View Maintenance in a Warehousing Environment[*]

Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, Jennifer Widom

Computer Science Department
Stanford University
Stanford, CA 94305-2140, USA
{zhuge,hector,joachim,widom}@cs.stanford.edu

Abstract

A warehouse is a repository of integrated information drawn from remote data sources. Since a warehouse effectively implements materialized views, we must maintain the views as the data sources are updated. This view maintenance problem differs from the traditional one in that the view definition and the base data are now decoupled. We show that this decoupling can result in anomalies if traditional algorithms are applied. We introduce a new algorithm, ECA (for "Eager Compensating Algorithm"), that eliminates the anomalies. ECA is based on previous incremental view maintenance algorithms, but extra "compensating" queries are used to eliminate anomalies. We also introduce two streamlined versions of ECA for special cases of views and updates, and we present an initial performance study that compares ECA to a view recomputation algorithm in terms of messages transmitted, data transferred, and I/O costs.

1 Introduction

Warehousing is an emerging technique for retrieval and integration of data from distributed, autonomous, possibly heterogeneous, information sources. A *data warehouse* is a repository of integrated information, available for queries and analysis (e.g., decision support, or data mining) [IK93]. As relevant information becomes available from a source, or when relevant information is modified, the information is extracted from the source, translated into a common model (e.g., the relational model), and integrated with existing data at the warehouse. Queries can be answered and analysis can be performed quickly and efficiently since the integrated information is directly available at the warehouse, with differences already resolved.

1.1 The Problem

One can think of a data warehouse as defining and storing integrated *materialized views* over the data from multiple, autonomous information sources. An important issue is the prompt and correct propagation of updates at the sources to the views at the warehouse. Numerous methods have been developed for materialized view maintenance in conventional database systems; these methods are discussed in Section 2.

Unfortunately, existing materialized view maintenance algorithms fail in a warehousing environment. Existing approaches assume that each source understands view management and knows the relevant view definitions. Thus, when an update occurs at a source, the source knows exactly what data is needed for updating the view.

However, in a warehousing environment, the sources can be legacy or unsophisticated systems that do not understand views. Sources can inform the warehouse when an update occurs, e.g., a new employee has been hired, or a patient has paid her bill. However, they cannot determine what additional data may or may not be necessary for incorporating the update into the warehouse views. When the simple update information arrives at the warehouse, we may discover that some additional source data is necessary to update the views. Thus, the warehouse may have to issue queries to some of the sources, as illustrated in Figure 1.1. The queries are evaluated at the sources later than the corresponding updates, so the source states may have changed. This decoupling between the base data on the one hand (at the sources), and the view definition and view maintenance machinery on the other (at the warehouse), can lead the warehouse to compute incorrect views.

We illustrate the problems with three examples. For these examples, and for the rest of this paper, we use the relational model for data and relational algebra select-project-join queries for views. Although we are using relational algebra, we assume that duplicates are retained in the materialized views. Duplicate retention (or at least a replication count) is essential if deletions are to be handled incrementally [BLT86, GMS93]. Note that

[*]This work was supported by ARPA Contract F33615-93-1-1339, by the Anderson Faculty Scholar Fund, by the Center for Integrated Systems at Stanford University, and by equipment grants from Digital Equipment Corporation and IBM Corporation. The US Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of the US Government.

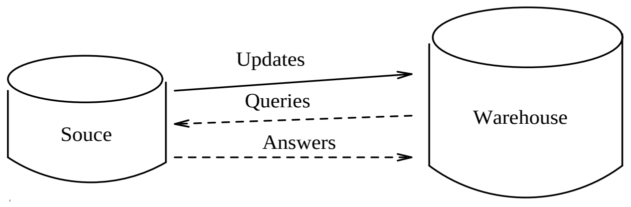

Figure 1.1: Update processing in a single source model

the type of solution we propose here can be extended to other data models and view specification languages.

Also, in the examples and in this paper we focus on a single source, and a single view over several base relations. Our methods extend to multiple views directly. Handling a view that spans several sources requires the same type of solution, but introduces additional issues; see Section 7 for a brief discussion.

Example 1: Correct View Maintenance

Suppose our source contains two base relations r_1 and r_2 as follows:

r_1 :

W	X
1	2

r_2 :

X	Y
2	4

The view at the warehouse is defined by the expression: $V = \Pi_W(r_1 \bowtie r_2)$. Initially the materialized view at the warehouse, MV, contains the single tuple [1]. Now suppose tuple [2,3] is inserted into r_2 at the source. For notation, we use $insert(r,t)$ to denote the insertion of tuple t into relation r (similarly for $delete(r,t)$), and we use $([t_1],[t_2],\ldots,[t_n])$ to denote a relation with tuples $t_1,t_2,\ldots,t_n$. The following events occur:

1. Update $U_1 = insert(r_2,[2,3])$ occurs at the source. Since the source does not know the details or contents of the view managed by the warehouse, it simply sends a notification to the warehouse that update U_1 occurred.
2. The warehouse receives U_1. Applying an incremental view maintenance algorithm, it sends query $Q_1 = \Pi_W(r_1 \bowtie [2,3])$ to the source. (That is, the warehouse asks the source which r_1 tuples match with the new [2,3] tuple in r_2.)
3. The source receives Q_1 and evaluates it using the current base relations. It returns the answer relation $A_1 = ([1])$ to the warehouse.
4. The warehouse receives answer A_1 and adds ([1]) to the materialized view, obtaining ([1],[1]).

The final view at the warehouse is correct, i.e., it is equivalent to what one would obtain using a conventional view maintenance algorithm directly at the source.[1] The next two examples show how the final view can be incorrect.

[1]As stated earlier, for incremental handling of deletions we need to keep both [1] tuples in the view. For instance, if [2,4] is later deleted from r_2, one (but not both) of the [1] tuples should be deleted from the view.

Example 2: A View Maintenance Anomaly

Assume we have the same relations as in Example 1, but initially r_2 is empty:

r_1 :

W	X
1	2

r_2 :

X	Y

Consider the same view definition as in Example 1: $V = \Pi_W(r_1 \bowtie r_2)$, and now suppose there are two consecutive updates: $U_1 = insert(r_2,[2,3])$ and $U_2 = insert(r_1,[4,2])$. The following events occur. Note that initially $MV = \emptyset$.

1. The source executes $U_1 = insert(r_2,[2,3])$ and sends U_1 to the warehouse.
2. The warehouse receives U_1 and sends $Q_1 = \Pi_W(r_1 \bowtie [2,3])$ to the source.
3. The source executes $U_2 = insert(r_1,[4,2])$ and sends U_2 to the warehouse.
4. The warehouse receives U_2 and sends $Q_2 = \Pi_W([4,2] \bowtie r_2)$ to the source.
5. The source receives Q_1 and evaluates it on the current base relations: $r_1 = ([1,2],[4,2])$ and $r_2 = ([2,3])$. The resulting answer relation is $A_1 = ([1],[4])$, which is sent to the warehouse.
6. The warehouse receives A_1 and updates the view to $MV \cup A_1 = ([1],[4])$.
7. The source receives Q_2 and evaluates it on the current base relations r_1 and r_2. The resulting answer relation is $A_2 = ([4])$, which is sent to the warehouse.
8. The warehouse receives answer A_2 and updates the view to $MV \cup A_2 = ([1],[4],[4])$.

If the view had been maintained using a conventional algorithm directly at the source, then it would be ([1]) after U_1 and ([1],[4]) after U_2. However, the warehouse first changes the view to ([1],[4]) (in step 6) and to ([1],[4],[4]) (in step 8), which is an incorrect final state. The problem is that the query Q_1 issued in step 4 is evaluated using a state of the base relations that differs from the state at the time of the update (U_1) that caused Q_1 to be issued. □

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

Example 3: Deletion Anomaly

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

r_1 :

W	X
1	2

r_2 :

X	Y
2	3

Suppose that the view definition is $V = \Pi_{W,Y}(r_1 \bowtie r_2)$. The following events occur. Note that initially $MV = ([1,3])$.

1. The source executes $U_1 = delete(r_1, [1,2])$ and notifies the warehouse.
2. The warehouse receives U_1 and emits $Q_1 = \Pi_{W,Y}([1,2] \bowtie r_2)$.
3. The source executes $U_2 = delete(r_2, [2,3])$ and notifies the warehouse.
4. The warehouse receives U_2 and emits $Q_2 = \Pi_{W,Y}(r_1 \bowtie [2,3])$.
5. The source receives Q_1. The answer it returns is $A_1 = \emptyset$ since both relations are now empty.
6. The warehouse receives A_1 and replaces the view by $MV - A_1 = ([1,3])$. (Difference is used here since the update to the base relation was a deletion [BLT86].)
7. Similarly, the source evaluates Q_2, returns $A_2 = \emptyset$, and the warehouse replaces the view by $MV - A_2 = ([1,3])$.

This final view is incorrect: since r_1 and r_2 are empty, the view should be empty too. □

1.2 Possible Solutions

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

- *Recompute the view* (RV). The warehouse can either recompute the full view whenever an update occurs at the source, or it can recompute the view periodically. In Example 2, if the warehouse sends a query to the source to recompute the view after it receives U_2, then the source will compute the answer relation $A = ([1],[4])$ (assuming no further updates) and the warehouse will correctly set $MV = ([1],[4])$. Recomputing views is usually time and resource consuming, particularly in a distributed environment where a large amount of data might need to be transferred from the source to the warehouse. In Section 6 we compare the performance of our proposed solution to this one.
- *Store at the warehouse copies of all relations involved in views* (SC). In Example 2, suppose that the warehouse keeps up-to-date copies of r_1 and r_2. When U_1 arrives, Q_1 can be evaluated "locally," and no anomaly arises. The disadvantages of this approach are: (1) the warehouse needs to store copies of all base relations used in its views, and (2) copied relations at the warehouse need to be updated whenever an update occurs at the source.
- *The Eager Compensating Algorithm* (ECA). The solution we advocate avoids the overhead of recomputing the view or of storing copies of base relations. The basic idea is to add to queries sent to the source *compensating queries* to offset the effect of concurrent updates. For instance, in Example 2, consider the receipt of $U_2 = insert(r_1, [4,2])$ in step 4. If we assume that messages are delivered in order, and that the source handles requests atomically, then when the warehouse receives U_2 it can deduce that its previous query Q_1 will be evaluated in an "incorrect" state—Q_1 will see the [4,2] tuple of the second insert. (Otherwise, the warehouse would have received the answer to Q_1 before it received the notification of U_2.) To compensate, the warehouse sends query:

$$Q_2 = \Pi_W([4,2] \bowtie r_2) - \Pi_W([4,2] \bowtie [2,3])$$

The first part of Q_2 is as before; the second part compensates for the extra tuple that Q_1 will see. We call this an "eager" algorithm because the warehouse is compensating (in step 4) even before the answer for Q_1 has arrived (in step 6). In Section 5.2 we briefly discuss a "lazy" version of this approach. Continuing with the example, we see that indeed the answer received in step 6, $A_1 = ([1],[4])$, contains the extra tuple [4]. But, because of the compensation, the A_2 answer received in step 8 is empty, and the final view is correct. In Section 5.2 we present the Eager Compensating Algorithm in detail, showing how the compensating queries are determined for an arbitrary view, and how query answers are integrated into the view.

We also consider two improvements to the basic ECA algorithm:

- *The ECA-Key Algorithm* (ECA^K). If a view includes a key from every base relation involved in the view, then we can streamline the ECA algorithm in two ways: (1) Deletions can be handled at the warehouse without issuing a query to the source. (2) Insertions require queries to the source, but compensating queries are unnecessary. To illustrate point (1), consider Example 3, and suppose W and Y are keys for r_1 and r_2. When the warehouse receives $U_1 = delete(r_1, [1,2])$ (step 1), it can immediately determine that all tuples of the form [1,x] are deleted from the view (where x denotes any value)—no query needs to be sent to the source. Similarly, $U_2 = delete(r_2, [2,3])$ causes all [x,3] tuples to be deleted from the view, without querying the source. The final empty view is correct. Section 5.4 provides an example illustrating point (2), and a description of ECA^K. Note that ECA^K does have the disadvantage that it can only be used for a subset of all possible views—those that contain keys for all base relations.

- *The ECA-Local Algorithm* (ECA^L). In ECA^K, the warehouse processes deletes locally, without sending any queries to the source, but inserts still require queries to be sent to the source. Generalizing this idea, for a given view definition and a given update, it is possible to determine whether or not the update can be handled locally; see, e.g., [GB94,BLT86]. We outline ECA^L, which combines local handling of updates with the compensation approach for maintenance of arbitrary views.

1.3 Outline of Paper

In the next section we briefly review related research. Then, in Section 3, we provide a formal definition of correctness for view maintenance in a warehousing environment. As we will see, there are a variety of "levels" of correctness that may be of interest to different applications. In Sections 4 and 5 we present our new algorithms, along with a number of examples. In Section 6 we compare the performance of our ECA algorithm to the view recomputation approach. In Section 7 we conclude and discuss future directions of our work. Additional details—additional examples, proofs, analyses, etc.—that are too lengthy and intricate to be included in the body of the paper are presented in [ZGMHW94] (available via anonymous ftp from host `db.stanford.edu`).

2 Related Research

Many incremental view maintenance algorithms have been developed. Most of them are designed for a traditional, centralized database environment, where it is assumed that view maintenance is performed by a system that has full control and knowledge of the view definition, the base relations, the updated tuples, and the view [HD92,QW91,SI84]. These algorithms differ somewhat in the view definitions they handle. For example, [BLT86] considers select-project-join (SPJ) views only, while algorithms in [GMS93] handle views defined by any SQL or Datalog expression. Some algorithms depend on key information to deal with duplicate tuples [CW91], while others use a counting approach [GMS93].

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

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

The algorithms in this paper could be viewed as a specialized concurrency control mechanism for a multidatabase system. Our algorithms exploit the semantics of the application (relational views) to provide a certain type of consistency without traditional locking. Paper [BGMS92] provides a survey of related work.

3 Correctness

Our first task is to define what correctness means in an environment where activity at the source is decoupled from the view at the warehouse. We start by defining the notion of *events*. In our context, an event corresponds to a sequence of operations performed at the same site. There are two types of events at the source:

1. S_up: the source executes an update U, then sends an update notification to the warehouse.
2. S_qu: the source evaluates a query Q using its current base relations, then sends the answer relation A back to the warehouse.

There are two types of events at the warehouse:

1. W_up: the warehouse receives an update U, generates a query Q, and sends Q to the source for evaluation.
2. W_ans: the warehouse receives the answer relation A for a query Q and updates the view based on A.

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

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

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

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

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

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

3.1 Levels of Correctness

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

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

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

- *Convergence*: For all finite executions, $V[ws_q] = V[ss_p]$. That is, after the last update and after all activity has ceased, the view is consistent with the relations at the source.
- *Weak Consistency:* For all executions and for all ws_i, there exists an ss_j such that $V[ws_i] = V[ss_j]$. That is, every state of the view corresponds to some valid state of the relations at the source.
- *Consistency:* For all executions and for every pair $ws_i < ws_j$, there exist $ss_k \leq ss_l$ such that $V[ws_i] = V[ss_k]$ and $V[ws_j] = V[ss_l]$. That is, every state of the view corresponds to a valid source state, and in a corresponding order.
- *Strong consistency:* Consistency and convergence.
- *Completeness:* Strong consistency, and for each ss_i there exists a ws_j such that $V[ws_j] = V[ss_i]$. That is, there is a complete order-preserving mapping between the states of the view and the states of the source.

Although completeness is a nice property since it states that the view "tracks" the base data exactly, we believe it is too strong a requirement and unnecessary in most practical warehousing scenarios. In some cases, convergence may be sufficient, i.e., knowing that "eventually" the warehouse will have a valid state, even if it passes through intermediate states that are invalid. In other cases, consistency (weak or not) may be required, i.e., knowing that every warehouse state is valid with respect to some source state. Examples 2 and 3 showed that a straightforward incremental view maintenance algorithm is not even weakly consistent in the environments we consider. We will show that our Eager Compensating Algorithm is strongly consistent, and hence a satisfactory approach for most warehousing scenarios.

4 Views and Queries

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

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

where *proj* is a set of attribute names, *cond* is a boolean expression, and $r_1, \ldots, r_n$ are base relations. Note that any relational algebra expression constructed with select, project, and join operations can be transformed into an equivalent expression of this form. For simplicity, we assume that $r_1, \ldots, r_n$ are distinct relations. Our algorithms can be extended to allow multiple occurrences of the same relation (e.g., by handling updates to such relations once for each appearance of the relation).

4.1 Signs

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

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

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

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

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

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

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

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

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

4.2 Query Expressions

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

$$T = \Pi_{proj}(\sigma_{cond}(\tilde{r}_1 \times \tilde{r}_2 \times \ldots \times \tilde{r}_n)) \qquad (4.1)$$

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

$$Q = \sum_i T_i. \qquad (4.2)$$

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

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

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

More formally, consider any query (or view definition) of the form $Q = \sum_i T_i$ (recall Equation 4.2). Let U be an update involving relation r_k, and let $tuple(U)$ be the updated tuple. Then $Q\langle U\rangle = \sum_i T_i\langle U\rangle$, where for each T_i (recall Equation 4.1):

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

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

5 The ECA Algorithm

In this section we present the details of our *Eager Compensating Algorithm* (ECA), introduced in Section 1 as a solution to the anomaly problem. ECA is an incremental view maintenance algorithm based on the centralized view maintenance algorithm described in [BLT86]. ECA anticipates the anomalies that arise due to the decoupling between base relation updates and view modification, and ECA compensates for the anomalies as needed to ensure correct view maintenance. Before we present ECA and its extensions, we first review the original incremental view maintenance algorithm.

5.1 The Basic Algorithm

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

Algorithm 5.1 (Basic Algorithm)

```
At the source:
• S_up_i: execute $U_i$;
          send $U_i$ to the warehouse;
          trigger event $W\_up_i$ at the warehouse
• S_qu_i: receive query $Q_i$;
          let $A_i = Q_i[ss_i]$;  ($ss_i$ is current source state)
          send $A_i$ to the warehouse;
          trigger event $W\_ans_i$ at the warehouse
At the warehouse:
• W_up_i: receive update $U_i$;
          let $Q_i = V\langle U_i\rangle$;
          send $Q_i$ to the source;
          trigger event $S\_qu_i$ at the source
• W_ans_i: receive $A_i$;
           update view: $MV = MV + A_i$
```

(end algorithm)

Notice that each update at the source triggers an *S_up* event, which then triggers a *W_up* event at the warehouse, triggering an *S_qu* event back at the source, and finally a *W_ans* event at the warehouse (recall Figure 1.1). As shown in Examples 2 and 3, this algorithm may lead to anomalies. Consequently, using our definitions from Section 3, this basic algorithm is neither convergent nor weakly consistent in a warehousing environment.

5.2 The Eager Compensating Algorithm

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

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

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

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

Algorithm 5.2 (ECA)
`COLLECT` $= \emptyset$.
The source events behave exactly as in Algorithm 5.1. At the warehouse:

```
• $W\_up_i$: receive $U_i$;
        let $Q_i = V\langle U_i\rangle - \sum_{Q_j \in UQS} Q_j\langle U_i\rangle$
        send $Q_i$ to the source;
        trigger event $S\_qu_i$ at the source
• $W\_ans_i$: receive $A_i$;
        let COLLECT = COLLECT + $A_i$;
        if $UQS = \emptyset$
            then { $MV \leftarrow MV +$ COLLECT;
                   COLLECT $\leftarrow \emptyset$ }
            else do nothing
```

(end algorithm)

5.3 Example

The following example illustrates ECA handling three insertions to three different base relations. A number of additional ECA examples are given in [ZGMHW94].

Example 4: ECA

Consider source relations:

r_1 :

W	X
1	2

r_2 :

X	Y

r_3 :

Y	Z

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

1. Warehouse receives $U_1 = insert(r_1, [4,2])$
   Warehouse sends
   $Q_1 = V\langle U_1\rangle = \Pi_W([4,2] \bowtie r_2 \bowtie r_3)$
2. Warehouse receives $U_2 = insert(r_3, [5,3])$
   $UQS = \{Q_1\}$. Warehouse sends
   $$\begin{aligned} Q_2 &= V\langle U_2\rangle - Q_1\langle U_2\rangle \\ &= \Pi_W(r_1 \bowtie r_2 \bowtie [5,3]) - \Pi_W([4,2] \bowtie r_2 \bowtie [5,3]) \end{aligned}$$
3. Warehouse receives $U_3 = insert(r_2, [2,5])$
   $UQS = \{Q_1, Q_2\}$. Warehouse sends
   $$\begin{aligned} Q_3 &= V\langle U_3\rangle - Q_1\langle U_3\rangle - Q_2\langle U_3\rangle \\ &= \Pi_W(r_1 \bowtie [2,5] \bowtie r_3) - \Pi_W([4,2] \bowtie [2,5] \bowtie r_3) \\ &\quad - \Pi_W((r_1 - [4,2]) \bowtie [2,5] \bowtie [5,3]) \end{aligned}$$
4. Warehouse receives $A_1 = [4]$
   `COLLECT` $= \emptyset + ([4]) = ([4])$, $UQS = \{Q_2, Q_3\}$
5. Warehouse receives $A_2 = [1]$
   `COLLECT` $= ([4]) + ([1]) = ([1],[4])$, $UQS = \{Q_3\}$
6. Warehouse receives $A_3 = \emptyset$
   `COLLECT` $= ([1],[4]) + \emptyset = ([1],[4])$, $UQS = \emptyset$
   Warehouse updates view: $MV = \emptyset +$ `COLLECT` $= ([1],[4])$, which is correct. □

A complete proof showing that the algorithm is strongly consistent is given in [ZGMHW94]. Notice, however, that ECA is not complete. Recalling Section 3.1, completeness requires that every source state be reflected in some view state. Clearly some source states may be "missed" by the ECA algorithm while it collects query answers. We can modify ECA to obtain a complete algorithm that we call the *Lazy Compensating Algorithm* (LCA). For each source update, LCA waits until it has received all query answers (including compensation) for the update, then applies the changes for that update to the view. A detailed description of LCA is beyond the scope of this paper. LCA is less efficient than ECA, and we believe that strong consistency is sufficient for most environments and completeness is generally not needed. Hence, we expect that ECA will be more useful than LCA in practice.

5.4 The ECA-Key Algorithm

We can "streamline" ECA in the case where the attributes in the projection list of the view definition (recall Section 4) contain key attributes for each of the base tables. (In fact, it could be advisable to design warehouse view definitions with this property in mind, for more efficient maintenance.) The *ECA-Key Algorithm* (ECA^K) proceeds as follows:

1. `COLLECT` is initialized with the current materialized view (not the empty set). Instead of storing modifications to MV, `COLLECT` is a "working copy" of MV.
2. When a delete is received at the warehouse, no query is sent to the source. Instead, the delete is applied to `COLLECT` immediately, using a special *key-delete* operation defined below.
3. When an insert is received at the warehouse, a query is sent to the source. However, no compensating queries are added. Thus, when an insert U is received, the query sent to the source is simply $V\langle U\rangle$.
4. As answers are received, they are accumulated in the `COLLECT` set, as in the original ECA. However, duplicate tuples are not added to the `COLLECT` set. When the view contains keys for all base relations, there can be no duplicates in the view. Thus, if a duplicate occurs, it is due to an anomaly and can be ignored.
5. When UQS is empty, the materialized view is updated, replacing it with the tuples in `COLLECT`.

Example 5: ECA^K

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

r_1 :

W	X
1	2

r_2 :

X	Y
2	3

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

1. $U_1 = insert(r_2, [2,4])$
   Warehouse sends $Q_1 = V\langle U_1\rangle = \Pi_{W,Y}(r_1 \bowtie [2,4])$
2. $U_2 = insert(r_1, [3,2])$
   Warehouse sends $Q_2 = V\langle U_2\rangle = \Pi_{W,Y}([3,2] \bowtie r_2)$
3. $U_3 = delete(r_1, [1,2])$
   Operation *key-delete*(V, r_1, [1,2]) (see below) deletes tuples of the form [1,x], obtaining `COLLECT` = `COLLECT` − ([1,3]) = $\emptyset$, $UQS = \{Q_1, Q_2\}$
4. Warehouse receives $A_1 = ([3,4])$
   `COLLECT` = `COLLECT` + ([3,4]) = ([3,4]), $UQS = \{Q_2\}$
5. Warehouse receives $A_2 = ([3,3],[3,4])$
   First, A_2 is added to `COLLECT`: duplicate tuple [3,4] is not added, so `COLLECT` = ([3,3],[3,4]). Next, since $UQS = \emptyset$, MV is set to `COLLECT`, so MV = ([3,3],[3,4]). Note that `COLLECT` is not reset to empty.

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

Observe that if we had installed `COLLECT` into MV in steps 3 or 4, without waiting for $UQS = \emptyset$, then the view would have temporarily assumed an invalid state (resulting in convergence but not consistency).

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

Now consider insertions. Since insertions cause queries to be sent to the source, anomalies can still arise. However, all such anomalies result in either duplicate view tuples (which we can detect and ignore), or missing tuples that would have been deleted anyway. As illustration, suppose Q_1 of Example 5 had been executed when U_1 occurred; it would have evaluated to ([1,4]). Instead, because of the delay, we have A_1 = ([3,4]). A_1 is incorrect in two ways: (1) It contains an extra tuple [3,4] produced because Q_1 was evaluated after insert U_2. (2) It is missing tuple [1,4] because Q_1 was evaluated after delete U_3. However, both of these problems are resolved at the warehouse: (1) The extra tuple [3,4] is identified as a duplicate when A_2 is received. (2) The missing tuple [1,4] would have been deleted by U_3. Details and a sketch for the ECA^K correctness proof appear in [ZGMHW94].

5.5 The ECA-Local Algorithm

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

In ECA^L, each update is handled either locally or nonlocally. A number of papers, e.g., [BLT86,GB94,TB88], describe conditions when, for a particular view definition and a particular base relation update, the view can be updated without further access to base relations (i.e., the view is *autonomously computable*, using the terminology of [BLT86]). These results can be used to identify which updates ECA^L will process locally. For updates that cannot be processed locally, ECA is used (assuming the key condition for ECA^K does not hold), with compensating queries whenever necessary.

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

5.6 Properties of the ECA family

ECA, and its extensions ECA^K and ECA^L, have the following desirable properties:

1. They are incremental, meaning that they update the warehouse based on updates to the source, rather than recomputing the complete warehouse view from scratch.
2. They do not place any additional burden on the sources (e.g., timestamps, locks, etc.).
3. When the update frequency is low, i.e., when the answer to a warehouse query comes back before the next update occurs at the source, then the ECA algorithms behave exactly like the original incremental view maintenance algorithm. (Compensating queries are used only when the answer to a query has not been received before the next update occurs at the source.)

6 Performance Evaluation

In Section 1.2 we outlined several strategies for view maintenance in a warehousing environment: recomputing the view (RV), storing copies of all base relations (SC), our Eager Compensating Algorithm (ECA), and our extensions ECA-Key (ECA^K) and ECA-Local (ECA^L). All of these approaches provide strong consistency (recall Section 3.1). Thus, a natural issue to explore is the relative performance of these strategies. In this paper we evaluate only the basic algorithms, RV and ECA. From the performance perspective, ECA^K is simply an enhancement to ECA that eliminates querying the source in certain cases. Since there is very little additional overhead in ECA^K, ECA^K should certainly be used when it is possible to arrange for warehouse views to include all base relation keys. Storing copies of base relations (SC) can be seen as an enhancement to any of our algorithms, requiring an "orthogonal" performance comparison (based on warehouse storage costs, etc.) that is beyond the scope of this paper. As discussed in Section 5.5, ECA^L requires complex processing at the warehouse, the measurement of which falls outside the scope of our performance evaluation. We plan to address performance issues for SC and ECA^L, along with a more extensive evaluation of ECA, as future work.

When we intuitively compare RV and ECA, it seems that ECA should certainly outperform RV, since ECA is an incremental update algorithm while RV recomputes the view from scratch. However, ECA may need to send many more queries to the source than RV. In addition, ECA's queries grow in complexity as compensations upon compensations are added. Hence, we seek to determine when it is more effective to recompute the entire view, rather than maintaining it incrementally with the associated extra activity.

To answer this question, we provide an initial performance analysis of RV and ECA. We note that this is not a comprehensive analysis, as there are a number of parameters we have not fully studied (ranging from the number of relations, to the exact sequence and timing of the updates, to the way queries are optimized at the source). Rather, we have selected a "representative" scenario that serves to illustrate the performance tradeoffs.

For the analysis, we focus on three separate cost factors: M, the number of messages sent between the source and warehouse, B, the number of bytes transferred from the source to the warehouse, and IO, the number of I/O's performed at the source. In RV and ECA, identical update notification messages are sent to the warehouse, so these costs are not included in our calculations. Throughout this section, we use the variables listed in Table 1, shown with their default values. The RV algorithm is described informally in Section 1.2; a formal specification is provided in [ZGMHW94].

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

Table 1: List of variables.

6.1 Performance Based on Number of Messages

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

Thus, in the situation least favorable for ECA, ECA sends $2k$ messages while RV sends 2 messages. Of course, the price of the RV approach in this case is that the state

[2]Because ECA uses signed queries (recall Section 4.1), and some sources—such as an SQL server—may need to handle the positive and negative parts of such queries separately, we may need to send a pair of queries for some updates instead of a single query. We assume the pair of queries is "packaged" as one message, and the pair of answers also is returned in a single message.

of the warehouse view lags well behind the state of the base relations. In the most favorable situation for ECA, ECA and RV both send $2k$ messages.

6.2 Performance Based on Data Transferred

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

Example 6: Example warehouse scenario

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

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

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

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

We make the following assumptions in our analysis:

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

Not surprisingly, for RV the fewest bytes are transferred (B_{RVBest}) when the view is recomputed only once, after U_3 has occurred. The worst case ($B_{RVWorst}$) is when the view is recomputed after each update. For ECA, the best case ($B_{ECABest}$) is when no compensating queries are needed, i.e., the updates are sufficiently spaced so that each query is processed before the next update occurs at the source. Note that in this case, ECA performs as efficiently as the original incremental algorithm (Algorithm 5.1). The worst case for ECA ($B_{ECAWorst}$) is when all updates occur before the first query arrives at the source. Intuitively, the difference between the best and worst cases of ECA represents the "compensation cost."

The calculations for analyzing our algorithms are rather complex and therefore omitted here; the complete derivations can be found in [ZGMHW94]. In particular, the expressions derived in [ZGMHW94] for the number of bytes transferred are:

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

Figure 6.2 shows the number B of bytes transferred as a function of the cardinality C. (In all of our graphs, parameters have the default values of Table 1 unless otherwise indicated.) Best and worst cases are shown for both algorithms. Thus, for each algorithm, actual performance will be somewhere in between the best and worst case curves, depending on the timing of update arrivals (for ECA) and the frequency of view recomputation (for RV).

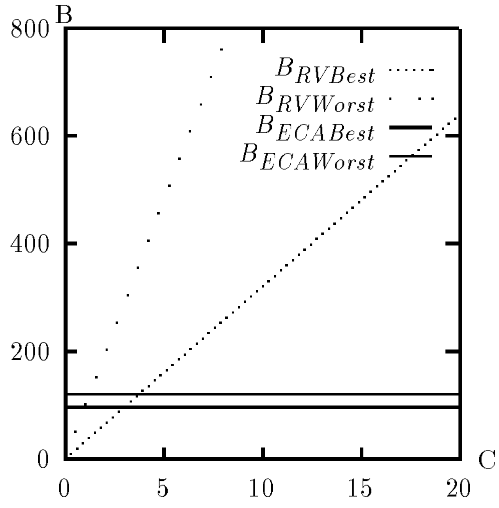

Figure 6.2: B versus C

From Figure 6.2 we see that in spite of the compensating queries, ECA is much more efficient than RV (in terms of data transferred), unless the relations involved are extremely small (less than approximately 5 tuples each). This result continues to hold over wide ranges of the join selectivity J, except if J is very small (see equations above).

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

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

Figure 6.3 shows the number of bytes transferred as a function of k when $C = 100$. As expected, there is a crossover point beyond which recomputing the view once (RV's best case) is superior to even the best case

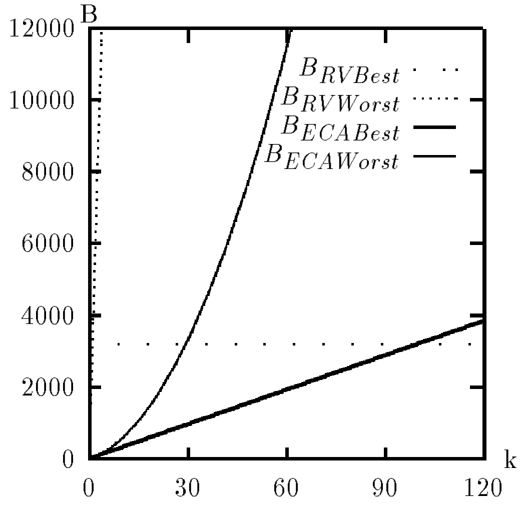

Figure 6.3: B versus k

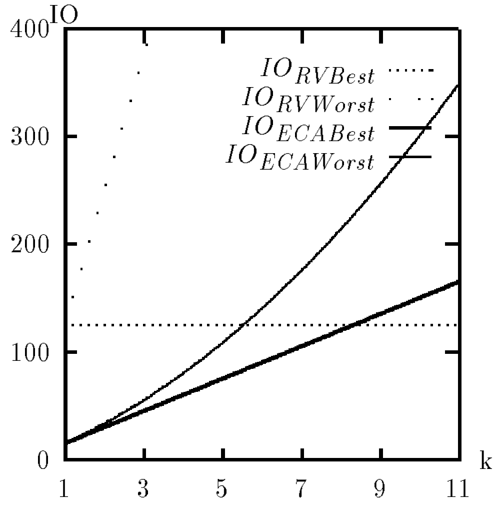

Figure 6.4: IO versus k, Scenario 2

for ECA. For our example, this crossover is at 100 updates. In the ECA worst case, when all updates occur at the source before any of the warehouse queries arrive, each warehouse query must compensate every preceding update. This behavior results in ECA transmitting additional data that is quadratic on the number of updates. Hence, in the situation least favorable for ECA, RV outperforms ECA when 30 or more updates are involved. Bear in mind that this situation occurs only if all updates precede all queries. If updates and queries are interleaved at the source, then performance will be somewhere between the ECA best and worst cases, and the crossover point will be somewhere between 30 and 100 updates.

Also notice that Figure 6.3 is for relatively small relations ($C = 100$); for larger cardinalities the crossover points will be at larger number of updates. Finally, note that the RV best case we have been comparing against assumes the view is recomputed once, no matter how many updates occur. If RV recomputes the view more frequently (such as once per some number of updates), then its cost will be substantially higher. In particular, $B_{RVWorst}$ is very expensive and always substantially worst than $B_{ECAWorst}$.

6.3 Performance Based on I/O

Estimating the number of I/O's performed at the source is similar to estimating the number of bytes transferred. Details of the estimation are discussed in [ZGMHW94]. We consider two extreme scenarios there: when indexing is used and ample memory is available, and when memory is very limited and there are no indexes. Studying these extremes lets us discover the conditions that are most favorable for the algorithms we consider. Due to the space limitations, we only present one graph to illustrate the type of results obtained. Figure 6.4 gives the number of I/O's as a function of the number of updates for the second scenario studied (limited memory and no indexes).

The shape of the curves in Figure 6.4 is similar to those in Figure 6.3, and thus our conclusions for I/O costs are similar to those for data transmission. The main difference is that the crossover points occur with smaller update sequences: $5 < k < 8$ in this case, as opposed to a crossover between $k = 30$ and $k = 100$ when data transfer is the metric. Intuitively, this means that ECA is not as effective at reducing I/O costs as it is at reducing data transfer. However, ECA can still reduce I/O costs over RV significantly, especially if relations are larger than the 100 tuples considered for these figures. Also, we expect that the I/O performance of ECA would improve if we incorporated multiple term optimization or caching into the analysis.

As a final note, we remind the reader that our results are for a particular three-relation view. In spite of this, we believe that our results are indicative of the performance issues faced in choosing between RV and ECA. Our results indicate that when the view involves more relations, ECA should still generally outperform RV.

7 Conclusion

Data warehousing is an emerging (and already very popular) technique used in many applications for retrieval and integration of data from autonomous information sources. However, warehousing typically is implemented in an ad-hoc way. We have shown that standard algorithms for maintaining a materialized view at a warehouse can lead to anomalies and inconsistent modifications to the view. The anomalies are due to the fact that view maintenance at the warehouse is decoupled from updates at the data sources, and we cannot expect the data sources to perform sophisticated functions in support of view management. Consequently, previously proposed view maintenance algorithms cannot be used in this environment.

We have presented a new algorithm, and outlined two extensions, for correctly maintaining materialized views in a warehousing environment. Our Eager Compensating Algorithm, ECA, and its streamlined versions, ECA^K and ECA^L, are all strongly consistent, meaning that the warehouse data always corresponds to a meaningful state of the source data. An initial performance study analyzing three different cost factors (messages, data traffic, and I/O) suggests that, except for very small relations, ECA is consistently more efficient than periodically recomputing the warehouse view from scratch.

Although in this paper we have addressed a restricted warehousing environment with only one source and one view, ECA can readily be adapted to more general scenarios. For example, in a warehouse consisting of multiple views where each view is over data from a single source, ECA is simply applied to each view separately.

In the future we plan to address the following additional issues.

- We will consider how ECA (and its extensions) can be adapted to views over multiple sources. Many aspects of the anomaly problem remain the same. However, additional issues are raised because warehouse queries (both regular queries and compensating queries) must be fragmented for execution at multiple sources. While fragmenting itself does not pose a novel problem (at least in the straightforward relational case), coordinating the query results and the necessary compensations for anomaly-causing updates may require some intricate algorithms.
- We will consider how ECA can be extended to handle a set of updates at once, rather than one update at a time. Since we expect that in practice many source updates will be "batched," this extension should result in a very useful performance enhancement.
- We will modify the algorithms to handle views defined by more complex relational algebra expressions (e.g., using union and/or difference) as well as other relational query languages (e.g., SQL or Datalog).
- We will explore how the algorithms can be adapted to other data models (e.g., an object-based data model).

References

[BGMS92] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of multidatabase transaction management. *VLDB Journal*, 1(2):181–239, October 1992.

[BLT86] J.A. Blakeley, P.-A. Larson, and F.W. Tompa. Efficiently updating materialized views. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 61–71, Washington, D.C., June 1986.

[CW91] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, pages 577–589, Barcelona, Spain, September 1991.

[GB94] Ashish Gupta and J. A. Blakeley. Updating materialized views using the view contents and the update. In *unpublished document*, 1994.

[GMS93] A. Gupta, I. Mumick, and V. Subrahmanian. Maintaining views incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 157–166, Washington, D.C., May 1993.

[Han87] E.N. Hanson. A performance analysis of view materialization strategies. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 440–453, 1987.

[HD92] J.V. Harrison and S.W. Dietrich. Maintenance of materialized views in a deductive database: An update propagation approach. In *Proceedings of the 1992 JICLSP Workshop on Deductive Databases*, pages 56–65, 1992.

[IK93] W.H. Inmon and C. Kelley. *Rdb/VMS: Developing the Data Warehouse.* QED Publishing Group, Boston, London, Toronto, 1993.

[LHM+86] B. Lindsay, L.M. Haas, C. Mohan, H. Pirahesh, and P. Wilms. A snapshot differential refresh algorithm. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Washington, D.C., May 1986.

[QW91] X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *IEEE Transactions on Knowledge and Data Engineering*, 3(3):337–341, September 1991.

[RK86] N. Roussopoulos and H. Kang. Preliminary design of ADMS+-: A workstation-mainframe integrated architecture for database management systems. In *Proceedings of the Twelfth International Conference on Very Large Data Bases*, pages 355–364, Kyoto, Japan, August 1986.

[SF90] A. Segev and W. Fang. Currency-based updates to distributed materialized views. In *Proceedings of the Sixth International Conference on Data Engineering*, pages 512–520, Los Alamitos, February 1990.

[SF91] A. Segev and W. Fang. Optimal update policies for distributed materialized views. *Management Science*, 37(7):851–70, July 1991.

[SI84] O. Shmueli and A. Itai. Maintenance of views. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 240–255, Boston, Massachusetts, May 1984.

[SP89a] A. Segev and J. Park. Maintaining materialized views in distributed databases. In *Proceedings of the Fifth International Conference on Data Engineering*, pages 262–70, Los Angeles, February 1989.

[SP89b] A. Segev and J. Park. Updating distributed materialized views. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):173–184, June 1989.

[TB88] F.W. Tompa and J.A. Blakeley. Maintaining materialized views without accessing base data. *Information Systems*, 13(4):393–406, 1988.

[ZGMHW94] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. Technical report, Stanford University, October 1994. Available via anonymous ftp from host `db.stanford.edu` as `pub/zhuge/1994/anomaly-full.ps`.