP representations, where m is the average multiplicity of L and O.

Figure 7.11 shows the total time for computing the deltas of LO, and installing the deltas of L and LO. In this graph, an update percentage of 5% was chosen. As the graph shows, computing *and* installing deltas can be done faster under the DUP representation, especially under the common case where there are no duplicates.

7.4.2 Deletion Installation

Assuming a view V uses the DUP representation and assuming V has no duplicates, the deletions of V can be installed using SQL-delete or cursor-delete. (See Section 7.3.2 for relevant discussion.) We compare the performance of these two deletion-installation methods by installing the deletions ∇O to O. In this experiment, we varied the update rate of O from 1% to 10%. Figure 7.12 shows that SQL-delete is much faster than cursor-delete, on average 69 times faster. One possible reason for this disparity is that SQL-delete is a declarative way of installing the deletions. Thus, the query optimizer of the back-end database can easily choose the most effective way of performing the SQL-delete. On the other hand, using a cursor-delete restricts the query optimizer to perform a join between each tuple examined in the cursor and O. As Figure 7.12 shows, the leeway given to the

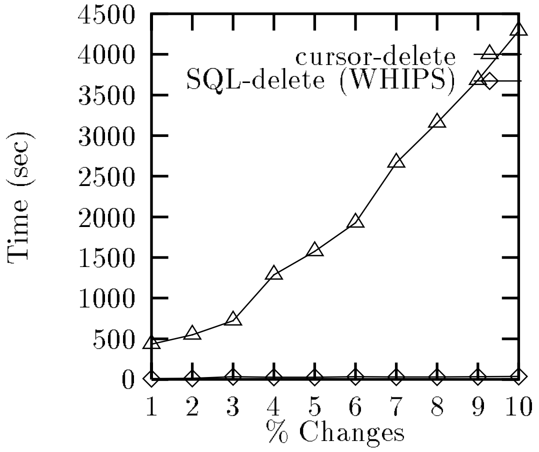

Figure 7.12: Cursor-delete vs. SQL-delete

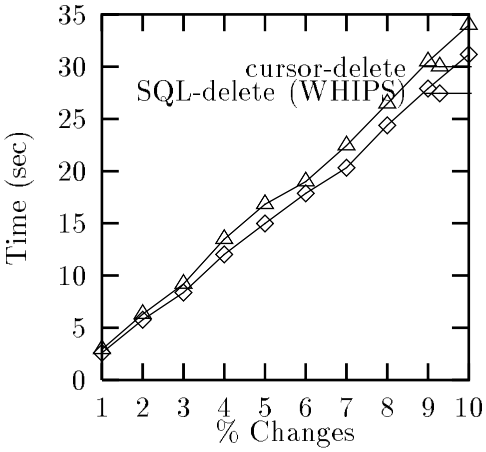

Figure 7.13: Cursor-delete vs. SQL Delete (with index)

query optimizer translates into significant benefits.

An index on the key of O (*i.e.*, $orderID$) can help speed up the join between O and $\triangledown O$ performed by the two deletion-installation methods. When this index is built, the disparity between cursor-delete and SQL-delete is reduced significantly as shown in Figure 7.13, where SQL-delete is only 1.10 times faster than cursor-delete.

7.4.3 Aggregate Maintenance Expressions

In Section 7.3.3, we discussed the summary-delta and the WHIPS algorithms for incrementally maintaining aggregate views. We compare both incremental maintenance algorithms against *full recomputation* and *partial recomputation* of the aggregate view. Full-recomputation simply recomputes the contents of the aggregate views based on the new state of the underlying view(s). On the other hand, partial-recomputation first determines which of the tuples in the aggregate view are "affected" by the deltas and removes them. Then partial-recomputation will recompute the new values for the affected aggregate view tuples and insert new tuples into the aggregate view.

We compare these four techniques experimentally by considering two aggregate views defined on L. The first view, V_{many}, groups tuples based on the *orderkey* attribute of L. This results in 150,000 groups that are contained in V_{many}. View V_{many} also contains

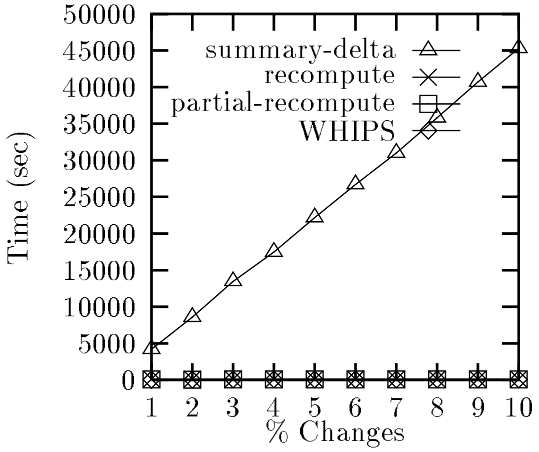

Figure 7.14: Maintaining Aggregate View V_{many}

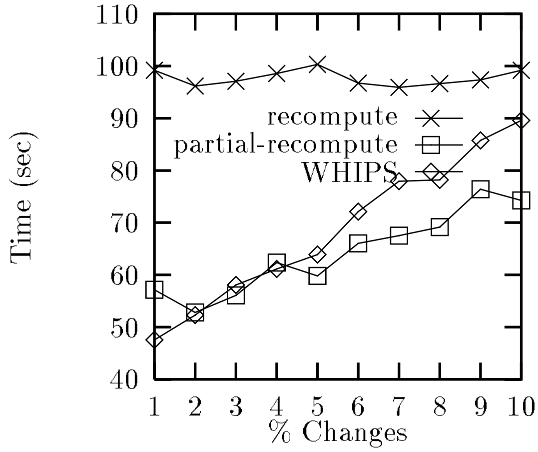

Figure 7.15: Maintaining V_{many}

aggregate values resulting from SQL `AVG`, `SUM`, and `COUNT` functions. The second view, V_{few}, has very few groups (7 groups) because it groups by the *linenumber* attribute of L. Like V_{many}, V_{few} also contains aggregate values resulting from SQL `AVG`, `SUM`, and `COUNT` functions.

In the first experiment, we focused on V_{many} and did not build any indices on V_{many}. Figure 7.14 shows that using the summary-delta algorithm can be disastrous. This is because the summary-delta algorithm must process the tuples one at a time, and for each tuple, a scan of V_{many} is required. The performance of the other three algorithms is shown more clearly in Figure 7.15. The figure shows that full-recomputation performs the worst among the three algorithms. Surprisingly, as the update percentage is increased, partial-recomputation performs better relative to the WHIPS maintenance algorithm.

In the next experiment, indices were built on the group-by attributes of V_{many}. Figure 7.16 shows that the performance of the summary-delta algorithm greatly improves. For each tuple in the summary-delta, the algorithm no longer needs to scan V_{many} because of the presence of the index. By comparing the lines for the WHIPS algorithm in Figures 7.15 and 7.16, we see that the performance of the WHIPS algorithms improves slightly when the index on the group-by attributes is built, as was to be expected. Although the indices help in the installation of V_{many}'s deletions, an additional cost is incurred in having to update the index for each insert and delete. This overhead degrades the performance of the

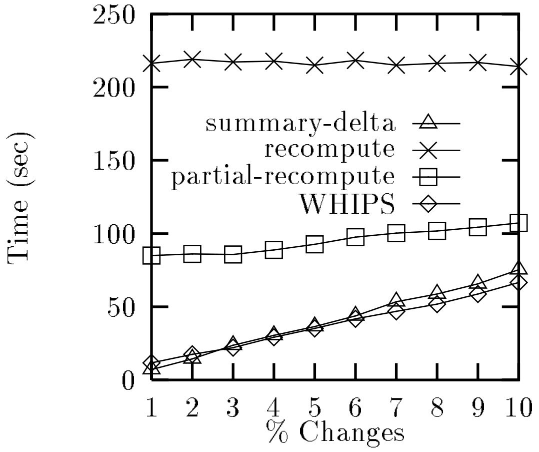

Figure 7.16: Maintaining V_{many} with Indices

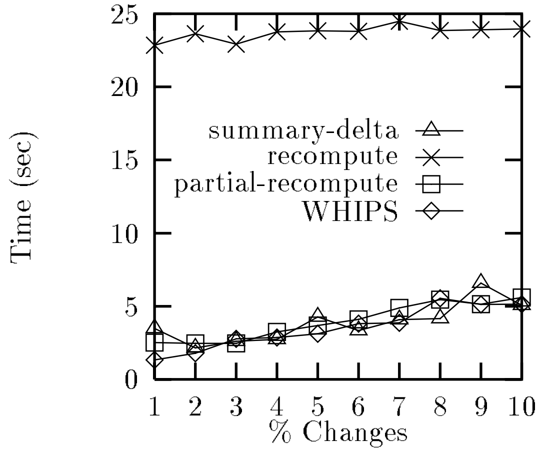

Figure 7.17: Maintaining V_{few}

partial-recomputation and the full-recomputation algorithms. Overall, the summary-delta algorithm is competitive with the WHIPS algorithm when an index is built on the aggregate view's group-by attributes.

In the next experiment, we used the various algorithms to update V_{few} (*i.e.*, aggregate view with 7 groups). Because V_{few} is so small, there is very little difference among the WHIPS, summary-delta and partial-recompute algorithms. For instance, the cost of scanning V_{few} for each summary-delta tuple is small because V_{few} only has 7 tuples. Thus, it is to be expected that the summary-delta algorithm performs as well as WHIPS. Note however that even if V_{few} is small, full-recompute is still significantly more expensive than the other three algorithms. This is because full-recomputation still needs to process all of the tuples in L in order to recompute V_{few}.

7.5 Related Work

We have reviewed in previous chapters a significant amount of research devoted to view maintenance. See [GM95] for a recent survey. However, to our knowledge, there has not been a paper that discusses the very important low-level details of view maintenance that are the focus of this chapter. For instance, [LYGM99] assumes that there is an *Inst* operation for installing the changes into a view, but does not cover the important details of how to implement *Inst*.

Reference [MQM97] proposes the summary-delta algorithm but does not discuss whether to compute the delta tables explicitly or not. We discussed in detail the benefits of computing the delta tables explicitly even when summary-deltas are computed.

One of the problems of using the summary-delta algorithm is that the installation phase can become the bottleneck. Reference [Qua97] developed a `modify` operator that can be used to efficiently install the changes of an aggregate view given its summary delta.

The `modify` operator can also be used to improve the efficiency of the installation of deletions. While the `modify` operator can be used for efficient installation, the `modify` operator has not been adopted or implemented in commercial RDBMSs. In this Chapter, we compared two methods (*i.e.*, SQL-delete and cursor-delete) that are supported by commercial RDBMSs and investigated when is it better to use one method over the other.

Reference [GMS93] assumes that the materialized views use the COUNT representation. On the other hand, [GL95] assumes that DUP representation is used. To our knowledge, this chapter is the first to investigate the pros and cons of both representations for view maintenance, and present supporting experiments.

We also briefly presented the WHIPS prototype in this chapter. Although there has been a significant amount of research devoted to view maintenance, there has only been a small number of system prototypes created for view maintenance. Reference [HZ96] presents the *Squirrel* integration mediator, which acts as a data warehouse. The data in Squirrel is stored in materialized views, just like in WHIPS. The main focus of Squirrel (as discussed in [HZ96]) was its support for *virtual attributes*. Reference [HZ96] does not discuss the very important low-level details of view maintenance that we discussed in this chapter.

Reference [CKL+97] likewise does not focus on the low-level details of view maintenance. Instead, it discusses how the *Sword* warehouse prototype supports different "maintenance policies." The policies refer to when a view needs to be updated with respect to the time the underlying data (of the view) changes. For instance, one policy, is to *immediately* update a view when the underlying data changes. Another policy is to update the view in a *deferred* fashion, *e.g.*, when the view is needed in answering a query.

Reference [Rou91] describes the ADMS prototype, which investigates the advantages of materializing a view using view-caches (*i.e.*, join indices) as opposed to view tuples. Although this study is important, commercial RDBMSs do not allow materialized views (*i.e.*, tables) to be represented using join indices. Hence, in practice, data warehouses are stuck materializing a view using view tuples. Still, both the DUP and the COUNT representation are possible when views are materialized using view tuples. Again, we investigated the pros and cons of both representations.

7.6 Chapter Summary

In this chapter, we discussed the critical design decisions that were made in developing the WHIPS prototype for efficient data warehousing. We showed through experiments why the design decisions made were appropriate. These decisions provide guidelines for anyone developing data warehouse management software that runs on top of a back-end RDBMS.

Chapter 8

Conclusions and Future Work

In this thesis we developed algorithms for improving the efficiency of data warehousing systems, including streamlining the warehouse update, lowering the warehouse storage cost, and recovering failed warehouse loads. In Chapter 2, we developed snapshot differential algorithms for efficiently detecting source changes. We reduced the snapshot differential problem to performing an outerjoin between the old and the new snapshots. We then augmented the outerjoin algorithms with compression. We also developed the *Window* algorithm that only performs a single-pass over the snapshots, yielding significant performance improvements.

In Chapter 3, we investigated how to most efficiently compute and install the changes of the warehouse views. We developed the *MinWork* algorithm that finds efficient VDAG strategies (for updating the views) under a linear cost model. We then showed experimentally that the VDAG strategies picked perform well on a commercial RDBMS.

We presented algorithms in Chapter 4 for choosing additional views and indices to materialize so that the warehouse can be more efficiently updated. We developed an algorithm based on A* search that picks the optimal combination of indices and views. Even though the A*-based algorithm prunes many of the choices, because of the enormity of the search space, heuristic algorithms are necessary. Heuristic algorithms as well as rules of thumb for picking views and indices were also developed in Chapter 4.

In Chapter 5, we presented techniques for reducing the storage cost of the warehouse. First, we developed a constraint language that can be used to describe the base views (as well as derived views). We showed that the language can describe many types of constraints. Using the constraints, we developed an algorithm that identifies base view tuples that will never be used in computing the changes of the derived views. Under the assumption that most analytical queries can be answered using derived views, these base view tuples can be archived resulting in a significant reduction of storage cost.

In Chapter 6, we developed the algorithm *DR* that can be used to resume failed load workflows (*i.e.*, cleaning processes). *DR* does not have any normal operation overhead and yet we showed experimentally that it can significantly reduce the recovery cost. Furthermore, *DR* does not require that the low-level details of the load workflow be known.

In Chapter 7, we presented the WHIPS prototype, and the decisions that were made regarding its implementation. We presented experiments that supported our design decisions.

We now describe several areas of future work. The first five areas, discussed in Sections 8.1 to 8.5, are significant extensions to the techniques discussed in the earlier chapters of the thesis. The last two areas, discussed in Sections 8.6 to 8.7, are research problems that were not touched upon in the thesis.

8.1 Parallel VDAG Maintenance

In Chapter 3, we discussed VDAG strategies for updating all of the views in the VDAG. We modeled a VDAG strategy, denoted $\overrightarrow{\mathcal{E}}$, as a *sequence of expressions*. Each expression was either a compute or an install expression, and the expressions are sent one at a time to the underlying database. An alternative model of a VDAG strategy is a *sequence of expression sets* denoted $\overrightarrow{\mathcal{S}}$, wherein each set can be handled by the database in parallel.

One of the techniques for solving a problem involving parallel processing is to "parallelize" a solution of the sequential problem. Hence, one approach is to parallelize the *MinWork* VDAG strategy $\overrightarrow{\mathcal{E}}$ to produce $\overrightarrow{\mathcal{S}}$. However, parallelizing the *MinWork* VDAG strategy may not be the best approach since the *MinWork* VDAG strategy only uses 1-way view strategies which require certain compute and install expressions to be performed before other expressions (see Chapter 3). Because of these numerous dependencies, many of the expressions in the *MinWork* VDAG strategy cannot be processed in parallel.

We have identified two techniques that allow more expressions to be processed in parallel.

1. The first technique is to use dual-stage view strategies (*i.e.*, view strategies that propagate the underlying changes simultaneously) instead of 1-way view strategies. At one extreme, if all the derived views use dual-stage view strategies, the only dependency

left between the expressions is that the expressions of V_j's view strategy must succeed the expressions of V_i's view strategy if V_j is defined over V_i.

2. If we only use dual-stage view strategies, we can remove any remaining dependencies among the expressions by "flattening" the VDAG. For instance, let us consider the VDAG in Figure 3.11. When updating V_5, it may be possible to treat V_5 as if it were defined on V_1, V_2 and V_3. If so, then the compute expressions of V_5 and V_4 can run in parallel. The expressions of V_5's dual-stage view strategy must succeed those of V_4's dual-stage view strategy. However, for the purpose of updating V_5, it is possible to treat V_5 as if it was defined on V_1, V_2 and V_3. Thus, the expressions of V_5's dual-stage view strategy do not access V_4 or δV_4 (*i.e.*, the changes to V_4) anymore. As a result, there are no dependencies between the expressions of V_5's view strategy and V_4's view strategy.

Unfortunately, using these techniques increases the total work incurred by the VDAG strategy. As a result, any benefit that arises from allowing more expressions to run in parallel may be offset by an increase in total work. An interesting direction of future work is to devise an algorithm that intelligently decides the extent to which these techniques should be applied.

8.2 VDAG Design

In Chapter 4, we developed algorithms to solve the view-index selection (VIS) problem. Recall that for the VIS problem, we are given a single derived view V from the VDAG, and we are tasked to find a set of supporting views and indices so that the maintenance cost of V and its supporting views and indices is minimized. Clearly, it is important to solve the general problem wherein we are given a VDAG, and we are tasked to find a set of supporting views and indices so that either the maintenance cost is minimized, or the response time to queries is minimized under certain constraints.

In [GM98], algorithms were proposed for selecting supporting views so that the response time to queries is minimized. Furthermore, after materializing the selected supporting views, the maintenance cost of the warehouse does not exceed a given threshold. However, [GM98] did not consider materializing indices. That is, they solved the view-selection problem and not the view-index-selection problem. It is well known that indices can be very useful in answering queries. Hence, another direction of future work to solve the VIS problem for a whole VDAG.

8.3 Cost-based Load Workflow Recovery

In Chapter 6, we developed the resumption algorithm *DR* for recovering failed load workflows. By deriving properties of the load workflow, *DR* was able to determine where it can discard input tuples to the various transforms of the workflows. Recall that we called the transforms for discarding input tuples filters. Since not all of the filters were beneficial, *DR* only assigned some of them. For instance, *DR* removed "redundant" filters that do not discard any tuples because of filters assigned earlier in the load workflow. Also, *DR* attempted to push the filters as close to the remote sources as possible. This way, the number of transforms that do not need to process the discarded tuples is maximized.

Although we showed that *DR* can reduce the recovery cost, we believe that a version of *DR* that decides in a cost-based fashion where to assign filters would reduce recovery cost even more. The first step in developing a cost-based *DR* is to develop a cost model for the processing performed by the various transforms. Ideally, the cost model should express the cost of processing n input tuples for each transform. Also, a cost model for the overhead of using the filters is required as well. Ideally, this cost model should express the cost of processing n input tuples for each filter. Given these cost models, we can then decide where it is beneficial to assign a filter in the load workflow, possibly using a greedy algorithm that assigns the filter with the most benefit first.

We also developed *DR-Log* in Chapter 6 which augments *DR* with logging. However, we did not develop a cost-based log placement algorithm. Developing a log placement algorithm that takes into account the amount of space available for logging tuples and the amount of normal operation overhead allowed, while minimizing the time to complete a resumed load, is another direction of future work.

8.4 Recovery of View Maintenance

In Chapter 6, we focused on developing recovery algorithms for the load workflow. We did not develop algorithms for recovering VDAG strategies (for view maintenance) because the recovery algorithm of the warehouse database can be used. However, we now show that a specialized recovery algorithm for VDAG strategies is useful. This specialized recovery algorithm can lower the storage cost of the data warehouse since it does not require using traditional "redo" or "undo" logging. In the example, we also illustrate the alternative where the recovery algorithm of the warehouse database is used.

EXAMPLE 8.4.1 Let us suppose we have a base view V_1, and two derived views V_2 and V_3 defined as follows.

- $Def(V_2) : \sigma_{P_2}(V_1)$
- $Def(V_3) : \sigma_{P_3}(V_2)$

Let us assume that the insertions to V_1 are reported and, thus, the derived views need to be updated. The following 5-step VDAG strategy can be used to update the warehouse.

1. $Comp(V_2, \{V_1\}) : \delta V_2 \leftarrow \sigma_{P_2}(\delta V_1)$
2. $Comp(V_3, \{V_2\}) : \delta V_3 \leftarrow \sigma_{P_3}(\delta V_2)$
3. $Inst(V_1)$ (see Chapter 3 for details of the $Inst$ expression)
4. $Inst(V_2)$
5. $Inst(V_3)$

Let us suppose that the VDAG strategy fails while in Step 2. If the entire VDAG strategy is one transaction, then the actions of the first two steps are undone, and the VDAG strategy transaction can be restarted. On the other hand, if nested transactions ([GR93]) are used, and each step is a sub-transaction, then Step 1 will not be redone and the VDAG strategy resumes at Step 2. More specifically, the database recovery algorithm "undoes" Step 2 (*i.e.*, the second sub-transaction) using undo logs, and then restarts the VDAG strategy at Step 2. However, because delta tables are used in the VDAG strategy, using undo and redo logs are not necessary. That is, assuming that we detect that the VDAG strategy failed at Step 2, we can issue an SQL `delete` statement to discard the contents of δV_3. Assuming the contents of δV_2 are in stable storage, Step 2 can then be redone. Intuitively, there is no need to use undo and redo logs because the same data saved in the logs is also saved in the delta tables. □

Although the specialized technique illustrated seems simple, there are many important details that need to be resolved.

- Techniques for detecting where the VDAG strategy failed are required. These techniques must not incur too many IOs for writing recovery information. Otherwise, the normal-operation overhead may become excessive.
- Efficient techniques for "undoing" a step are required. We showed that a simple SQL `delete` may suffice in some cases. However, undoing a step may not be as simple if there are multiple *Comp* expressions used for populating a delta table (as in Chapter 3). Also, the SQL `delete` only works for undoing *Comp* expressions. *Inst* expressions also need to be undone.
- The disadvantage of not using a log is that the overhead during normal operation may actually increase! In the example, it was required that δV_2 be in stable storage so that Step 2 can be redone. Unfortunately, this requires that δV_2 be flushed to disk to finish Step 1. Flushing too often can increase the normal operation overhead excessively. Techniques that are similar to "lazy checkpointing" [GR93] are required. Also, it may be possible to declare delta tables as append-only SQL tables to minimize random IOs.

In summary, there are some important details that need to be worked out to develop a specialized recovery algorithm for VDAG strategies. After developing such an algorithm, it would then be important to investigate the advantages of using the specialized recovery algorithm as opposed to using the recovery mechanism of the warehouse database (*e.g.*, in terms of storage cost, and normal-operation overhead).

8.5 Reducing the Deployment Time

As discussed in Chapter 6, the load workflow for warehouse creation is different from the one for warehouse update. Each load workflow takes weeks to design. On the other hand, once the VDAG of a warehouse is designed, the VDAG strategy necessary to propagate the changes up the VDAG can be derived from the definitions of the views mechanically. Hence, much of the time in deploying a data warehouse is spent in the design of the load workflow

One way to reduce the deployment time of a data warehouse is to avoid redesigning a new load workflow for the warehouse update. That is, given the load workflow for warehouse creation, it may be possible to construct a load workflow for the warehouse update automatically. We now illustrate how one may use *DR* for this purpose.

EXAMPLE 8.5.1 Let us suppose that view V is a warehouse (base) view, and it is derived from source relation R and possibly from other source relations as well. Let us suppose that the insertions to source relation R are detected. Let us denote the insertions as $\triangle R$. V can be updated as follows.

1. Given $\triangle R$, we can use *DR* to derive which of the V tuples are "affected" by $\triangle R$. Recall that *DR* derives the identifying attributes A of R. (More precisely, *DR* derives the identifying attributes of the edge in the load workflow emanating from R's extractor.) Given a V tuple t_V, the R tuples that match t_V on A are contributors to t_V. Similarly, an R tuple t_R contributes to all of the V tuples that match t_R on A. These are the V tuples that are affected by t_R. The new R tuples in $\triangle R$ may affect some tuples already in V. They can be found by matching tuples based on the identifying attribute as well. That is, the affected V tuples can be found using $V \ltimes_A \triangle R$.

2. The affected V tuples are removed from the warehouse.

3. Recall that *DR* takes as input a load workflow and outputs a similar load workflow but with filters. Let us call this load workflow with filters the "filter load workflow." We then use the remaining V tuples to instantiate the filter load workflow derived.

4. R is updated by inserting the $\triangle R$ tuples if they have not been inserted already. The extractors of the filter load workflow are restarted. The tuples produced by the load workflow (with filters) are appended to V.

□

The technique described above works when there are deletions and updates to R as well. (The technique is similar to the *DRed* algorithm proposed in [GMS93], where they delete affected tuples, and then rederive new attribute values for the affected tuples.) However, the technique may not work well for large load workflows, since it may often be the case that the identifying attributes of R are empty. If so, the technique would remove every tuple from the view, and essentially recompute the view from scratch. Hence, an interesting future work is to develop techniques that are complementary to the one described above, and to compare the performance of the various techniques.

8.6 Approximate Query Answering

This thesis did not discuss techniques for improving the processing of analytical queries. One way to more swiftly answer analytical queries is to give approximate answers rather than exact answers. Some important techniques for providing approximate answers have been proposed in [AGPR99, HHW97]. However, there is still much work that needs to be done.

In [AGPR99], the Aqua system precomputes statistical summaries, called *synopses*, on the warehouse views. The synopses take the form of various types of samples and histograms. Using the synopses, many analytical queries can be answered very swiftly but approximately. The Aqua system returns the error bounds of the approximate answer.

Clearly, an analyst may be unsatisfied with the approximate answer returned because he is not comfortable with the error bounds. An improved system may allow the user to specify error bounds requirements, and return an approximate answer that is within the specified bounds. To support such a system, the following features are required.

- Different sets of synopses may need to be maintained. Using more accurate synopses may yield tighter error bounds, but may take longer to produce an approximate answer. Hence, the appropriate synopses must be chosen so that the queries can be answered as fast as possible while respecting the requested error bounds.

- For each synopses, error bounds estimates should derived before the query is answered. This way, the system can efficiently select the synopses to use.

8.7 Forecasting Warehouse Data

We mentioned in Chapter 1 that the warehouse may keep a historical record of the source data. An interesting direction of future work is to develop algorithms that perform analysis on the warehouse data and "forecast" the contents of the warehouse views in the future.

Forecasting the warehouse data is helpful in many applications. For instance, if the warehouse stores data from a large retail store, the forecasts can be used to, say, find branches of the retail store that may run out of supply for certain products in the future.

We believe that techniques in data mining can be useful in forecasting warehouse data. For instance, using classification algorithms, we might be able to categorize the tuples in the various warehouse views. After the tuples are classified, we may be able to derive patterns that are useful for forecasting. For example, we might be able to derive that tuples of category `X` are inserted 100 times a day on average. Using these patterns, the contents of the warehouse views can be forecasted. Clearly, the appropriate data mining techniques need to be identified and integrated.

Appendix A

Chapter 3 Proofs

Theorem A.0.1 *For any given view, the best 1-way view strategy is optimal over the space of all view strategies (Theorem 3.4.1).* $\square$

Proof: Let view W be defined over the set of views $\mathcal{V}$. We show that given any non-1-way view strategy for W we can find a 1-way view strategy that is at least as efficient as the non-1-way view strategy.

Consider a non-1-way view strategy for W:

$$\overrightarrow{\mathcal{E}} = \langle\ \overrightarrow{\mathcal{E}_{prec}}, Comp(W, \mathcal{Y}), \overrightarrow{\mathcal{E}_{inst}}, \overrightarrow{\mathcal{E}_{succ}}\ \rangle.$$

In $\overrightarrow{\mathcal{E}}$, $\mathcal{Y}$ is a subset of $\mathcal{V}$ and $|\mathcal{Y}| > 1$; $\overrightarrow{\mathcal{E}_{prec}}$ is a possibly empty sequence of expressions preceding $Comp(W, \mathcal{Y})$; $\overrightarrow{\mathcal{E}_{inst}}$ is the sequence of $Inst$ expressions immediately following $Comp(W, \mathcal{Y})$ that installs the changes of $\mathcal{Y}$; $\overrightarrow{\mathcal{E}_{succ}}$ is the sequence of expressions that completes the view strategy.

We define a mapping called "separator" that transforms $\overrightarrow{\mathcal{E}}$ (based on some $Y_1 \in \mathcal{Y}$) into:

$$\overrightarrow{\mathcal{E}'} = \langle\ \overrightarrow{\mathcal{E}_{prec}}, Comp(W, \{Y_1\}), Inst(Y_1), Comp(W, \mathcal{Y} - \{Y_1\}), \overrightarrow{\mathcal{E}'_{inst}}, \overrightarrow{\mathcal{E}_{succ}} \rangle.$$

In $\overrightarrow{\mathcal{E}'}$, $\overrightarrow{\mathcal{E}_{prec}}$ and $\overrightarrow{\mathcal{E}_{succ}}$ are the same as in $\overrightarrow{\mathcal{E}}$, while $\overrightarrow{\mathcal{E}'_{inst}}$ is almost identical to $\overrightarrow{\mathcal{E}_{inst}}$ except that it does not have $Inst(Y_1)$ in it. Intuitively, the transformation "separates" the propagation of the changes of Y_1 from $Comp(W, \mathcal{Y})$, as well as the installation of the changes of Y_1 from the sequence $\overrightarrow{\mathcal{E}_{inst}}$.

It can be verified easily that conditions **C1** through **C6** hold for $\overrightarrow{\mathcal{E}'}$ if they hold for $\overrightarrow{\mathcal{E}}$. This implies that $\overrightarrow{\mathcal{E}'}$ is correct if $\overrightarrow{\mathcal{E}}$ is correct. So, "separator" preserves the correctness of the view strategy.

We now show that each application of "separator" results in a view strategy that is at least as efficient as the original view strategy. All the expressions in $\overrightarrow{\mathcal{E}_{prec}}$ and $\overrightarrow{\mathcal{E}_{succ}}$ incur the same amount of work in both $\overrightarrow{\mathcal{E}}$ and $\overrightarrow{\mathcal{E}'}$, since the same expressions are used and the expressions are evaluated on the same database state. Also, the install expressions in $\overrightarrow{\mathcal{E}_{inst}}$ incur the same amount of work as $Inst(Y_1)$ and the install expressions in $\overrightarrow{\mathcal{E}'_{inst}}$ since the same changes are installed. Hence, we must show that the compute expressions $Comp(W, \{Y_1\})$ and $Comp(W, \mathcal{Y} - \{Y_1\})$ in $\overrightarrow{\mathcal{E}'}$ do not incur more work than the compute expression $Comp(W, \mathcal{Y})$ in $\overrightarrow{\mathcal{E}}$.

Without loss of generality, let $\mathcal{Y} = \{Y_1, \ldots, Y_m\}$ and $\mathcal{V} = \mathcal{X} \cup \mathcal{Y} \cup \mathcal{Z}$, where $\mathcal{X} = \{X_1, \ldots, X_l\}$ and $\mathcal{Z} = \{Z_1, \ldots, Z_n\}$. Furthermore, suppose the changes of the views in $\mathcal{X}$ are propagated and installed in $\overrightarrow{\mathcal{E}_{prec}}$, while the changes of the views in $\mathcal{Z}$ are propagated and installed in $\overrightarrow{\mathcal{E}_{succ}}$. Finally, let us denote the extension of a view V after its changes have been installed as V'.

Note that $Comp(W, \mathcal{Y})$ of $\overrightarrow{\mathcal{E}}$ has $2^m - 1$ terms, since $|\mathcal{Y}| = m$ (see Section 3.3.3). We denote each of these terms as A_v, where v is a bit vector composed of m bits, whose values depend on the views and delta relations accessed by the term. The ith bit in v is set if term A_v accesses δY_i instead of Y_i, and vice versa. For instance, if $\mathcal{Y} = \{Y_1, Y_2\}$, there will be three terms: term A_{10} combines δY_1 and Y_2, term A_{01} combines Y_1 and δY_2, and term A_{11} combines δY_1 and δY_2. Each of these three terms accesses $\{X'_1, \ldots, X'_l\}$ and $\{Z_1, \ldots, Z_n\}$ as well.

In $\overrightarrow{\mathcal{E}'}$, $Comp(W, \mathcal{Y} - \{Y_1\})$ has $(2^{m-1} - 1)$ terms. We denote each term of $Comp(W, \mathcal{Y} - \{Y_1\})$ as C_v, where v is an $(m-1)$ bit vector. The ith bit of v is set if term C_v accesses δY_{i+1} instead of Y_{i+1} and vice versa. We map each term of $Comp(W, \mathcal{Y} - \{Y_1\})$ to a pair of terms of $Comp(W, \mathcal{Y})$. In particular, term C_v is mapped to the pair of terms A_{0v} and A_{1v}. The work incurred by term C_v is

$$c \cdot (|X'_1| + \ldots + |X'_l| + |Y'_1| + K + |Z_1| + \ldots + |Z_n|),$$

where K is the sum of the sizes of the views and delta relations of $\mathcal{Y} - \{Y_1\}$ considered by C_v.

On the other hand, the work incurred by the pair of terms A_{1v} and A_{0v} is

$$c \cdot (2 \cdot |X_1'| + \ldots + 2 \cdot |X_l'| + (|Y_1| + |\delta Y_1|) + 2 \cdot K + 2 \cdot |Z_1| + \ldots + 2 \cdot |Z_n|).$$

Since $|Y_1| + |\delta Y_1|$ is at least as large as $|Y_1'|$, we infer that the term C_v does not incur more work than the pair of terms A_{0v} and A_{1v} that it is mapped to.

$Comp(W, \{Y_1\})$ has exactly one term, which accesses $\{X_1', \ldots, X_l'\}$, $\{\delta Y_1, Y_2, \ldots, Y_m\}$, and $\{Z_1, \ldots, Z_n\}$. We map it to term $A_{10..0}$ of $Comp(W, \mathcal{Y})$, which incurs the same amount of work (because it uses the same combination of views and delta relations).

So far, we have shown that each term of $Comp(W, \mathcal{Y} - \{Y_1\})$ and $Comp(W, \{Y_1\})$ can be mapped to $Comp(W, \mathcal{Y})$ terms that incur the same or a larger amount of work. One can see that no $Comp(W, \mathcal{Y})$ term participates in the mapping of two different terms of $Comp(W, \mathcal{Y} - \{Y_1\})$ and $Comp(W, \{Y_1\})$. Firstly, any two $Comp(W, \mathcal{Y} - \{Y_1\})$ terms, C_v and $C_{v'}$, are mapped to two disjoint sets of $Comp(W, \mathcal{Y})$ terms. That is, C_v maps to A_{0v} and A_{1v}, while $C_{v'}$ maps to $A_{0v'}$ and $A_{1v'}$. Since $v \neq v'$, it follows that A_{0v} is not $A_{0v'}$, and A_{1v} is not $A_{1v'}$. Secondly, no term C_v of $Comp(W, \mathcal{Y} - \{Y_1\})$ has a bit vector v with all zeroes. So, C_v is never mapped to the term $A_{10...0}$, which is the term used in mapping the only term of $Comp(W, \{Y_1\})$.

From the above argument, we deduce that $Comp(W, \mathcal{Y} - \{Y_1\})$ and $Comp(W, \{Y_1\})$ in $\overrightarrow{\mathcal{E}'}$ do not incur more work than $Comp(W, \mathcal{Y})$ in $\overrightarrow{\mathcal{E}}$. Hence, each application of "separator" leads to a view strategy that is at least as efficient as the one that is transformed. Starting from a non-1-way view strategy for W, successive applications of "separator" lead to a 1-way view strategy for W that is at least as efficient as the original view strategy. Thus, the best 1-way view strategy for W is optimal over all the view strategies for W. $\Box$

Theorem A.0.2 *Given a view V defined over the views $\mathcal{V}$, let the view ordering $\overrightarrow{\mathcal{V}}$ arrange the views in increasing $|V_i'| - |V_i|$ values, for each $V_i \in \mathcal{V}$. Then, a 1-way view strategy for V that is consistent with $\mathcal{V}$ will incur the least amount of work among all the 1-way view strategies for V (Theorem 3.4.2).* $\Box$

Proof: Consider a view W defined over views $\mathcal{V} = \{V_1, \ldots, V_n\}$. Let $\overrightarrow{\mathcal{E}}$ be a view strategy for W that incurs the least amount of work. We show that $\overrightarrow{\mathcal{E}}$ must be consistent with some view ordering that orders the views based on increasing $|V_j'| - |V_j|$ values. The proof is by contradiction. That is, we assume that $\overrightarrow{\mathcal{E}}$ is not consistent with any such view ordering and show that this contradicts the fact that $\overrightarrow{\mathcal{E}}$ incurs the least amount of work.

If $\overrightarrow{\mathcal{E}}$ is a 1-way view strategy that is not consistent with a view ordering based on increasing $|V_j'| - |V_j|$ values, it must be of the form:

$$\langle \overrightarrow{\mathcal{E}_{prec}}, Comp(W, \{Y_j\}), Inst(Y_j), Comp(W, \{Y_i\}), Inst(Y_i), \overrightarrow{\mathcal{E}_{succ}} \rangle,$$

where $|Y_i'| - |Y_i| < |Y_j'| - |Y_j|$, for some Y_i and Y_j in $\mathcal{V}$.

We show that a different 1-way view strategy $\overrightarrow{\mathcal{E}'}$:

$$\langle \overrightarrow{\mathcal{E}_{prec}}, Comp(W, \{Y_i\}), Inst(Y_i), Comp(W, \{Y_j\}), Inst(Y_j), \overrightarrow{\mathcal{E}_{succ}} \rangle$$

incurs less work than $\overrightarrow{\mathcal{E}}$, thus contradicting the assumption that $\overrightarrow{\mathcal{E}}$ incurs the least amount of work.

All the expressions in $\overrightarrow{\mathcal{E}_{prec}}$ and $\overrightarrow{\mathcal{E}_{succ}}$ incur the same amount of work in both view strategies $\overrightarrow{\mathcal{E}}$ and $\overrightarrow{\mathcal{E}'}$, since the same expressions are used and the expressions are evaluated on the same database state. Also, the install expressions $Inst(Y_i)$ and $Inst(Y_j)$ incur the same amount of work in both strategies since the same changes are installed. Hence, we must show that the two compute expressions $Comp(W, \{Y_i\})$ and $Comp(W, \{Y_j\})$ incur less work in $\overrightarrow{\mathcal{E}'}$ than in $\overrightarrow{\mathcal{E}}$.

Without loss of generality, let $\mathcal{V} = \{X_1, \ldots, X_l\} \cup \{Y_i, Y_j\} \cup \{Z_1, \ldots Z_m\} = \mathcal{X} \cup \{Y_i, Y_j\} \cup \mathcal{Z}$, such that the changes of the views in $\mathcal{X}$ are propagated and installed in $\overrightarrow{\mathcal{E}_{prec}}$, while the changes of the views in $\mathcal{Z}$ are propagated and installed in $\overrightarrow{\mathcal{E}_{succ}}$.

The work incurred by $Comp(W, \{Y_j\})$ and $Comp(W, \{Y_i\})$ in view strategy $\overrightarrow{\mathcal{E}}$ is

$$c \cdot (\sum_{k=1..l} |X_k'| + |\delta Y_j| + |Y_i| + \sum_{k=1..m} |Z_k|) \;+\; c \cdot (\sum_{k=1..l} |X_k'| + |Y_j'| + |\delta Y_i| + \sum_{k=1..m} |Z_k|).$$

The work incurred by $Comp(W, \{Y_i\})$ and $Comp(W, \{Y_j\})$ in view strategy $\overrightarrow{\mathcal{E}'}$ is

$$c \cdot (\sum_{k=1..l} |X_k'| + |\delta Y_i| + |Y_j| + \sum_{k=1..m} |Z_k|) \;+\; c \cdot (\sum_{k=1..l} |X_k'| + |Y_i'| + |\delta Y_j| + \sum_{k=1..m} |Z_k|).$$

Note that the only difference between the above two work estimates is that the former uses $|Y_i|$ and $|Y_j'|$ while the latter uses $|Y_j|$ and $|Y_i'|$. Since $|Y_i'| - |Y_i| < |Y_j'| - |Y_j|$, we deduce that the two $Comp$ expressions incur less work in $\overrightarrow{\mathcal{E}'}$. Hence, the total work incurred by $\overrightarrow{\mathcal{E}'}$ is less than that of $\overrightarrow{\mathcal{E}}$. This contradicts our supposition that $\overrightarrow{\mathcal{E}}$ is a view strategy for W with the least amount of work.

Thus, the best 1-way view strategy is the one that is consistent with a view ordering that arranges views in increasing order of size changes. $\square$

Theorem A.0.3 *Given a view defined over n other views in the warehouse, MinWorkSingle finds an optimal view strategy for the view in $O(n\ log\ n)$ time (Theorem 3.4.3).* □

Proof: *MinWorkSingle* produces a 1-way view strategy because it only uses $Comp(W, \mathcal{V})$ expressions, where $|\mathcal{V}| = 1$. This view strategy is correct because it satisfies **C1**, **C2**, **C3**, **C4**, **C5**, and **C6** (see Section 3.4 for correctness discussion). Since the *Inst* expressions of this view strategy are ordered based on increasing $|V'| - |V|$ values, it is consistent with a view ordering that orders the views based on increasing $|V'| - |V|$ values. By Theorem 3.4.1 and Theorem 3.4.2, it follows that the view strategy produced by *MinWorkSingle* is optimal.

In producing the optimal view strategy, *MinWorkSingle* uses a step that sorts the views in $O(n\ log\ n)$ time, where n is the number of views. All other steps require $O(n)$ time or less. Hence, *MinWorkSingle* runs in $O(n\ log\ n)$ time. □

Theorem A.0.4 *Given a VDAG G, a VDAG strategy for G that uses optimal view strategies for all the views of G is optimal over all VDAG strategies for G (Theorem 3.5.1).* □

Proof: We start by observing that all VDAG strategies for G incur the same amount of work for their *Inst* expressions as they have the same set of changes to install. Two different VDAG strategies may differ in their amounts of total work by incurring differing amounts of work for their *Comp* expressions.

Let $\overrightarrow{\mathcal{E}}$ be a VDAG strategy for G. Consider the partitioning of the set of *Comp* expressions in $\overrightarrow{\mathcal{E}}$ based on the derived views whose updates the *Comp* expressions are computing. We have as many partitions (of $\overrightarrow{\mathcal{E}}$) as there are derived views in G. In fact, each partition of *Comp* expressions of $\overrightarrow{\mathcal{E}}$ corresponds to the set of *Comp* expressions in the view strategy used by $\overrightarrow{\mathcal{E}}$ for the derived view under consideration. Moreover, the amount of work incurred by each partition of the *Comp* expressions of $\overrightarrow{\mathcal{E}}$ is the same as the amount of work incurred by these *Comp* expressions in the view strategy. This is because these *Comp* expressions are executed in the VDAG strategy and in the view strategy with the same database state due to the fact that the view strategy is a subsequence of the VDAG strategy. Thus, the amount of work incurred by the *Comp* expressions of the VDAG strategy $\overrightarrow{\mathcal{E}}$ is the sum of the amounts of work incurred by the *Comp* expressions of the view strategies of the derived views of G. Note that the view strategies for the base views have no *Comp* expressions.

Let $\overrightarrow{\mathcal{E}_o}$ be a VDAG strategy for G that uses optimal view strategies for all the views of G, and let $\overrightarrow{\mathcal{E}_x}$ be another VDAG strategy for G. That is, the amount of work incurred by the set of *Comp* expressions in a view strategy used by $\overrightarrow{\mathcal{E}_o}$ is at most equal to the amount of work incurred by the set of *Comp* expressions in the corresponding view strategy used by $\overrightarrow{\mathcal{E}_x}$. Now, it follows from the earlier argument that the amount of work incurred by the *Comp* expressions in $\overrightarrow{\mathcal{E}_o}$ is at most equal to that incurred by the *Comp* expressions of $\overrightarrow{\mathcal{E}_x}$. Since $\overrightarrow{\mathcal{E}_o}$ and $\overrightarrow{\mathcal{E}_x}$ incur the same amount of work for their *Inst* expressions, we conclude that the total work incurred by $\overrightarrow{\mathcal{E}_o}$ is at most as much as that incurred by $\overrightarrow{\mathcal{E}_x}$.

Thus, we see that a VDAG strategy for G that uses optimal view strategies incurs the least amount of total work. □

Theorem A.0.5 *For any VDAG G, a 1-way VDAG strategy for G that is consistent with a desired view ordering is an optimal VDAG strategy for G. (Theorem 3.5.2).* □

Proof: We now prove that a 1-way VDAG strategy for G that is consistent with a desired view ordering uses optimal view strategies to update all the views of G. Based on Theorem A.0.4, this VDAG strategy is optimal for G.

Let us consider a derived view V_i defined over views $\mathcal{V}_i$. Based on Theorem A.0.2, an optimal view strategy $\overrightarrow{\mathcal{E}_i}$ for V_i is the 1-way view strategy consistent with the view ordering $\overrightarrow{\mathcal{V}_i}$ that orders all the views in $\mathcal{V}_i$ in increasing $|V'| - |V|$ values.

On the other hand, a 1-way VDAG strategy consistent with the desired view ordering updates V_i using a 1-way view strategy $\overrightarrow{\mathcal{E}_i'}$ consistent with the desired view ordering $\overrightarrow{\mathcal{V}}$ the orders *all* of the VDAG views in increasing $|V'| - |V|$ values.

Since both $\overrightarrow{\mathcal{E}_i}$ and $\overrightarrow{\mathcal{E}_i'}$ are 1-way view strategies for V_i, they use the same *Comp* and *Inst* expressions. Furthermore, we now show that the order of the *Comp* and *Inst* expressions are the same. Let us suppose $Comp(V_i, \{V_j\}) < Comp(V_i, \{V_k\})$ in $\overrightarrow{\mathcal{E}_i}$, but $Comp(V_i, \{V_k\}) < Comp(V_i, \{V_j\})$ in $\overrightarrow{\mathcal{E}_i'}$. This implies that $V_j < V_k$ in $\overrightarrow{\mathcal{V}_i}$ but $V_k < V_j$ in $\overrightarrow{\mathcal{V}}$. This is not possible since both view orderings are based on increasing $|V'| - |V|$ values. Since the *Comp* expressions are in the same order in both view strategies, all the expressions including the *Inst* expressions, must be in the same order based on **C3** and **C4**. Hence, $\overrightarrow{\mathcal{E}_i'}$ incurs the same amount of work as $\overrightarrow{\mathcal{E}_i}$ which is an optimal view strategy.

Since this argument holds for any derived view V_i, we have proven that a 1-way VDAG strategy for G that is consistent with a desired view ordering uses optimal view strategies to update all the views of G. □

Theorem A.0.6 *Given a VDAG G, if $EG(G, \overrightarrow{\mathcal{V}})$ is acyclic where $\overrightarrow{\mathcal{V}}$ is a desired view ordering, a topological sort of $EG(G, \overrightarrow{\mathcal{V}})$ yields an optimal VDAG strategy for G (Theorem 3.5.3).* □

Proof: We prove the theorem by first presenting and proving the following lemma.

Lemma A.0.1 *Given an acyclic $EG(G, \overrightarrow{\mathcal{V}})$ for a given VDAG G and a view ordering $\overrightarrow{\mathcal{V}}$, a 1-way VDAG strategy consistent with $\overrightarrow{\mathcal{V}}$ is obtained by topologically sorting the expression graph.* □

To prove the lemma, we first show that the 1-way VDAG strategy satisfies all correctness conditions. **C1** and **C2** are satisfied because the expression graph includes a node for each expression used in a VDAG strategy. Furthermore, a topological sort of the expression graph includes all of the nodes in the graph. **C6** is also satisfied since there is only one node for each expression, and, hence, the topological sort does not duplicate any expression.

Condition **C3**, $\forall V_i \in G(\mathcal{V}) : Comp(V, \{...V_i...\} < Inst(V_i)$, holds because an edge $Inst(V_i) \rightarrow Comp(V,$ $\{...V_i...\})$ is in the expression graph for each derived view V_i. Hence, a topological sort of the expression graph puts $Inst(V_i)$ after $Comp(V, \{...V_i...\})$. Similarly, for **C4**, **C5** and **C8**, the expression graph has edges that ensure that the topological sort will order the expressions appropriately.

Since we just argued that the view strategy employed for each view satisfies conditions **C1** to **C6**, it follows that **C7** holds.

We now prove that the 1-way VDAG strategy $\overrightarrow{\mathcal{E}}$ produced is consistent with $\overrightarrow{\mathcal{V}}$. Let us suppose $\overrightarrow{\mathcal{E}}$ is not consistent with $\overrightarrow{\mathcal{V}}$. If so, there must be a view V_k such that $\overrightarrow{\mathcal{E}}$ employs view strategy $\overrightarrow{\mathcal{E}_k}$ to update V_k. Furthermore, $Comp(V_k, \{V_j\}) < Comp(V_k, \{V_i\})$ in $\overrightarrow{\mathcal{E}_k}$, while $V_i < V_j$ in $\overrightarrow{\mathcal{V}}$. However, the expression graph has an edge $Comp(V_k, \{V_j\}) \rightarrow Comp(V_k, \{V_i\})$ that ensures that the topological sort puts $Comp(V_k, \{V_i\})$ ahead of $Comp(V_k, \{V_j\})$. Since $\overrightarrow{\mathcal{E}_k}$ is a subsequence of $\overrightarrow{\mathcal{E}}$, it must be that $Comp(V_k, \{V_i\})$ is ahead of $Comp(V_k, \{V_j\})$ in $\overrightarrow{\mathcal{E}_k}$. This proves that $\overrightarrow{\mathcal{E}}$ is consistent with $\overrightarrow{\mathcal{V}}$.

Finally, since the expression graph only includes $Comp$ expressions of the form $Comp(V, \mathcal{V})$, where $|\mathcal{V}| = 1$, it must be that the VDAG strategy produced is a 1-way VDAG strategy that is consistent with $\overrightarrow{\mathcal{V}}$.

With Lemma A.0.1, we can now easily prove the theorem. Given a $\overrightarrow{\mathcal{V}}$-acyclic VDAG, a topological sort of the expression graph produces a 1-way VDAG strategy consistent with $\overrightarrow{\mathcal{V}}$

according to Lemma A.0.1. It follows from Theorem A.0.5 that the 1-way VDAG strategy produced is optimal. □

Lemma A.0.2 *For a tree VDAG, every view ordering results in an acyclic expression graph (Lemma 3.5.1).* □

Proof: We begin the proof by providing some notation that will also be used in subsequent proofs (for Lemma A.0.3 and Theorem A.0.8). We label the edges of an expression graph based on the "constraint" that requires the edge. For instance, an edge of the form $Comp(V_k, \{V_j\}) \to Comp(V_j, \{V_i\})$ is labeled $C8$ since condition **C8** requires it. Similarly, an edge of the form $Inst(V_i) \to Comp(V, \{V_i\})$ is labeled $C3$; an edge of the form $Inst(V) \to Comp(V, \{V_i\})$ is labeled $C5$; and an edge of the form $Comp(V, \{V_j\}) \to Comp(V, \{V_i\})$ is labeled $\overrightarrow{\mathcal{V}}$. Assuming there is an edge of the form $Comp(V, \{V_j\}) \to Comp(V, \{V_i\})$, there must be an edge $Comp(V, \{V_j\}) \to Inst(V_i)$ as required by **C4**. Edges of this form are labeled $C4$.

We denote paths based on these edge labels. For instance, path $Comp(V_k, \{V_j\}) \to Comp(V_j, \{V_i\}) \to Comp(V_i, \{V_h\})$ is a $C8C8$ path. $C8^+$ denote paths composed of at least one $C8$ edge followed by zero or more $C8$ edges. $C8^*$ denotes either an empty path or a $C8^+$ path. Path $Comp(V, \{V_j\}) \to Inst(V_i) \to Comp(V, \{V_i\})$ is a $C4C3$ path. $(C4C3)^+$ denote paths composed of at least a $C4$ edge followed by a $C3$ edge, and possibly followed by a series of $C4$ and $C3$ edges alternating. $(C4C3)^*$ denotes either an empty path or a $(C4C3)^+$ path.

We distinguish between two types of $C4C3$ paths. A path of the form $Comp(V, \{V_j\}) \to Inst(V_i) \to Comp(V, \{V_i\})$ is a *local* $C4C3$ path. This is because both $Comp(V, \{V_j\})$ and $Comp(V, \{V_i\})$ belong to the same view strategy. On the other hand, a path of the form $Comp(V, \{V_j\}) \to Inst(V_i) \to Comp(V', \{V_i\})$ is a *non-local* $C4C3$ path assuming $V \neq V'$.

We simplify the expression graph by omitting edges labeled $C5$ and $\overrightarrow{\mathcal{V}}$ because for any cycle that uses these edges, some other cycle can be constructed using only $C3$, $C4$ and $C8$ edges. More specifically, for any cycle that uses a $\overrightarrow{\mathcal{V}}$ edge $Comp(V, \{V_j\}) \to Comp(V, \{V_i\})$, a cycle can be constructed by replacing the $\overrightarrow{\mathcal{V}}$ edge with the $C4C3$ path $Comp(V, \{V_j\}) \to Inst(V_i) \to Comp(V, \{V_i\})$. This path exists because the edge $Inst(V_i) \to Comp(V, \{V_i\})$ is required by **C3**, and the edge $Comp(V, \{V_j\}) \to Inst(V_i)$ is required by **C4** due to the presence of the $\overrightarrow{\mathcal{V}}$ edge $Comp(V, \{V_j\}) \to Comp(V, \{V_i\})$.

For any cycle that uses the $C5$ edge $Inst(V_j) \rightarrow Comp(V_j, \{V_i\})$, a cycle can be constructed by replacing the edge with the $C3C8$ path $Inst(V_j) \rightarrow Comp(V, \{V_j\}) \rightarrow Comp(V_j, \{V_i\})$. The existence of this path is guaranteed because there will never be a cycle that uses the edge $Inst(V_j) \rightarrow Comp(V_j, \{V_i\})$ where there is no view V defined on V_j. This is because there must be an edge $Comp(V, \{...\}) \rightarrow Inst(V_j)$ that completes the cycle since there are no edges between $Inst$ expressions in the expression graph. Since the edge $Comp(V, \{...\}) \rightarrow Inst(V_j)$ must be a $C4$ edge, V must be defined on V_j. Since V is defined on V_j, there must be a $C3$ edge $Inst(V_j) \rightarrow Comp(V, \{V_j\})$. Finally, because of the existence of the $C5$ edge $Inst(V_j) \rightarrow Comp(V_j, \{V_i\})$ in the first place, we can deduce that there is an expression $Comp(V_j, \{V_i\})$, and therefore a $C8$ edge $Comp(V, \{V_j\}) \rightarrow Comp(V_j, \{V_i\})$.

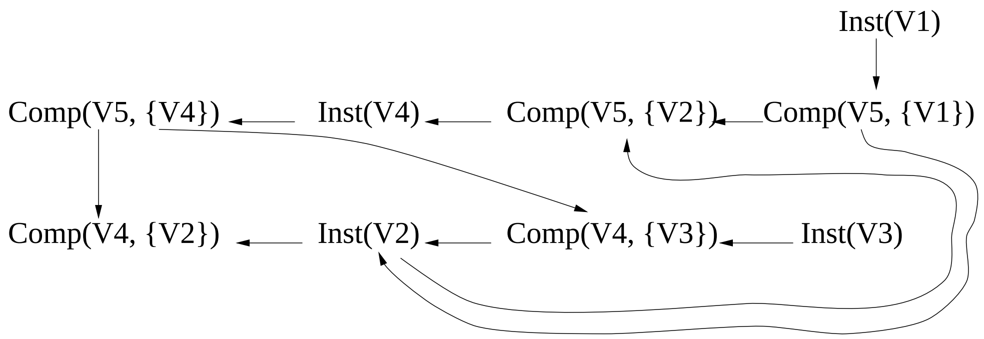

Figure A.1: Simplified Expression Graph

As an example, the simplified version of the expression graph for the VDAG shown in Figure 3.11 and the view ordering $\overrightarrow{\mathcal{V}} = \langle V_4, V_2, V_1, V_3, V_5 \rangle$ is shown in Figure A.1. Note that we have removed any expressions (*i.e.*, $Inst(V_5)$) that have no outgoing edges. Note also that there is only one cycle in the simplified expression graph which uses the path $C8C4C3C4C3$ (starting with the $C8$ edge $Comp(V_5, \{V_4\}) \rightarrow Comp(V_4, \{V_3\})$).

Using this simplified expression graph, we now derive a general form of cycles in the expression graph. First we make the following observations.

- There are no cycles using $C8$ edges only. This is because a $C8$ edge $Comp(V_k, \{V_j\}) \rightarrow Comp(V_j, \{V_i\})$ corresponds to the VDAG edges $V_k \rightarrow V_j$, and $V_j \rightarrow V_i$. Hence, any cycle using only $C8$ edges implies a cycle in the VDAG – a contradiction.
- Clearly there are no cycles using $C3$ edges only, nor any cycles using $C4$ edges only. This is because each $C3$ edge starts with an $Inst$ expression and ends with a $Comp$

expression. Therefore, a $C3C3$ path is not even possible. Similarly, a $C4C4$ path is not possible. By the same argument, there can be no cycles using $C8$ and $C3$ edges only, nor can there be any cycles using $C8$ and $C4$ edges only.

- There are no cycles using $C3$ and $C4$ edges only as explained below.

To explain the last observation, we introduce the function $Pos(E)$ applied to an expression in the graph. $Pos(Inst(V_i))$ returns the position of V_i in the view ordering $\overrightarrow{\mathcal{V}}$ that was used to construct the expression graph. $Pos(Comp(V, \{V_i\}))$ also returns the position of V_i in the view ordering $\overrightarrow{\mathcal{V}}$. Given any edge $A = E_j \rightarrow E_i$, the *starting position* of the edge is $Pos(E_j)$, and the *ending position* of the edge is $Pos(E_i)$.

Given any cycle $A_1A_2 \ldots A_n$, the starting and ending positions of the first edge A_1 and the last edge A_n must be the same, since edge A_1 must emanate from the expression that edge A_n is going.

If a cycle is composed of only $C4$ and $C3$ edges, it must be of the form $(C4C3)^+$. (Alternatively, the cycle could be denoted $(C3C4)^+$.) For a $C4$ edge, the starting position must be greater than the ending position. This is because a $C4$ edge $Comp(V, \{V_j\}) \rightarrow Inst(V_i)$ is required since $V_i < V_j$ in the view ordering which implies that $Pos(Comp(V, \{V_j\})) > Pos(Inst(V_i))$. On the other hand, for a $C3$ edge $Inst(V_i) \rightarrow Comp(V, \{V_i\})$, the starting and ending position is the same. Hence, for any path of the form $(C4C3)^+$, the starting position of the first edge must be greater than the ending position of the last edge. Hence, it is impossible to construct a cycle of the form $(C4C3)^+$.

Thus, cycles must be composed of $C3$, $C4$ and $C8$ edges. An example of such a cycle is the $C8C4C3C4C3$ cycle in Figure A.1 starting with the $C8$ edge $Comp(V_5, \{V_4\}) \rightarrow Comp(V_4, \{V_3\})$. In general, cycles are of the form

$$C8^+(C4C3)^+(C8^+(C4C3)^+)^*.$$

Since cycles must have $C8$ edges, we can assume without loss of generality that they start at some $C8$ edge. Since cycles must have some $C4$ and $C3$ edges, a $C4$ edge must follow the initial path of $C8$ edges. A $C3$ edge cannot follow since a $C3$ edge emanates from an $Inst$ expression. Since a $C4$ edge ends in an $Inst$ expression, a $C3$ edge must follow a $C4$ edge since a $C3$ edge is the only type of edge that emanates from an $Inst$ expression. The initial path $C8^+(C4C3)^+$ can be followed by zero or more paths of the same form. (Note that a cycle $C8^+(C4C3)^+C8^+$ can be denoted as a $C8^+(C4C3)^+$ cycle by changing the starting edge of the cycle.)

Crux of the proof: With this notation in hand, and with this general description of a cycle, we can now prove that for tree VDAGs, there are no cycles.

Given a cycle $C8^+(C4C3)^+(C8^+(C4C3)^+)^*$, there must be at least one non-local $C4C3$ path. Let us assume otherwise. This implies that all $C4C3$ paths in the cycle

$$C8^+(C4C3)^+(C8^+(C4C3)^+)^*$$

are local. However, given any path of the form $C8^+(C4C3)^+$, where all the $C4C3$ paths are local, the path can be shortened into a path of the form $C8^+$! For instance, a $C8C4C3C4C3$ path

$$Comp(V, \{V_k\}) \rightarrow Comp(V_k, \{V_j\}) \rightarrow Inst(V_i) \rightarrow$$
$$Comp(V_k, \{V_i\}) \rightarrow Inst(V_h) \rightarrow Comp(V_k, \{V_h\}),$$

can be shortened to $Comp(V, \{V_k\}) \rightarrow Comp(V_k, \{V_h\})$, since the existence of this edge is guaranteed to ensure that the condition **C8** is met. Thus, if there is a cycle

$$C8^+(C4C3)^+(C8^+(C4C3)^+)^*$$

that only uses local $C4C3$ paths, then there must be a cycle using $C8$ edges only which we have observed to be impossible.

The existence of a non-local $C4C3$ path implies that the VDAG is not a tree. To see this, a non-local $C4C3$ path is of the form $Comp(V, \{V_i\}) \rightarrow Inst(V_i) \rightarrow Comp(V', \{V_i\})$ where $V \neq V'$. This implies that there are at least two views defined on V_i. This further implies that there are at least two paths that end in V_i in the VDAG, which is not possible in a tree VDAG. □

Lemma A.0.3 *For a uniform VDAG, every view ordering results in an acyclic expression graph (Lemma 3.5.2).* □

Proof: In the proof of Lemma A.0.2, we showed that cycles are of the form

$$C8^+(C4C3)^+(C8^+(C4C3)^+)^*,$$

in general.

Similar to the *Pos* function defined in the proof of Lemma A.0.2, we first define the $Level(E)$ function applied to an expression E. $Level(Inst(V_i))$ returns $Level(V_i)$ of the view

V_i in the VDAG that was used to construct the expression graph. Similarly, $Level(Comp(V,$-$\{V_i\}))$ returns $Level(V_i)$. Given an edge $A = E_j \rightarrow E_i$, we say that *starting level* of A is $Level(E_j)$ and the *ending level* of A is $Level(E_i)$.

In any cycle $A_1A_2 \ldots A_n$, it is clear that the starting level of A_1 is the same as the ending level of A_n. This is because the expression from which A_1 emanates is the same as the expression that A_n is going to.

We now make the following observations. For any path of the form $C8^+$, the starting level of the first edge is greater than the ending level of the last edge. This is because for any $C8$ edge $Comp(V_k, \{V_j\}) \rightarrow Comp(V_j, \{V_i\})$, the starting level of the edge must be greater than the ending level because V_j is defined on V_i.

The next two observations only hold for expressions constructed from uniform VDAGs. For any $(C4C3)^+$ path composed of only local $C4C3$ paths, the starting level of the first edge is the same as the ending level of the last edge. This is because for a $C4C3$ path $Comp(V, \{V_j\}) \rightarrow Inst(V_i) \rightarrow Comp(V, \{V_i\})$ implies that V is defined on both V_j and V_i. Since for a uniform VDAG, V is defined only on views with the same *Level* value, it must be that $Level(Comp(V, \{V_j\})) = Level(Comp(V, \{V_i\}))$ because $Level(V_j) = Level(V_i)$.

For any $(C4C3)^+$ path composed of only non-local $C4C3$ paths, the starting level of the first edge is the same as the ending level of the last edge. This is because for a $C4C3$ path $Comp(V', \{V_j\}) \rightarrow Inst(V_i) \rightarrow Comp(V, \{V_i\})$ implies that V' is defined on both V_i and V_j. To see this, edge $Comp(V', \{V_j\}) \rightarrow Inst(V_i)$ implies the existence of the expression $Comp(V', \{V_i\})$, which in turn implies that V' is defined on V_i. Clearly, $Comp(V', \{V_j\})$ implies that V' is defined on V_j. For a uniform VDAG, it must be that $Level(V_i) = Level(V_j)$ because V' is defined on views with the same *Level* value. Hence, $Level(Comp(V', \{V_j\})) = Level(Comp(V, \{V_i\}))$.

Let us assume that there is a cycle of the form $C8^+(C4C3)^+(C8^+(C4C3)^+)^*$. However, for an expression graph constructed from a uniform VDAG, the first edge in the cycle

$$C8^+(C4C3)^+(C8^+(C4C3)^+)^*$$

must have a starting level that is greater than the ending level of the last edge of the cycle. Since for any cycle these two levels must be the same, we have arrived at a contradiction, proving the theorem. □

Theorem A.0.7 *Given a VDAG G, and a desired view ordering $\overrightarrow{\mathcal{V}}$, MinWork produces* *optimal VDAG strategies if $EG(G, \overrightarrow{\mathcal{V}})$ is acyclic. In particular, MinWork always produces optimal VDAG strategies for tree VDAGs and uniform VDAGs (Theorem 3.5.4).* □

Proof: Given a VDAG G and a desired view ordering $\overrightarrow{\mathcal{V}}$, such that $EG(G, \overrightarrow{\mathcal{V}})$, *MinWork* constructs the output VDAG strategy by constructing the expression graph and topologically sorting it. By Theorem A.0.6, the *MinWork* VDAG strategy is optimal. □

Theorem A.0.8 *Given a VDAG G and a view ordering $\overrightarrow{\mathcal{V}}$, we can come up with a view ordering $\overrightarrow{\mathcal{V}'}$ = ModifyOrdering(G, $\overrightarrow{\mathcal{V}}$) such that EG(G, $\overrightarrow{\mathcal{V}'}$) is acyclic. That is, MinWork will always succeed in producing a VDAG strategy (Theorem 3.5.5).* □

Proof: Let us assume that there is a cycle of the form $C8^+(C4C3)^+(C8^+(C4C3)^+)^*$. For any cycle, the starting level of the first edge is the same as the ending level of the last edge as shown in the proof of Lemma A.0.3.

We also observed in that proof that for any path of the form $C8^+$, the starting level of the first edge is greater than the ending level of the last edge.

We now make the following observations that hold for an expression graph constructed using $\overrightarrow{\mathcal{V}'} = ModifyOrdering(G, \overrightarrow{\mathcal{V}})$. For any $(C4C3)^+$ path composed of only local $C4C3$ paths, the starting level of the first edge is greater than or equal to the ending level of the last edge. This is because for a $C4C3$ path $Comp(V, \{V_j\}) \rightarrow Inst(V_i) \rightarrow Comp(V, \{V_i\})$ implies that $V_i < V_j$ in $\overrightarrow{\mathcal{V}'}$. Hence, it must that $Level(V_i) \leq Level(V_j)$, and therefore $Level(Comp(V, \{V_j\})) \geq Level(Comp(V, \{V_i\}))$.

Similarly, for any $(C4C3)^+$ path composed of only non-local $C4C3$ paths, the starting level of the first edge is greater than or equal to the ending level of the last edge. This is because for a $C4C3$ path $Comp(V', \{V_j\}) \rightarrow Inst(V_i) \rightarrow Comp(V, \{V_i\})$ implies that $V_i < V_j$ in $\overrightarrow{\mathcal{V}'}$. To see this, edge $Comp(V', \{V_j\}) \rightarrow Inst(V_i)$ implies the existence of the expression $Comp(V', \{V_i\})$, and the edge $Inst(V_i) \rightarrow Comp(V', \{V_i\})$. This implies that $V_i < V_j$ in the view ordering. Hence, $Level(V_j) \geq Level(V_i)$, and $Level(Comp(V', \{V_j\})) \geq Level(Comp(V, \{V_i\}))$.

From these observations, the starting level the first edge of any cycle

$$C8^+(C4C3)^+(C8^+(C4C3)^+)^*$$

must be greater than the ending level of the last edge of the cycle. We have arrived at a contradiction, proving the theorem. □

Theorem A.0.9 *Given a view ordering $\overrightarrow{V}$, all the correct 1-way VDAG strategies that are consistent with $\overrightarrow{V}$ incur the same amount of work (Theorem 3.6.1).* $\square$

Proof: Let the set of VDAG views be $\{V_1, \ldots, V_n\}$. Consider the view ordering $\langle V_1, \ldots, V_n \rangle$. Let $\overrightarrow{\mathcal{E}}$ and $\overrightarrow{\mathcal{E}'}$ be two different 1-way VDAG strategies that are consistent with this view ordering. We show that $\overrightarrow{\mathcal{E}}$ and $\overrightarrow{\mathcal{E}'}$ incur the same amount of work.

Note that $\overrightarrow{\mathcal{E}}$ and $\overrightarrow{\mathcal{E}'}$ must have the same set of *Comp* and *Inst* expressions since they are 1-way VDAG strategies for the same VDAG. They differ in the sequencing of these expressions. Each *Inst* expression incurs the same amount of work in $\overrightarrow{\mathcal{E}}$ and $\overrightarrow{\mathcal{E}'}$, because the work involved is independent of the position at which the *Inst* expression occurs in the VDAG strategy. We show below that each *Comp* expression also incurs the same amount of work in both the VDAG strategies. From this, we conclude that $\overrightarrow{\mathcal{E}}$ and $\overrightarrow{\mathcal{E}'}$ have the same amount of total work.

Since $\overrightarrow{\mathcal{E}}$ and $\overrightarrow{\mathcal{E}'}$ are 1-way VDAG strategies that are consistent with the view ordering $\langle V_1, \ldots, V_n \rangle$, they must be of the form:

$$\begin{aligned}
\overrightarrow{\mathcal{E}} &= \langle \overrightarrow{\mathcal{E}_1}, Inst(V_1), \overrightarrow{\mathcal{E}_2}, Inst(V_2), \ldots, \overrightarrow{\mathcal{E}_n}, Inst(V_n) \rangle \\
\overrightarrow{\mathcal{E}'} &= \langle \overrightarrow{\mathcal{E}'_1}, Inst(V_1), \overrightarrow{\mathcal{E}'_2}, Inst(V_2), \ldots, \overrightarrow{\mathcal{E}'_n}, Inst(V_n) \rangle
\end{aligned}$$

where $\overrightarrow{\mathcal{E}_i}$ and $\overrightarrow{\mathcal{E}'_i}$ are sequences of *Comp* expressions. Note that each of these sequences can contain multiple *Comp* expressions because a view can participate in the definition of multiple derived views.

Consider any *Comp* expression, say $Comp(W, \{Y\})$, that occurs in the two VDAG strategies. There are two cases to examine:

- **Case 1:** $Comp(W, \{Y\})$ is in $\overrightarrow{\mathcal{E}_j}$ and in $\overrightarrow{\mathcal{E}'_j}$.
- **Case 2:** $Comp(W, \{Y\})$ is in $\overrightarrow{\mathcal{E}_j}$ and in $\overrightarrow{\mathcal{E}'_k}$, where $k \neq j$.

Case 1: The same set of views have been installed when $Comp(W, \{Y\})$ is evaluated in both the VDAG strategies. This means that $Comp(W, \{Y\})$ will be evaluated with the same database state in both the VDAG strategies, and hence the work incurred by $Comp(W, \{Y\})$ will be the same in $\overrightarrow{\mathcal{E}}$ and $\overrightarrow{\mathcal{E}'}$.

Case 2: Without loss of generality, we assume that $k > j$. Consider the set of views, $\mathcal{I} = \{V_j, \ldots, V_{k-1}\}$, whose install expressions are after $Comp(W, \{Y\})$ in $\overrightarrow{\mathcal{E}}$ and before

$Comp(W, \{Y\})$ in $\overrightarrow{\mathcal{E}'}$. We show that $\mathcal{I}$ contains no view that participates in the definition of W.

First, we note that Y cannot be in $\mathcal{I}$. Otherwise, the view strategy for W used by $\overrightarrow{\mathcal{E}'}$ would be incorrect (because $Inst(Y)$ would precede $Comp(W, \{Y\})$, violating condition **C3**). In fact, Y must be in $\{V_k, \ldots, V_n\}$ in order for $\overrightarrow{\mathcal{E}'}$ to be correct. Now, if W were to be defined over a view V that is in $\mathcal{I}$, $Inst(V)$ would appear between $Comp(W, \{Y\})$ and $Inst(Y)$ in $\overrightarrow{\mathcal{E}}$ (a violation of condition **C4** with respect to the view strategy for W). This is not possible since $\overrightarrow{\mathcal{E}}$ is a correct VDAG strategy.

In general, the work incurred in evaluating a *Comp* expression is dependent on the database state in which the expression is evaluated. $Comp(W, \{Y\})$ is evaluated in $\overrightarrow{\mathcal{E}}$ after installing the set of views $\{V_1, \ldots, V_{j-1}\}$ while it is evaluated in $\overrightarrow{\mathcal{E}'}$ after installing the set of views $\{V_1, \ldots, V_{k-1}\}$. However, W is not defined over any view from $\{V_j, \ldots, V_{k-1}\}$ and so the work incurred in evaluating $Comp(W, \{Y\})$ will not be affected by the state of these views. Hence, $Comp(W, \{Y\})$ will incur the same amount of work in $\overrightarrow{\mathcal{E}}$ and $\overrightarrow{\mathcal{E}'}$. □

Theorem A.0.10 *Prune is guaranteed to produce the best 1-way DAG strategy for a given VDAG (Theorem 3.6.2).* □

Proof: Let us assume otherwise. Let $\overrightarrow{\mathcal{E}}$ be the VDAG strategy produced by *Prune*, and let $\overrightarrow{\mathcal{E}'}$ be a VDAG strategy such that $Work(\overrightarrow{\mathcal{E}'}) < Work(\overrightarrow{\mathcal{E}})$. By Lemma 3.6.1, $\overrightarrow{\mathcal{E}'}$ must be in some partition. However it cannot be in the partition $\overrightarrow{\mathcal{E}}$ is in because by Theorem 3.6.1, $\overrightarrow{\mathcal{E}}$ will incur the same amount of work as $\overrightarrow{\mathcal{E}'}$. Hence it must be in some partition where *Prune* picks $\overrightarrow{\mathcal{E}''}$. Since *Prune* picks $\overrightarrow{\mathcal{E}}$ finally, it must be that $Work(\overrightarrow{\mathcal{E}}) <= Work(\overrightarrow{\mathcal{E}''})$. According to Theorem 3.6.1, $Work(\overrightarrow{\mathcal{E}}) <= Work(\overrightarrow{\mathcal{E}''}) = Work(\overrightarrow{\mathcal{E}'})$ – a contradiction. This proves that *Prune* finds the best 1-way VDAG strategy. □

Appendix B

Chapter 4 Cost Model

In this section we give our formulas for deriving the overall cost of maintaining a set of views due to changes to the warehouse relations. The formulas are based upon cost models for queries and updates [ST85] appearing elsewhere. The formulas represent a fairly accurate and detailed cost model, upon which we based our implementation of an algorithm that used exhaustive search to find the optimal set of supporting views and indices for a given primary view. The results of experiments using this algorithm were used to justify our rules of thumb in Section 4.5 and our results in Section 4.7.

The main formula given in this section is $Cost_v(\mathcal{V})$, which is the cost of maintaining a set of views $\mathcal{V}$. The other formulas are used to support $Cost_v(\mathcal{V})$. We redefine the approximate formulas given in Section 4.5 for the cost of maintaining a view or an index in this section to use our more detailed cost model. Note that we do not give formulas for benefit, but one can derive $Benefit_v(V) = Cost_v(\mathcal{V}) - Cost_v(\mathcal{V} \cup \{V\})$.

Table 4.2 lists additional statistical functions that are used in the cost formulas in this section, In addition to the notation of Table 4.2, we need to define $H(V, R.A)$ as the height of an index on V for attribute $R.A$. Note that much of the statistical information for views can be derived from statistical information for the warehouse relations and the selectivities of selection and join conditions.

Table B.1 gives our formula for $Cost_v(\mathcal{V})$ and its supporting formulas. Note that $Eval(expr)$ is the traditional query optimization cost function. In the formulas we use $\triangle R$, $\triangledown R$, and μR to represent the set of insertions, deletions, and updates to R respectively. We have implemented an exhaustive-search query optimizer that calculates $Eval(expr)$ by considering all possible query plans. It uses as the cost estimates for each operator in the tree the formulas appearing in Table B.2. The optimizer evaluates the cost of each possible query plan and selects the plan with the minimum cost. In addition, the optimizer considers possibly using materialized views in the evaluation of the expression, and considers reusing results of other expressions (which have been saved in $\triangle V_R^{save}$ relations).

Two more formulas need to be explained:

$$yao(n,p,k) = \begin{cases} k & k < p/2 \\ (k+p)/3 & p/2 < k <= 2p \\ p & 2p < k \end{cases}$$

The *yao* function returns an estimate of the number of page read operations given that k out of n tuples are read from a relation spanning p pages. The yao function assumes that either the memory buffer is large enough to hold the entire relation, or that the tuple accesses have been sorted beforehand so that tuples from the same page will be requested one after the other. Since the assumption that a relation fits entirely in memory is unrealistic for a data warehouse and we assume that tuple accesses are not usually sorted beforehand, our formulas often make use of a function Y_{WAP} presented in [ML89] for estimating the number of page read operations given k tuple fetches and a memory buffer of m pages.

$$Y_{WAP}(n,p,k,m) \begin{cases} min(k,p) & p <= m \\ k & p > m \text{ and } k <= m \\ m + (k-m)(p-m)/p & p > m \text{ and } k > m \end{cases}$$

Name	Formula	Description
$Cost_v(\mathcal{V})$	$\sum_{V \in \mathcal{V}} Cost_v(V)$	Derive cost to maintain a set of views by summing cost to maintain each view.
$Cost_v(V)$	$\sum_{R \in \mathcal{R}(V)}(Prop_{ins}(R,V)$ $+ Prop_{del}(R,V)$ $+ Prop_{upd}(R,V))$	Sum the cost of propagate V changes to each relation into V
$Prop_{ins}(R,V)$	$Eval(\triangle R \bowtie R_2 \bowtie \ldots \bowtie R_k \rightarrow \triangle V_R)$, $+ Apply_{ins}(\triangle V_R, V)$ $+ Apply_{ins}(\triangle V_R, \triangle V_R^{save})$ $+ ApplyIx(\triangle V_R, V)$	Evaluate effect on V of $\triangle R$, which we call $\triangle V_R$, where $\{R, R_2, \ldots, R_k\} = \mathcal{R}(V)$ Insert $\triangle V_R$ into V Save it for possible reuse as $\triangle V_R^{save}$ (small cost anyway) Update indices on V
$Prop_{del}(R,V)$	$Eval(V \bowtie_{\text{key of R}} \triangledown R \rightarrow \triangledown V_R)$ $+ Apply_{delupd}(\triangledown V_R, V)$ $+ ApplyIx(\triangledown V_R, V)$	Evaluate effect on V of $\triangledown R$, which we call $\triangledown V_R$ Delete $\triangledown V_R$ from V Update indices on V
$Prop_{upd}(R,V)$	$Eval(V \bowtie_{\text{key of R}} \mu R \rightarrow \mu V_R)$ $+ Apply_{delupd}(\mu V_R, V)$	Evaluate effect on V of $\triangledown R$, which we call μV_R Update μV_R in V
$Apply_{ins}(R,V)$	$P(R)$	Append tuples in R to V
$Apply_{delupd}(R,V)$	$yao(T(V), P(V), T(R))$	Delete or update tuples of R in V ($R \subseteq V$). Exact locations of tuples of R in V are derived when R is derived. If index join is used to derive R instead of nested-block join, then use $Y_{WAP}(T(V), P(V), T(R), P_m)$ instead of $yao(T(V), P(V), T(R))$.
$ApplyIx(R,V)$	$\sum_{R.A \in \text{indices on } V}$ $(Y_{WAP}(T(V),$ $P(V, R.A),$ $T(R) * (H(V, R.A) - 1))$ $+ Y_{WAP}(T(V), P(V, R.A), T(R)))$	For each index on V, sum approximate number of index pages to read assuming root cached, plus approximate number of index pages to write (leaves only).

Table B.1: Cost Formulas

Operator	Formula	Description
$Eval$(Nested-block Join $E_1 \bowtie E_2$)	$Eval(E_1) + \lceil P(E_1)/P_m \rceil * Eval(E_2)$	Assume try to fit as much of left-hand expression result in memory as possible, then evaluate right-hand expression.
$Eval$(Index Join $E join V$)	$Eval(E)$	Cost of evaluating the left hand expression
	$+ Y_{WAP}(T(V), P(V, S.B), T(E) * X, P_m/2)$	Let $X = H(V, S.B) - 2 + \lceil P(V, S.B) * S(V, JC)/T(V) \rceil$. Let the join condition JC be on indexed attribute $S.B$ in V, then Y_{WAP} is the number of index pages to read, assuming buffer memory is split between index and relation
	$+ Y_{WAP}(T(V), P(V), T(E) * S(V, JC), P_m/2)$	Number of relation pages to read
$Eval$(Relation Scan V)	$P(V)$	
$Eval$(Index Scan V)	$H(V, S.B) - 1 + \lceil P(V, S.B) * \frac{S(V,SC)}{T(V)} \rceil$	Let the selection condition SC be on indexed attribute $S.B$ of V, then this line computes the number of index pages to read.
	$+ Y_{WAP}(T(V), P(V), S(V, SC), P_m)$	Number of relation page to read

Table B.2: Query-Optimizer Cost Formulas

Appendix C

Chapter 5 Proofs

Before we prove the lemmas, we define some notation. Given a maintenance subexpression $E = \pi_{\mathcal{A}} \sigma_{\mathcal{P}}(\times_{R \in \mathcal{R}} R)$, we use $\texttt{Res}(E)$ to denote $\sigma_{\mathcal{P}}(\times_{R \in \mathcal{R}} R)$. That is, $\texttt{Res}(E)$ projects all the attributes of all the views involved in the cross product. We use t_{res} to denote a tuple in $\texttt{Res}(E)$. Note that every t_{res} "manifests" itself in the result of E because we use bag semantics. Furthermore, since we require aggregate views to have the `COUNT` aggregate function, this observation holds for aggregate views. Similarly, we use t_{mapres} to denote a tuple in $\texttt{Res}(\texttt{Map}(E, T))$. Assuming $\mathcal{R}' \subseteq \mathcal{R}$, $t_{res}[\mathcal{R}']$ denotes the tuple resulting from t_{res} that includes only the attributes of the views in $\mathcal{R}'$. We also use $t_{res}[\mathcal{A}]$ to denote the tuple resulting from t_{res} that includes only the attributes in $\mathcal{A}$. Given a maintenance subexpression $E = \pi_{\mathcal{A}} \sigma_{\mathcal{P}}(\times_{R \in \mathcal{R}} R)$, we use $\mathcal{P}[t_{mapres}]$ to be the condition resulting from replacing each attribute with its value in t_{mapres}.

We now formalize the definition of when a tuple t is "needed" by E.

Definition C.0.1 (needed) Let $E = \pi_{\mathcal{A}} \sigma_{\mathcal{P}}(\times_{R \in \mathcal{R}} R)$, and $T \in \mathcal{R}$. Let $\mathcal{D}$ be the delta relations in $\mathcal{R}$. Tuple $t \in T$ is *needed* by E if and only if for some extension of the delta relations, $\exists t_{res} \in \texttt{Res}(E)$ such that $t_{res}[\{T\}] = t$ and for all $R \in \mathcal{R}$ that is not a delta relation, $t_{res}[\{R\}] \in R$. $\square$

Intuitively, the definition states that $t \in T$ is needed by $E = \pi_{\mathcal{A}} \sigma_{\mathcal{P}}(\times_{R \in \mathcal{R}} R)$ if there is a tuple t_{res} in $\texttt{Res}(E)$ that t can "contribute" to. If t is removed from T, then $t_{res}[\mathcal{A}]$ is also removed from the result of E. (Note that this also holds even when t has duplicates because removing t would decrease the number of duplicates of $t_{res}[\mathcal{A}]$.)

Proof of Lemma 5.3.1 For the proof, we denote $\texttt{Map}(E, T)$ as $\pi_{\texttt{Attrs}(T)} \sigma_{\mathcal{P}'}(\times_{R \in (\mathcal{R} - \mathcal{D})} R)$, where $\mathcal{P}'$ is obtained from $\mathcal{P}$ using `Closure` and `Ignore`. $\mathcal{D}$ are the delta relations in $\mathcal{R}$.

Proof: (Lemma 5.3.1)
(Necessity) Assume $t \in T$ is needed. By Definition C.0.1, there exist a tuple $t_{res} \in \texttt{Res}(E)$ so that $t_{res}[\{T\}] = t$. We can obtain $t_{mapres} \in \texttt{Res}(\texttt{Map}(E,T))$ as $t_{res}[\mathcal{R} - \mathcal{D}]$. This follows from the soundness of the `Closure` procedure ([Ull89a]) and the definition of the `Ignore` procedure which guarantee that $\mathcal{P} \Rightarrow \mathcal{P}'$. It follows that $t = t_{mapres}[\{T\}](= t_{res}[\{T\}])$ since the attribute values were not changed in obtaining t_{mapres} from t_{res}. Hence $t \in \texttt{Map}(E,T)$ and $t \in \texttt{Needed}(T, \mathcal{E})$ (since $E \in \mathcal{E}$).
(Sufficiency) Assume $t \in \texttt{Needed}(T, \mathcal{E})$. Hence, for some $E \in \mathcal{E}$, $t \in \texttt{Map}(E,T)$. This implies that $t_{mapres} \in \texttt{Res}(\texttt{Map}(E,T))$ with $t_{mapres}[\{T\}] = t$. Since $\texttt{Map}(E,T)$ is not empty (because of the presence of t), $\mathcal{P}'$ does not have the atomic condition `false` ,*i.e.*, $\mathcal{P}'$ is satisfiable. Since `Ignore` does not remove `false`, it must be that $\mathcal{P}$ is also satisfiable. Furthermore, since the `Closure` procedure is complete ([Ull89a]), we are guaranteed that $\mathcal{P}[t_{mapres}]$ is satisfiable.

To see this, there are five types of atomic conditions in $\mathcal{P}$. (R_i and R_j denote non-delta relations. D_k and D_l denote delta relations. K denotes a constant. θ is $=$, $>$ $\geq$, $\leq$, $<$, or $\neq$) They are: (1) $R_i.a\ \theta\ R_j.b$; (2) $R_i.a\ \theta\ K$; (3) $R_i.a\ \theta\ D_k.b$; (4) $D_k.a\ \theta\ D_l.b$; (5) $D_k.a\ \theta\ K$. Recall that $\mathcal{P}$ is satisfiable. Since $t_{mapres} \in \texttt{Res}(\texttt{Map}(E,T))$ it follows that $\mathcal{P}'[t_{mapres}] =$ `true`. Since $\mathcal{P}'$ is obtained using `Closure` on $\mathcal{P}$ (then `Ignore`), $\mathcal{P}' \Rightarrow \texttt{Ignore}(\mathcal{P}, \mathcal{D})$ and $\texttt{Ignore}(\mathcal{P}, \mathcal{D})[t_{mapres}] =$ `true`. Hence, $\mathcal{P}[t_{mapres}]$ is a conjunction of `true` and atomic conditions of type (3), (4) and (5). Type (3) atomic conditions become type (5) since the attribute references are replaced by the values in t_{mapres}. Since `Closure` is complete, any type (5) condition that implies a type (2) condition through type (3) and type (4) conditions were inferred. Since t_{mapres} satisfies these inferred type (2) conditions, it must be that the conjunction of type (4) and type (5) conditions in $\mathcal{P}[t_{mapres}]$ is satisfiable.

Since $\mathcal{P}[t_{mapres}]$ is satisfiable, we can construct t_{res} as follows. For all attributes in $\mathcal{R} - \mathcal{D}$, copy the values from t_{mapres}. For attributes in $\mathcal{D}$, assign values so that $\mathcal{P}[t_{res}]$ is `true`. The existence of these values is guaranteed by the fact that $\mathcal{P}[t_{mapres}]$ is satisfiable. Since $\mathcal{P}[t_{res}]$ is `true`, $t_{res} \in \texttt{Res}(E)$. By Definition C.0.1, t is needed. $\square$

Proof of Lemma 5.4.1 Since we now prove Lemma 5.4.1 which deals with constraints, we assume a maintenance subexpression $E = \pi_{\mathcal{A}}\sigma_{\mathcal{P}}(\times_{R\in\mathcal{R}} R)$ in quantifier representation. We also denote $\texttt{Map}_{\mathcal{C}}(E,T)$ as $\pi_{\texttt{Attrs}(T)}\sigma_{\mathcal{P}'}(\times_{R\in(\mathcal{R}-\mathcal{D})} R)$, where $\mathcal{P}'$ is obtained from $\mathcal{P}$

using $\texttt{Closure}_{\mathcal{C}}$ and Ignore. Before we prove Lemma 5.4.1, recall that the lemma makes three statements: (1) All the needed T tuples are in $\texttt{Needed}_{\mathcal{C}}(T, \mathcal{E}_{\mathcal{C}})$. (2) $\texttt{Needed}_{\mathcal{C}}(T, \mathcal{E}_{\mathcal{C}}) \subseteq \texttt{Needed}(T, \mathcal{E}_{\mathcal{C}}) \subseteq \texttt{Needed}(T, \mathcal{E})$. (3) Under certain restrictions on the `not exists` constraints in $\mathcal{C}$, only the needed T tuples are in $\texttt{Needed}_{\mathcal{C}}(T, \mathcal{E}_{\mathcal{C}})$. We prove these statements in turn.

Proof: (Statement (1), Lemma 5.4.1)
Assume $t \in T$ is needed. By Definition C.0.1, there exist a tuple $t_{res} \in \texttt{Res}(E)$ so that $t_{res}[\{T\}] = t$. We can obtain $t_{mapres} \in \texttt{Res}(\texttt{Map}_{\mathcal{C}}(E, T))$ as $t_{res}[\mathcal{R} - \mathcal{D}]$. This follows from the soundness of the $\texttt{Closure}_{\mathcal{C}}$ procedure and the definition of the Ignore procedure which guarantee that $\mathcal{P} \Rightarrow \mathcal{P}'$. ($\texttt{Closure}_{\mathcal{C}}$ is sound in that it only derives conditions that are implied by $\mathcal{P}$.) It follows that $t = t_{mapres}[\{T\}](= t_{res}[\{T\}])$ since the attribute values were not changed in obtaining t_{mapres} from t_{res}. Hence $t \in \texttt{Map}_{\mathcal{C}}(E, T)$ and $t \in \texttt{Needed}_{\mathcal{C}}(T, \mathcal{E}_{\mathcal{C}})$ assuming $E \in \mathcal{E}_{\mathcal{C}}$. □

Proof: (Statement (2), Lemma 5.4.1)
For each $E \in \mathcal{E}$, $\texttt{Needed}_{\mathcal{C}}(T, \mathcal{E}_{\mathcal{C}})$ uses $\texttt{Map}(E_{\mathcal{C}}, T)$; $\texttt{Needed}(T, \mathcal{E}_{\mathcal{C}})$ uses $\texttt{Map}(E_{\mathcal{C}}, T)$; and $\texttt{Needed}(T, \mathcal{E})$ uses $\texttt{Map}(E, T)$. Let $E = \pi_{\mathcal{A}} \sigma_{\mathcal{P}} (\times_{R \in \mathcal{R}} R)$. Let $E_{\mathcal{C}} = \pi_{\mathcal{A}} \sigma_{\mathcal{P}_q} (\times_{R \in \mathcal{R}} R)$. Since $\mathcal{P}_q \Rightarrow \mathcal{P}$ due to additional `exists` and `not exists` conditions, $\texttt{Map}(E_{\mathcal{C}}, T) \subseteq \texttt{Map}(E, T)$. It follows that $\texttt{Needed}(T, \mathcal{E}_{\mathcal{C}}) \subseteq \texttt{Needed}(T, \mathcal{E})$.

Let $\texttt{Map}(E_{\mathcal{C}}, T) = \pi_{\texttt{Attrs}(T)} \sigma_{\mathcal{P}'_q} (\times_{R \in \mathcal{R}} R)$, and $\texttt{Map}_{\mathcal{C}}(E_{\mathcal{C}}, T) = \pi_{\texttt{Attrs}(T)} \sigma_{\mathcal{P}''_q} (\times_{R \in \mathcal{R}} R)$. More conditions (implied by `exists` and `not exists` conditions) are added in $\mathcal{P}''_q$, and they are not in $\mathcal{P}'_q$. Therefore, $\mathcal{P}''_q \Rightarrow \mathcal{P}'_q$. It follows that $\texttt{Map}_{\mathcal{C}}(E_{\mathcal{C}}, T) \subseteq \texttt{Map}(E_{\mathcal{C}}, T)$. Hence, $\texttt{Needed}_{\mathcal{C}}(T, \mathcal{E}_{\mathcal{C}}) \subseteq \texttt{Needed}(T, \mathcal{E}_{\mathcal{C}})$. □

Proof: (Statement (3), Lemma 5.4.1)
Assume $t \in \texttt{Needed}_{\mathcal{C}}(T, \mathcal{E}_{\mathcal{C}})$. For some $E \in \mathcal{E}_C$, it must be that $t \in \texttt{Map}_{\mathcal{C}}(E, T)$. This implies that $t_{mapres} \in \texttt{Res}(\texttt{Map}_{\mathcal{C}}(E, T))$ with $t_{mapres}[\{T\}] = t$. Since $\texttt{Map}_{\mathcal{C}}(E, T)$ is not empty, the selection condition expression of $\texttt{Map}_{\mathcal{C}}(E, T)$ denoted $\mathcal{P}'$ does not have the atomic condition `false` ,*i.e.*, $\mathcal{P}'$ is satisfiable. Since Ignore does not remove `false`, it must be that the selection condition expression of E, denoted $\mathcal{P}$, is also satisfiable. Furthermore, assuming $\texttt{Closure}_{\mathcal{C}}$ is complete, we are guaranteed that $\mathcal{P}[t_{mapres}]$ is satisfiable.

Since $\mathcal{P}[t_{mapres}]$ is satisfiable, we can construct t_{res} as follows. For all attributes in $\mathcal{R} - \mathcal{D}$, copy the values from t_{mapres}. For attributes in $\mathcal{D}$, assign values so that $\mathcal{P}[t_{res}]$ is `true`. The existence of these values is guaranteed by the fact that $\mathcal{P}[t_{mapres}]$ is satisfiable. Since $\mathcal{P}[t_{res}]$ is `true`, $t_{res} \in \texttt{Res}(E)$. By Definition C.0.1, t is needed. □

The proof of Statement (3) of Lemma 5.4.1 relies on having a complete $\texttt{Closure}_{\mathcal{C}}$ algorithm. Before we present the algorithm and prove its completeness, we introduce some notation. As before, we use S_i to denote an existentially quantified tuple variable over S, and S_j^{asj} to denote a universally quantified tuple variable over S. We use "S" and "S'" to denote either S_i or S_j^{asj}. We use X to denote a reference to some attribute $T.a$. Finally, we use θ to denote either $<$, $\leq$, $\neq$, $=$, $\geq$ or $>$. Since an atomic condition $T.b > S.a$ ($T.b \geq S.a$) can always be expressed as $S.a < T.b$ ($S.a \leq T.b$, respectively), we focus on the first four comparison operators.

We now present in detail the $\texttt{Closure}_{\mathcal{C}}$ algorithm and prove that it is complete. The algorithm uses the following axioms to obtain all atomic conditions implied by $\mathcal{P}$.

The following 8 axioms are for inferring equalities from a conjunction of atomic conditions.

E1: $S.a = S.a$

E2: $S.a = T.b \Rightarrow T.b = S.a$

E3: $S.a = T.b \wedge T.b = U.c \Rightarrow S.a = U.c$

E4: $S.a = T.b \Rightarrow S.a \leq T.b$

E5: $S.a \leq T.b \wedge T.b \leq U.c \Rightarrow S.a \leq U.c$

E6: $S.a \leq T.b \wedge T.b \leq S.a \Rightarrow S.a = T.b$

E7*: $S_i^{asj}.a\ \theta\ T.b \Rightarrow S.a\ \theta\ T.b$

E8*: $S_i^{asj}.a = T_j.b \Rightarrow S_i^{asj}.a = S_k^{asj}.a$

Axioms E1–E6 are fairly standard and are clearly sound. Axioms E7 and E8 are the two additional axioms introduced in Section 5.4 and are also sound.

The following 9 axioms derive inequalities. The first 8 axioms are called Armstrong's axioms, and were proven in [Ull89a] to be sound and complete when none of the tuple variables is universally quantified. Axiom I9 is identical to Axiom E7 and is also sound.

I1: $S.a \leq S.a$

I2: $S.a < T.b \ \Rightarrow\ S.a \leq T.b$

I3: $S.a < T.b \ \Rightarrow\ S.a \neq T.b$

I4: $S.a \leq T.b \wedge S.a \neq T.b \ \Rightarrow\ S.a < T.b$

I5: $S.a \neq T.b \ \Rightarrow\ T.b \neq S.a$

I6: $S.a < T.b \wedge T.b < U.c \ \Rightarrow\ S.a < U.c$

I7: $S.a \leq T.b \wedge T.b \leq U.c \ \Rightarrow\ S.a \leq U.c$

I8: $S.a \leq U.c \wedge U.c \leq T.b \wedge S.a \leq V.d \wedge V.d \leq T.b \wedge U.c \neq V.d \ \Rightarrow \ S.a \neq T.b$
I9*: $S_j^{asj}.a \ \theta \ X \ \Rightarrow \ S_i.a \ \theta \ X$.

We assume we have a procedure `CloseEqual` that fires Axioms E1–E8 and a procedure `CloseInequal` that fires Axioms I1–I9 when given a conjunction of atomic conditions. $\texttt{Closure}_{\mathcal{C}}$ uses the two procedures in the following steps.

1. Use `CloseEqual` to obtain all equality atomic conditions. Using these equality atomic conditions, we place two attributes $S.a$ and $T.b$ in the same equivalence class C if $S.a = T.b$ results from `CloseEqual`.

2. For each equivalence class C, pick an attribute $S_i.a$ where S_i is an existentially quantified tuple variable. For each attribute $T_j.b$ (that is not $S_i.a$) in C, replace each atomic condition $T_j.b \ \theta \ X$ with $S_i.a \ \theta \ X$.

3. Use `CloseInequal` to obtain all the inequality atomic conditions.

4. Add additional atomic conditions by examining each equivalence class. For an equivalence class C, assume $S_i.a$ was the attribute picked in Step 2. For each attribute $T_j.b$ (that is not $S_i.a$) in C, introduce the atomic condition $T_j.b \ \theta \ X$ if $S_i.a \ \theta \ X$ is in the closure.

$\texttt{Closure}_{\mathcal{C}}$ is clearly sound since it does not derive any atomic condition that is not implied by $\mathcal{P}$.

Assuming both `CloseEqual` and `CloseInequal` are complete, it is not hard to show that $\texttt{Closure}_{\mathcal{C}}$ is complete. Suppose there is an atomic condition $T.b \ \theta \ X$ that is implied by the given conjunction of atomic conditions, but it is not derived by $\texttt{Closure}_{\mathcal{C}}$. It must be the case that θ is not $=$ because otherwise `CloseEqual` would have produced it (in Step 1). It must also be the case that T is existentially quantified and $T.b$ is not an attribute that was picked in Step 2 of $\texttt{Closure}_{\mathcal{C}}$. Otherwise, `CloseInequal` would have produced $T.b \ \theta \ X$ in Step 3. However, it is guaranteed that Step 4 produces $T.b \ \theta \ X$. Otherwise, `CloseInequal` must have failed to produce $S_i.a \ \theta \ X$, where $S_i.a$ is the attribute that belongs to the same equivalence class as $T.b$ that was picked in Step 2. This is implies that `CloseInequal` is incomplete, contradicting our assumption. We now prove the completeness of `CloseEqual` and `CloseInequal`.

We prove the completeness of `CloseEqual` by proving the following lemma which states that given a conjunction of atomic conditions $\mathcal{P}$ input to $\texttt{Closure}_{\mathcal{C}}$, there is no $S.a = T.b$ that is implied by $\mathcal{P}$ but is not in $\mathcal{P}^+$ (the output of `CloseEqual`).

Lemma C.0.4 *Let $\mathcal{P}$ be a conjunction of atomic conditions input to $\texttt{Closure}_{\mathcal{C}}$, such that* `false` *is not implied by $\mathcal{P}$. Then every equality $S.a = T.b$ not in $\mathcal{P}^+$ has some assignment of a set of integers to each attribute used in $\mathcal{P}$ that makes all the atomic conditions in $\mathcal{P}^+$ true but $S.a\ \theta\ T.b$ false.* □

Proof: (Lemma C.0.4 (Completeness of `CloseEqual`))
After applying the set of axioms, we derive a set of equivalent classes where each equivalent class contains a set of attributes that are inferred to be equal (from Axioms E1–E3, E6–E8). We construct a graph as follows: Each node N in the graph corresponds to an equivalent class of attributes $N.attrs$. There is a directed edge from node N to node M iff $U.c < V.d$ or $U.c \leq V.d$ for some $U.c$ in $M.attrs$ and $V.d$ in $N.attrs$.

We now show that the graph is acyclic. Suppose that there is a directed cycle. This means that we have a chain of inequalities. Since we assume $\mathcal{P}$ is satisfiable, this chain must consist of only $\leq$. However, Axioms E5 and E6, we should have easily derived the fact that all attributes in these nodes are equal, i.e., all attributes belong to one equivalent class. This is a contradiction. Therefore, the graph is a DAG.

We assign values to attributes as follows: First find a topological sort of the graph such that M comes before N in the order if there is an edge from N to M. Assign a strictly increasing sequence of integers to nodes in the topological order. Attributes corresponding to the same node are assigned the same integer. (Note that the attribute of a universally quantified tuple will have only one integer value.)

We show that this assignment satisfies $\mathcal{P}$. If $U.c = V.d$ is in $\mathcal{P}^+$, then $U.c$ and $V.d$ belong to the same node in the graph, and hence are assigned to the same integer. If $U.c! = V.d$ is in $\mathcal{P}^+$, then $U.c$ and $V.d$ belong to different nodes (otherwise $\mathcal{P}$ is unsatisfiable), and hence are assigned to different integers. If $U.c \leq V.d$ is in $\mathcal{P}^+$ and $U.c$ and $V.d$ correspond to the same node, then $U.c \leq V.d$ is satisfied because they are assigned to the same integer. If $U.c \leq V.d$ is in $\mathcal{P}^+$ but $U.c$ and $V.d$ correspond to different nodes, there should be an edge from the node for $V.d$ to the node for $U.d$, so the assignment guarantees that $U.c < V.d$. If $U.c < V.d$ is in $\mathcal{P}^+$, then $U.c$ and $V.d$ must correspond to different nodes (otherwise $\mathcal{P}$ is unsatisfiable), and there should be an edge from the node for $V.d$ to the node for $U.c$, so the assignment guarantees that $U.c < V.d$. In conclusion, $\mathcal{P}$ is satisfied by this attribute assignment.

Now, suppose that $S.a = T.b$ cannot be derived from $\mathcal{P}$ using the set of axioms. $S.a$ and $T.b$ must correspond to different nodes in the graph, or else $S.a = T.b$ is already inferred. However, different nodes are assigned to different integers. Therefore $S.a = T.b$ does not hold under this assignment. □

We prove the completeness of `CloseInequal` by proving the following lemma which states that given a conjunction of atomic conditions $\mathcal{P}$ produced by Steps 1–2 of $\texttt{Closure}_{\mathcal{C}}$, there is no inequality $S.a\ \theta\ T.b$ that is implied by $\mathcal{P}$ but is not in $\mathcal{P}^+$ (the output of `CloseInequal`). The proof extends the one presented in [Ull89a] which only handled selection, join and semi-join conditions.

Lemma C.0.5 *Let $\mathcal{P}$ be a conjunction of atomic conditions produced by Steps 1–2 of $\texttt{Closure}_{\mathcal{C}}$, such that `false` is not implied by $\mathcal{P}$. Then every inequality $S.a\ \theta\ T.b$ not in $\mathcal{P}^+$ has some assignment of a set of integers to each attribute used in $\mathcal{P}$ that makes all the atomic conditions in $\mathcal{P}^+$ true but $S.a\ \theta\ T.b$ false.* □

Proof: (Lemma C.0.5 (Completeness of `CloseInequal`))
The inequality $S.a\ \theta\ T.b$ can be of three types. For now we assume that neither S nor T are universally quantified tuple variables.

Case 1: θ is $\leq$. We now construct an assignment that satisfies $\mathcal{P}^+$ but makes $S.a > T.b$. Let $\mathcal{A}$ be those attributes $U.c$ for which $S.a \leq U.c$ is in $\mathcal{P}^+$, and let $\mathcal{B}$ be those attributes $V.d$ for which $V.d \leq T.b$ is in $\mathcal{P}^+$. Let $\mathcal{C} = \mathcal{V} - \mathcal{A} - \mathcal{B}$, where $\mathcal{V}$ is the set of attributes used by $\mathcal{P}$. Note that for any attribute $U.c \in \mathcal{A}$, $V.d \in \mathcal{B}$, and $W.e \in \mathcal{C}$, it is possible that $V.d \leq W.e$ and/or $W.e \leq U.c$, but not $W.e \leq V.d$ (else by Axiom I7, $W.e$ would be in $\mathcal{B}$), nor $U.c \leq W.e$ (then, $W.e$ would be in $\mathcal{A}$). Also, $\mathcal{A}$ and $\mathcal{B}$ are disjoint since otherwise $S.a \leq T.b$ would be in $\mathcal{P}^+$ (by Axiom I7) contrary to our assumption. Since $\mathcal{C}$ is disjoint from $\mathcal{A}$ and $\mathcal{B}$, we conclude that all three attribute sets are disjoint.

We can now topologically sort the elements of each attribute set with respect to the order $\leq$. That is, $U.c \in \mathcal{A}$ comes before $U'.c' \in \mathcal{A}$ if $U.c \leq U'.c'$ is in $\mathcal{P}^+$ (and likewise for the elements of $\mathcal{B}$ and $\mathcal{C}$). There may be cycles in the order, i.e., we derive both $U.c \leq U'.c'$ and $U'.c' \leq U.c$. In this case, it is guaranteed that one of U and U' is universally quantified. U and U' cannot be both existentially quantified since Step 2 of $\texttt{Closure}_{\mathcal{C}}$ picks only one attribute of one existentially quantified variable from each equivalence class. Given that one of U and U' is universally quantified, we break the cycle arbitrarily by assuming $U.c$ comes before $U'.c'$.

We can then order the attributes in $\mathcal{V}$ as follows: (1) the elements in $\mathcal{B}$, in order; (2) the elements in $\mathcal{C}$, in order; and (3) the elements in $\mathcal{A}$, in order. We can then initially assign distinct integers 1,2,... to the attributes in order. If some attribute $R.a \in \mathcal{V}$ maps to integer n, we denote this as $\text{IntMap}(R.a) = \{n\}$. For an attribute $R_i.a \in \mathcal{V}$ (i.e., R_i is an existentially quantified tuple variable), this is the final IntMap assignment of $R_i.a$. For an attribute $R_j^{asj}.a$, its final IntMap assignment depends on the equivalence classes determined in Step 1 of $\texttt{Closure}_{\mathcal{C}}$. If there is some attribute $S_i.b$ (attribute of some existentially quantified variable S_i) that is in the same equivalence class as $R_j^{asj}.a$, we set $\text{IntMap}(R_j^{asj}.a)$ to $\text{IntMap}(S_i.b)$. If there is no such attribute $S_i.b$ but there is an attribute $S_i^{asj}.b$ (attribute of some universally quantified variable S_i^{asj}), we set $\text{IntMap}(R_j^{asj}.a)$ to $\text{IntMap}(S_i^{asj}.b)$. If $R_j^{asj}.a$'s equivalence class has no other elements other than $R_j^{asj}.a$, we assign a set of integers to $R_j^{asj}.a$ as follows: $\text{IntMap}(R_j^{asj}.a) = \bigcup_{\forall R.a \in \mathcal{V}} \text{IntMap}(R.a)$, where R is either existentially quantified or universally quantified.

For this IntMap assignment, $S.a$ is given a larger value than $T.b$, so $S.a \leq T.b$ does not hold. Now we must show that all the atomic conditions in $\mathcal{P}^+$ hold.

Consider $U.c \neq V.d$ in $\mathcal{P}^+$. This clearly holds if $U.c$ and $V.d$ are in different attribute sets (i.e., $\mathcal{A}$, $\mathcal{B}$ or $\mathcal{C}$), because IntMap assigns a disjoint set of integers to attributes belonging to different attribute sets. If $U.c$ and $V.d$ are in the same attribute set, they must be in different equivalence classes as determined in Step 1 of $\texttt{Closure}_{\mathcal{C}}$. Otherwise, $U.c = V.d$ would have been derived by `CloseEqual` implying that $\mathcal{P}$ is contradictory. Furthermore, it cannot be the case that $U.c$ is actually $R_i.c$, and $V.d$ is actually $R_i.c$ as well. Otherwise, Axiom E1 would have derived $U.c = V.d$. Finally, it cannot be the case that $U.c$ is actually $R_i.c$, and $V.d$ is actually $R_j^{asj}.c$. This is because Axiom I9 would derive $U.c \neq U.c$, which indicates that $\mathcal{P}$ is contradictory. $\text{IntMap}(U.c) = \text{IntMap}(V.d)$ only holds if the attributes belong to the same equivalence class. $\text{IntMap}(U.c) \subset \text{IntMap}(V.d)$ only holds if $U.c$ is actually $R_i.c$, and $V.d$ is actually $R_j^{asj}.c$. Similarly, $\text{IntMap}(V.d) \subset \text{IntMap}(U.c)$ only holds if $U.c$ is actually $R_j^{asj}.c$, and $V.d$ is actually $R_i.c$. These three cases are avoided by using `CloseEqual` and Axiom I9. Hence, any $U.c \neq V.d$ in $\mathcal{P}^+$ always holds.

Now consider $U.c \leq V.d$ in $\mathcal{P}^+$. Let us suppose that $U.c$ and $V.d$ are in the same set of attributes ($\mathcal{A}$, $\mathcal{B}$, or $\mathcal{C}$). If both U and V are existentially quantified variables, then $U.c \leq V.d$ since the topological order within each attribute set respects $\leq$. If U is a universally quantified variable and V is not, $U.c \leq V.d$ holds. To see this, Axiom I9 ensures that $U.c \leq V'.d$ is derived for each tuple variable V' that goes over the same view as V. Hence, IntMap assigns a set of values to $V.d$, where each value in the set is greater or equal to than IntMap($U.c$). On the other hand, if U is a universally quantified variable and V is not, $U.c \leq V.d$ holds. To see this, Axiom I9 ensures that $U'.c \leq V.d$ is derived for each tuple variable U' that goes over the same view as U. Hence, IntMap assigns a set of values to $U.c$, where each value in the set is less than or equal to IntMap($V.d$). Finally, if both U and V are universally quantified, Axiom I9 ensures that $U'.c \leq V'.d$ is derived for each pair of tuple variables U' and V'. Hence, IntMap assigns a set of values to $U.c$ and a set of values to $V.d$, such that each value in IntMap($V.d$) is greater than each value in IntMap($U.c$). Note that if $U.c$ is actually $T_i^{asj}.c$ and $V.d$ is actually $T_j^{asj}.c$, then IntMap assigns the same singleton set of integers to $U.c$ and $V.d$. Hence, $U.c \leq V.d$ still holds.

So far, we have shown that $U.c \leq V.d$ holds if both $U.c$ and $V.d$ are in the same attribute set. We now show that it holds even if $U.c$ and $V.d$ are in different attribute sets. Surely, if $V.d$ is in $\mathcal{C}$, or $U.c$ is in $\mathcal{A}$ and $V.d$ is in $\mathcal{B}$ or $\mathcal{C}$, $U.c \leq V.d$ holds. We are left with the possibility that $U.c$ is in $\mathcal{A}$ and $V.d$ is in $\mathcal{C}$ or $\mathcal{B}$, or $U.c$ is in $\mathcal{C}$ and $V.d$ is in $\mathcal{B}$. However, if $U.c$ is in $\mathcal{A}$ and $U.c \leq V.d$ is in $\mathcal{P}^+$, then $V.d$ would be in $\mathcal{A}$ by Axiom I7, and not in $\mathcal{B}$ nor $\mathcal{C}$. Similarly, if $V.d$ is in $\mathcal{B}$ then it is not possible that $U.c \leq V.d$ and $U.c$ is in $\mathcal{C}$, because $U.c$ would have to be in $\mathcal{B}$, by Axiom I7.

Finally, we must consider $U.c < V.d$ in $\mathcal{P}^+$. We can rule out the possibility that $U.c$ is actually $R.c$ and $V.d$ is actually $R'.c$, and one of R or R' is universally quantified. Otherwise, we can derive either $R.c < R.c$ or $R'.c < R'.c$ which implies that $\mathcal{P}$ is contradictory. With these possibilities ruled out, the argument that $U.c \leq V.d$ is true holds for $U.c < V.d$ as well.

Case 2: θ is $\neq$. We now construct an assignment that satisfies $\mathcal{P}^+$ but makes $S.a = T.b$. Once the construction is done, many of the arguments for Case 1 hold for the present case as well. Since the present case considers that $S.a$ and $T.b$ are not equal, let us suppose that $S.a$ is less than $T.b$. Let $\mathcal{D}$ be those attributes $W.e$ such that $S.a \leq W.e$ and $W.e \leq T.b$ are in $\mathcal{P}^+$, which includes $S.a$ and $T.b$ themselves. Let $\mathcal{A}$ be those attributes $U.c$ for which $X \leq U.c$ is in $\mathcal{P}^+$, for some $X \in \mathcal{D}$, but $U.c$ itself is not in $\mathcal{D}$. Let $\mathcal{B}$ be those attributes $V.d$ for which $V.d \leq X$ is in $\mathcal{P}^+$, for some $X \in \mathcal{D}$, but $V.d$ itself is not in $\mathcal{D}$. Let $\mathcal{C} = \mathcal{V} - \mathcal{A} - \mathcal{B} - \mathcal{D}$, where $\mathcal{V}$ is the set of attributes used in $\mathcal{P}$. As in Case 1, it can be easily shown that $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, and $\mathcal{D}$ are disjoint based on the axioms.

We then topologically sort the elements in each attribute set w.r.t. $\leq$. We then combine the attributes into one sequence with the attributes in $\mathcal{B}$ first, $\mathcal{C}$ second, $\mathcal{D}$ third and $\mathcal{A}$ last. We initially assign increasing distinct integers to each attribute except for the attributes in $\mathcal{D}$, where the same integer is assigned. The IntMap function introduced in Case 1 is used to give the final assignments to each attribute.

Clearly, $S.a = T.b$ since both $S.a$ and $T.b$ are in $\mathcal{D}$. We now show that all the atomic conditions in $\mathcal{P}^+$ hold.

Consider $U.c \neq V.d$ in $\mathcal{P}^+$. As in Case 1, if $U.c$ and $V.d$ are in different attribute sets (i.e., $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, and $\mathcal{D}$), then $U.c \neq V.d$ holds. If $U.c$ and $V.d$ belong to the same attribute set, the argument given in Case 1 that $U.c \neq V.d$ holds for attribute sets $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$. Although the argument does not hold for $\mathcal{D}$, $U.c$ and $V.d$ cannot be in $\mathcal{D}$. If they were, Axiom I8 derives that $S.a \neq T.b$ contradicting our assumption that $S.a \neq T.b$ is not in $\mathcal{P}^+$.

Consider $U.c \leq V.d$. If $U.c$ and $V.d$ are in the same attribute set, the argument given in Case 1 holds if $U.c$ and $V.d$ are either in $\mathcal{A}$, $\mathcal{B}$ or $\mathcal{C}$. Since all the attributes in $\mathcal{D}$ are assigned the same integer, $U.c \leq V.d$ holds if $U.c, V.d \in \mathcal{D}$.

So far, we have shown that $U.c \leq V.d$ holds if both $U.c$ and $V.d$ are in the same attribute set. We now show that it holds even if $U.c$ and $V.d$ are in different attribute sets. Clearly, there are many cases where $U.c$ and $V.d$ reside in different attributes sets, and $U.c \leq V.d$ still holds by virtue of the ordering imposed on the sets (i.e., $\mathcal{B}$, $\mathcal{C}$, $\mathcal{D}$, $\mathcal{A}$). For instance, if $U.c$ is in $\mathcal{D}$ and $V.c$ is in $\mathcal{A}$, then surely $U.c \leq V.d$ holds. We now consider the following possibility – $U.c$ is in $\mathcal{A}$ and $V.d$ is in some other attribute set. Hence, $X \leq U.c$, where $X \in \mathcal{D}$, must be in $\mathcal{P}^+$. However, $X \leq V.d$ is derived by Axiom I7 and $V.d$ must also be in $\mathcal{A}$. We now consider the following possibility – $V.d$ is in $\mathcal{B}$ and $U.c$ is either in $\mathcal{C}$ or $\mathcal{D}$. In this case, Axiom I7 enforces that $U.c$ must also be in $\mathcal{B}$. We are left with the possibility that $U.c$ is in $\mathcal{D}$, and $V.d$ is in $\mathcal{B}$. Since $U.c \leq V.d$ is in $\mathcal{P}^+$ and $U.c \in \mathcal{D}$, by definition $V.d$ must be in $\mathcal{A}$. Because the sets are ordered as $\mathcal{B}$, $\mathcal{C}$, $\mathcal{D}$, then $\mathcal{A}$, $U.c \leq V.d$ must hold.

Finally, we must consider $U.c < V.d$ in $\mathcal{P}^+$. As in Case 1, we can rule out the possibility that $U.c$ is actually $R.c$ and $V.d$ is actually $R'.c$, and one of R or R' is universally quantified. Otherwise, we can derive either $R.c < R.c$ or $R'.c < R'.c$ which implies that $\mathcal{P}$ is contradictory. With these possibilities ruled out, the argument that $U.c \leq V.d$ is true holds for $U.c < V.d$ as well.

Case 3: θ is $<$. If $S.a \leq T.b$ is not in $\mathcal{P}^+$, then use the construction of Case 1 where IntMap makes all the atomic conditions in $\mathcal{P}^+$ true but makes $S.a < T.b$ false (i.e., $S.a > T.b$ is true). If $S.a \neq T.b$ is not in $\mathcal{P}^+$, then use the construction of Case 2 where IntMap makes all atomic conditions in $\mathcal{P}^+$ true but makes $S.a \neq T.b$ false (i.e., $S.a = T.b$ is true). If both $S.a \leq T.b$ and $S.a \neq T.b$ is in $\mathcal{P}^+$, then by Axiom I4, $S.a < T.b$ is in $\mathcal{P}^+$ as well contrary to our assumption.

So far, we have assumed that neither S nor T in $S.a\ \theta\ T.b$ is universally quantified. Given this assumption, we have proved that if $S.a\ \theta\ T.b$ is not in $\mathcal{P}^+$, then there is an assignment that makes all the atomic conditions in $\mathcal{P}^+$ true but not $S.a\ \theta\ T.b$. We note that the IntMap assignment that makes $S.a\ \theta\ T.b$ false can be used if S (or T) is universally quantified. Suppose S_i^{asj} is a universally quantified tuple variable going over the same view as S. Since $S_i^{asj}.a\ \theta\ T.b$ implies $S.a\ \theta\ T.b$, the IntMap assignment that makes $S.a\ \theta\ T.b$ false also makes $S_i^{asj}.a\ \theta\ T.b$ false. $\square$

Bibliography

[AASY97] D. Agrawal, A. El Abbadi, A. Singh, and T. Yurek. Efficient view maintenance in data warehouses. In Peckham [Pec97], pages 417–425.

[ABB93] Rakesh Agrawal, Sean Baker, and David Bell, editors. *Proceedings of the Nineteenth International Conference on Very Large Databases*, Dublin, Ireland, August 24-27 1993.

[AGPR99] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. The AQUA approximate query answering system. In A. Delis, C. Faloutsos, and S. Ghandeharizadeh, editors, *Proceedings of ACM SIGMOD 1999 International Conference on Management of Data*, pages 574–576, Philadelphia, Pennsylvania, June 1999.

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley Publishing Company, 1995.

[AL80] M. E. Adiba and B. Lindsay. Database snapshots. In *Proceedings of the Sixth International Conference on Very Large Databases*, pages 86–91, Montreal, Canada, October 1980.

[Ass92] Association for Computing Machinery. *Proceedings of ACM SIGMOD 1992 International Conference on Management of Data*, San Diego, California, June 2–5 1992.

[Ass95] Association for Computing Machinery. *Proceedings of the Fourteenth Symposium on Principles of Database Systems (PODS)*, San Jose, CA, May 22-24 1995.

[BCL89] J. Blakeley, N. Coburn, and P. Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Transactions on Database Systems*, 14(3):369–400, September 1989.

[BDGM95] S. Brin, J. Davis, and H. Garcia-Molina. Copy detection mechanisms for digital documents. In Carey and Schneider [CS95], pages 328–339.

[BGMF88] D. Barbara, H. Garcia-Molina, and B. Feijoo. Exploiting symmetries for low-cost comparison of file copies. In *Proceedings of the International Conference on Distributed Computing Systems*, San Jose, California, June 1988.

[BHM90] P. A. Bernstein, M. Hsu, and B. Mann. Implementing recoverable requests using queues. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 112–122. Association for Computing Machinery, May 23–25 1990.

[BLT86] J. Blakeley, P. Larson, and F. Tompa. Efficiently Updating Materialized Views. In Zaniolo [Zan86], pages 61–71.

[BN97] P. A. Bernstein and E. Newcomer. *Principles of Transaction Processing.* Morgan Kaufmann, San Mateo, CA, 1997.

[BPT97] E. Baralis, S. Paraboschi, and E. Teniente. Materialized view selection in a multi-dimensional datacube. In *VLDB*, pages 156–165, 1997.

[BT88] J. A. Blakeley and F. W. Tompa. Maintaining materialized views without accessing base data. *Information Systems*, 13(4):393–406, 1988.

[Car97] Felipe Carino. High-performance, parallel warehouse servers and large-scale applications, October 1997. Talk about Teradata given in Stanford Database Seminar.

[CBC93] S. Choenni, H. Blanken, and T. Chang. On the selection of secondary indices in relational databases. *Data and Knowledge Engineering*, 11:207–33, 1993.

[CD97] S. Chaudhuri and U. Dayal. An overview of data warehousing and olap technology. *Sigmod Record*, 26(1):65–74, March 1997.

[CGL+96] L. Colby, T. Griffin, L. Libkin, I. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *Proceedings of ACM SIGMOD 1996 International Conference on Management of Data*, pages 469–480, 1996.

[CKL+97] L. Colby, A. Kawaguchi, D. Lieuwen, I. Mumick, and K. Ross. Supporting multiple view maintenance policies. In Peckham [Pec97], pages 405–416.

[CLR92] T. H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1992.

[CN97] S. Chaudhuri and V. Narasaya. An efficient, cost-driven index selection tool for microsoft sql server. In Jarke et al. [JCD+97], pages 146–155.

[Com] TPC Committee. Transaction Processing Council. Available at: http://www.tpc.org/.

[CRGMW96] S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In Jagadish and Mumick [JM96].

[CS95] M. Carey and D. Schneider, editors. *Proceedings of ACM SIGMOD 1995 International Conference on Management of Data*, San Jose, CA, May 23-25 1995.

[CW91] Stefano Ceri and Jennifer Widom. Deriving production rules for incremental view maintenance. In Lohman et al. [LSC91], pages 108–119.

[DGN95] Umeshwar Dayal, Peter M.D. Gray, and Shojiro Nishio, editors. *Proceedings of the 21^{st} International Conference on Very Large Databases*, Zurich, Switzerland, September 11-15 1995.

[DT87] Umeshwar Dayal and Irv Traiger, editors. *Proceedings of ACM SIGMOD 1987 International Conference on Management of Data*, San Francisco, CA, May 27-29 1987.

[For82] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[FRS93] F. Fabret, M. Regnier, and E. Simon. An adaptive algorithm for incremental evaluation of production rules in database. In Agrawal et al. [ABB93], pages 455–66.

[FST88] S. Finkelstein, M. Schkolnick, and P. Tiberio. Physical database design for relational databases. *ACM Transactions on Database Systems*, 13(1):91–128, 1988.

[FWA86] W. K. Fuchs, K. Wu, and J. Abraham. Low-cost comparison and diagnosis of large remotely located files. In *Proceedings of the Fifth Symposium on Reliability in Distributed Software and Database Systems*, January 1986.

[GHQ95] A. Gupta, V. Harinarayan, and D. Quass. Generalized projections: A powerful approach to aggregation. In Dayal et al. [DGN95].

[GJM96] A. Gupta, H. Jagadish, and I. Mumick. Data integration using self-maintainable views. In *Proceedings of the Fifth International Conference on Extending Database Technology*, Avignon, France, March 1996. Industrial Session.

[GL95] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In Carey and Schneider [CS95], pages 328–339.

[GM95] A. Gupta and I. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. [LW95], pages 3–19.

[GM98] H. Gupta and I. Mumick. Selection of views to materialize under a maintenance-time constraint. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 453–470, 1998.

[GMLY98] H. Garcia-Molina, W. Labio, and J. Yang. Expiring data in a warehouse. In Gupta et al. [GSW98], pages 500–511.

[GMS87] H. Garcia-Molina and K. Salem. Sagas. In Dayal and Traiger [DT87], pages 249–259.

[GMS93] A. Gupta, I. Mumick, and V. Subrahmanian. Maintaining views incrementally. In *Proceedings of ACM SIGMOD 1993 International Conference on Management of Data*, Washington, DC, May 26-28 1993.

[Gol95] Rob Goldring. IBM Datapropagator relational application guide. *IBM White Paper*, 1(1), 1995.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, San Mateo, CA, 1993.

[Gra93] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[GSW98] A. Gupta, O. Shmueli, and J. Widom, editors. *Proceedings of the 24^{th} International Conference on Very Large Databases*, New York, New York, August 24-27 1998.

[Gup97] H. Gupta. Selection of views to materialize in a data warehouse. In *Proceedings of the International Conference on Database Theory (ICDT)*, 1997.

[Han87] E. Hanson. A performance analysis of view materialization strategies. In Dayal and Traiger [DT87], pages 440–453.

[Han92] E. Hanson. Rule condition testing and action execution in Ariel. In *Proceedings of ACM SIGMOD 1992 International Conference on Management of Data* [Ass92], pages 49–58.

[HC94] L. Haas and M. Carey. SEEKing the truth about ad hoc join costs. Technical report, IBM Almaden Rsearch Center, 1994.

[HGMW$^+$95] J. Hammer, H. Garcia-Molina, J. Widom, W. Labio, and Y. Zhuge. The Stanford Data Warehousing Project. [LW95], pages 41–48.

[HHW97] J. M. Hellerstein, P. J. Haas, and H.J. Wang. Online aggregation. In Peckham [Pec97], pages 171–182.

[HRU96] V. Harinarayan, A. Rajaraman, and J.D. Ullman. Implementing data cubes efficiently. In Jagadish and Mumick [JM96], pages 205–216.

[HT77] J.W. Hunt and Szymanski T.G. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5), 1977.

[Huy97] P. Huyn. Multiple-view self-maintenance in data warehousing environment. In Jarke et al. [JCD$^+$97], pages 26–35.

[HZ96] Richard Hull and Gang Zhou. A framework for supporting data integration using the materialized and virtual approaches. In Jagadish and Mumick [JM96].

[IC94] W.H. Inmon and E. Conklin. Loading data into the warehouse. *Tech Topic*, 1(11), 1994.

[Inf] Informatica. Powermart 4.0 overview. Available at: http://www.informatica.com/pm_tech_over.html.

[Inm92] W. H. Inmon. *Building the Data Warehouse.* Johm Wiley, 1992.

[Inm96] W. H. Inmon. *The Data Warehouse Toolkit.* Johm Wiley, 1996.

[JCD+97] Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors. *Proceedings of the 23rd International Conference on Very Large Databases*, Athens, Greece, August 25-29 1997.

[JM96] H. V. Jagadish and Inderpal Singh Mumick, editors. *Proceedings of ACM SIGMOD 1996 International Conference on Management of Data*, Montreal, Canada, June 1996.

[JMS95] H. Jagadish, I. Mumick, and A. Silberschatz. View maintenance issues in the chronicle data model. In *Proceedings of the Fourteenth Symposium on Principles of Database Systems (PODS)* [Ass95], pages 113–124.

[JNSS97] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, and S. Sudarshan. Incremental organization for data recording and warehousing. In Jarke et al. [JCD+97], pages 16–25.

[Kin81] J. J. King. QUIST : A system for semantic query optimization in relatio nal data bases. In *Proceedings of the Seventh International Conference on Very Large Databases*, pages 510–517, Cannes, France, September 1981.

[KR87] B. Kähler and O. Risnes. Extended logging for database snapshots. In Stocker and Kent [SK87], pages 389–398.

[KR98] Y. Kotidis and N. Roussopoulos. An alternative storage organization for rolap aggregate views based on cubetrees. In *VLDB*, pages 249–258, 1998.

[Lev96] A. Y. Levy. Obtaining complete answers from incomplete databases. In Vijayaraman et al. [VBMS96], pages 402–412.

[LGM95] W.J. Labio and H. Garcia-Molina. Comparing very large database snapshots. Technical Report STAN-CS-TN-95-27, Computer Science Department, Stanford University, June 1995.

[LGM96] W. Labio and H. Garcia-Molina. Efficient snapshot differential algorithms for data warehousing. In Vijayaraman et al. [VBMS96], pages 63–74.

[LHM+86] B. Lindsay, L. Haas, C. Mohan, H. Pirahesh, and P. Wilms. A snapshot differential refresh algorithm. In Zaniolo [Zan86], pages 53–60.

[LMSS95] A. Levy, A. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *Proceedings of the Fourteenth Symposium on Principles of Database Systems (PODS)* [Ass95], pages 95–104.

[Loh85] G. Lohman. Query processing in R*. In *Query Processing in Database Systems*, Berlin, West Germany, March 1985.

[LSC91] G. M. Lohman, A. Sernadas, and R. Camps, editors. *Proceedings of the Seventeenth International Conference on Very Large Databases*, Barcelona, Spain, September 3-6 1991.

[LW95] D. Lomet and J. Widom, editors. *Special Issue on Materialized Views and Data Warehousing,* IEEE Data Engineering Bulletin 18(2), June 1995.

[LYGM99] W. J. Labio, R. Yerneni, and H. Garcia-Molina. Shrinking the warehouse update window. Technical report, Stanford University, 1999. Available at http://www-db.stanford.edu/pub/papers/setvm.ps.

[ME92] P. Mishra and M. Eich. Join processing in relational databases. *ACM Computing Surveys*, 24(1), 1992.

[Min88] J. Minker, editor. *Foundations of Semantic Query Optimization for Deductive Databases.* Morgan Kaufmann, Washington D.C., 1988.

[Mir87] D. P. Miranker. Treat: A better match algorithm for ai production systems. In *Proceedings of AAII 87 Conference on Artificial Intelligence*, pages 42–47, August 1987.

[ML89] L.F. Mackert and G.M. Lohman. Index scans using a finite lru buffer: A validated i/o model. *ACM Transactions on Database Systems*, 14(3):401–24, 1989.

[MN92] C. Mohan and I. Narang. Algorithms for creating indexes for very large tables without quiescing updates. In *Proceedings of ACM SIGMOD 1992 International Conference on Management of Data* [Ass92], pages 361–370.

[MQM97] I. S. Mumick, D. Quass, and B. S. Mumick. Maintenance of data cubes and summary tables in a warehouse. In Peckham [Pec97], pages 100–111.

[MW94] U. Manber and S. Wu. Glimpse: A tool to search through entire file systems. In *Proceedings of the winter USENIX Conference*, January 1994.

[Nil71] N. Nilsson. *Problem Solving Methods in Artificial Intelligence*. McGraw-Hill, 1971.

[NY82] M. Nicholas and K. Yazdanian. Integrity checking in deductive databases. In H. Galliere and J. Minker, editors, *Logic and Databases*, pages 325–346. Plenum Press, 1982.

[Pec97] J. Peckham, editor. *Proceedings of ACM SIGMOD 1997 International Conference on Management of Data*, Tucson, Arizona, May 1997.

[PMW90] B. Partee, A. Meulen, and R. Wall. *Mathematical Methods in Linguistics*. Kluwer Academic Publishers, 1990.

[QGMW96] D. Quass, A. Gupta, I. Mumick, and J. Widom. Making views self-maintainable for data warehousing. In *Proceedings of the Fifth International Conference on Parallel and Distributed Information Systems (PDIS)*, pages 158–169, December 1996.

[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.

[Qua97] Dallan Quass. *Materialized Views in Data Warehouses.* PhD thesis, Stanford University, Stanford, CA 94305, 1997.

[QW91] Xiaolei Qian and Gio Wiederhold. Incremental recomputation of active relational expressions. *IEEE Transactions on Knowledge and Data Engineering*, pages 337–341, 1991.

[Rou82] N. Roussopoulos. View indexing in relational databases. *ACM Transactions on Database Systems*, 7(2):258–90, 1982.

[Rou91] Nick Roussopoulos. The incremental access method of view cache: Concept, algorithms, and cost analysis. *ACM Transactions on Database Systems*, 16(3):535–563, September 1991.

[RS91] S. Rozen and D. Shasha. A framework for automating physical database design. In Lohman et al. [LSC91], pages 401–11.

[RSS96] K. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In Jagadish and Mumick [JM96], pages 447–458.

[RZ89] R. Reinsch and M. Zimowski. Method for restarting a long-running, fault-tolerant operation in a transaction-oriented data base system without burdening the system log. U.S. Patent 4,868,744, IBM, September 1989.

[Sag98] Sagent Technology, Inc., Palo Alto, CA. *Sagent Data Mart Population Guide*, 1998.

[SDJL96] D. Srivastava, S. Dar, H. V. Jagadish, and A. Levy. Answering queries with aggregation using views. In Vijayaraman et al. [VBMS96], pages 318–329.

[SDN98] A. Shukla, P. Deshpande, and J. F. Naughton. Materialized view selection for multidimensional datasets. In Gupta et al. [GSW98], pages 488–499.

[Sel88] T. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems*, 13(1):23–52, 1988.

[SF91] A. Segev and W. Fang. Optimal update policies for distribtued materialized views. *Management Science*, 17(7):851–70, 1991.

[SGM95] N. Shivakumar and H. Garcia-Molina. Scam: A copy detection mechanism for digital documents. In *Proceedings of the 2nd International Conference in Theory and Practice of Digital Libraries*, Austin, Texas, June 1995.

[Sha86] L. Shapiro. Join processing in database systems with large main memories. *ACM Transactions on Database Systems*, 11(3), 1986.

[SK87] P. Stocker and W. Kent, editors. *Proceedings of the Thirteenth International Conference on Very Large Databases*, Brighton, England, September 1-4 1987.

[SO89] S. Shenoy and Z. Ozsoyoglu. Design and implementation of a semantic query optimizer. *IEEE Transactions on Knowledge and Data Engineering*, 1(3):344–361, 1989.

[SP89] A. Segev and J. Park. Updating distributed materialized views. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):173–184, June 1989.

[ST85] M. Schkolnick and P. Tiberio. Estimating the cost of updates in a relational database. *ACM Transactions on Database Systems*, 10(2):163–79, 1985.

[SZ91] A. Segev and J.L. Zhao. Data management for large rule systems. In Lohman et al. [LSC91], pages 297–307.

[Tec] Sagent Technologies. Personal correspondence with customers.

[TS97] D. Theodoratos and T. Sellis. Data warehouse configuration. In Jarke et al. [JCD+97], pages 126–135.

[TSI94] Odysseas G. Tsatalos, Marvin H. Solomon, and Yannis E. Ioannidis. The GMAP: A versatile tool for physical data independence. In Jorge Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *Proceedings of the 20^{th} International Conference on Very Large Databases*, pages 367–378, Santiago, Chile, September 12-15 1994.

[Ull88] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume 1.* Computer Science Press, 1988.

[Ull89a] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume 2.* Computer Science Press, 1989.

[Ull89b] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volumes 1 and 2.* Computer Science Press, 1989.

[UW97] Jeffrey D. Ullman and Jennifer Widom. *A First Course in Database Systems.* Prentice Hall, 1997.

[VBMS96] T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors. *Proceedings of the 22^{nd} International Conference on Very Large Databases*, Bombay, India, September 3-6 1996.

[Vis98] D. Vista. Incremental view maintenance as an optimization problem. In *Proceedings of the Sixth International Conference on Extending Database Technology*, pages 374–388, Valencia, Spain, March 1998.

[WCK93] A. Witkowski, F. Cariño, and P. Kostamaa. NCR 3700 — The Next-Generation Industrial Database Computer. In Agrawal et al. [ABB93], pages 230–243.

[WH92] Y. Wang and E. Hanson. A performance comparison of the rete and treat algorithms for testing database rule conditions. In Li-Yan Yuan, editor, *Proceedings of the Eighteenth International Conference on Very Large Databases*, pages 88–97, Vancouver, Canada, August 23-27 1992.

[Wid95] Jennifer Widom. Research problems in data warehousing. In *Proceedings of the Fourth International Conference on Information and Knowledge Management (CIKM)*, 1995.

[WN95] J. L. Wiener and J. F. Naughton. Oodb bulk loading revisited: The partitioned-list approach. In Dayal et al. [DGN95], pages 30–41.

[YKL97] J. Yang, K. Karlapalem, and Q. Li. Algorithms for materialized view design in a data warehousing environment. In Jarke et al. [JCD+97], pages 136–145.

[YL87] H. Yang and P.-A. Larson. Query transformation for PSJ-queries. In Stocker and Kent [SK87], pages 245–254.

[Zan86] Carlo Zaniolo, editor. *Proceedings of ACM SIGMOD 1986 International Conference on Management of Data*, Washington, D.C., May 28-30 1986.

[ZGMHW95] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In Carey and Schneider [CS95], pages 316–327.