Algorithms for Deferred View Maintenance

Latha S. Colby
Bell Laboratories
colby@bell-labs.com

Timothy Griffin
Bell Laboratories
griffin@bell-labs.com

Leonid Libkin
Bell Laboratories
libkin@bell-labs.com

Inderpal Singh Mumick
Bell Laboratories
mumick@bell-labs.com

Howard Trickey
Bell Laboratories
howard@bell-labs.com

Abstract

Materialized views and view maintenance are important for data warehouses, retailing, banking, and billing applications. We consider two related view maintenance problems: 1) how to maintain views after the base tables have already been modified, and 2) how to minimize the time for which the view is inaccessible during maintenance.

Typically, a view is maintained *immediately*, as a part of the transaction that updates the base tables. Immediate maintenance imposes a significant overhead on update transactions that cannot be tolerated in many applications. In contrast, *deferred* maintenance allows a view to become inconsistent with its definition. A *refresh* operation is used to reestablish consistency. We present new algorithms to incrementally refresh a view during deferred maintenance. Our algorithms avoid a *state bug* that has artificially limited techniques previously used for deferred maintenance.

Incremental deferred view maintenance requires auxiliary tables that contain information recorded since the last view refresh. We present three scenarios for the use of auxiliary tables and show how these impact per-transaction overhead and view refresh time. Each scenario is described by an invariant that is required to hold in all database states. We then show that, with the proper choice of auxiliary tables, it is possible to lower both per-transaction overhead and view refresh time.

1 Introduction

Interest in materialized views has increased in recent years [GM96], primarily due to the expanding range of their applications [GM95]. Most of the research on materialized views has focussed on techniques for incrementally updating materialized views when the base tables used to derive the views are updated [BLT86, CW91, GL95, GMS93, Han87, QW91, RK86, SI84, SP89].

Maintenance of a view may involve several steps, one of which brings the view table up-to-date. We call this step *refresh*. There may be other steps involved in the process of maintaining a view. For example, it may be necessary to maintain auxiliary tables that store the history of updates to the base tables. View maintenance techniques depend on *when* the view is refreshed. A view can be refreshed within the transaction that updates the base tables, or the refresh can be delayed. The former case is referred to as *immediate* view maintenance, while the latter is called *deferred* view maintenance. Deferred maintenance may be done periodically or on-demand when certain conditions arise. In the past, the term deferred maintenance has sometimes been used for on-demand maintenance.

Most of the work on view maintenance has involved the immediate case [BLT86, CW91, GL95, QW91]. The immediate maintenance approach has the disadvantage that each update transaction incurs the overhead of updating the view. The overhead increases with the number of views and their complexity.

In some applications, immediate view maintenance is simply not possible. For example, in a data warehousing system, if a component database does not know what views exist at the warehouse, it cannot modify transactions updating base tables so that they also refresh materialized views. Even in a centralized system where all the views are known, it may be necessary to minimize the per-transaction overhead imposed by view maintenance. In such cases, deferred maintenance is most appropriate.

Other applications may have a certain tolerance for out-of-date data, or even require that the view be frozen for analysis and other functions [AL80]. In this case, the view could be refreshed periodically or just before querying. Deferred maintenance also allows several updates to be batched together.

This paper contributes to the work on deferred view maintenance by presenting solutions to the following problems.

Minimize View Downtime By *downtime* we mean the execution time required by the transaction that refreshes the view table. While the view is being refreshed, an exclusive write lock is typically held over the view, and all queries and scans against the view are disallowed. Therefore, we would like to do maintenance in a manner that minimizes the time for which access to the view is blocked (during refresh), and at the same time minimizes the overhead on update transactions.

Avoid the State Bug Incremental view maintenance is typically based on "incremental queries" that avoid the need to recompute a materialized view from scratch. These queries use the updates made to base tables to compute changes that can be directly applied to a materialized view table to bring it up-to-date. Such queries can be evaluated in one of two states: the pre-update state where the base-table changes have not yet been applied, or the post-update state where the base-table changes have been applied. Most of the algorithms for view maintenance assume that the incremental queries are evaluated in the pre-update state. In the deferred case, since the base tables have already been modified, the pre-update algorithms are not directly applicable. In fact, direct application of pre-update algorithms in the post-update state can result in incorrect answers, a fact we call the *state bug.* The state bug can be avoided by severely restricting the class of updates and views considered. However, such restrictions limit the scope of deferred maintenance techniques. What is required is a general post-update algorithm that avoids the state bug and allows a large class of updates and views.

In the rest of the introduction, we elaborate on the above points and outline the contributions of the paper.

1.1 Minimizing view downtime

Consider the following example, patterned after a real application at a large retailing company.

Example 1.1 Point-of-sale information is collected in a sales table, and a customer table is used to keep records pertaining to customers. The sales table can be very large and can contain duplicates.

sales(custId, itemNo, quantity, salesPrice)
customer(custId, name, address, score)

```
CREATE VIEW V (custId, name, score, itemNo, quantity) AS
    SELECT c.custId, c.name, c.score, s.itemNo,
           s.quantity
    FROM customer c, sales s
    WHERE c.custId = s.custId AND
          s.quantity != 0 AND
          c.score = "High" .
```

Suppose that insertions into the sales table are made continuously. The view V, defined above, uses a join of these tables to compute sales made to highly valued customers. (In practice, views with aggregation are more likely. For simplicity, we omit aggregation since it is orthogonal to the problems that we discuss.) Suppose further that this view is materialized in a table MV, and that it is refreshed once every 24 hours. Between refreshes, the table MV is used by decision support applications for market analysis. If we assume that the entire view is write-locked during refresh, then it is important to minimize this view downtime. □

Contribution 1: We define consistency for databases that support deferred view maintenance in terms of invariants that describe relationships between base tables, materialized views, and auxiliary tables. Solutions to the deferred view maintenance problem are algorithms for extending user transactions with auxiliary operations needed to maintain the view invariants, and additional operations for refreshing materialized view tables. We present three such invariants together with associated algorithms for deferred view maintenance. Each solution can accommodate various update policies, and we present policies that differ in their impact on refresh times and update transaction overhead. One of these policies provides for minimal view downtime while also minimizing the overhead on update transactions.

1.2 Avoiding the state bug

We illustrate the state bug by applying the algorithm of [BLT86] in both pre- and post-update states.

Example 1.2 Let us suppose that we have a view U defined as follows and materialized in table MU (we assume SQL duplicate semantics).

```
CREATE VIEW U (A) AS
    SELECT R.A
    FROM R, S
    WHERE R.B = S.B .
```

Suppose that the contents of R, S and MU are as shown below.

R:

A	B
a_1	b_1

S:

B	C
b_1	c_1
b_1	c_2
b_2	c_1

MU:

A
a_1
a_1

Suppose that R and S are to be updated by inserting the tuple $[a_1, b_2]$ in R and the tuple $[b_2, c_2]$ in S. We can use the algorithm of [BLT86, Han87] for calculating the incremental update to MU. The algorithm of [BLT86] is a pre-update algorithm that is based on the availability of the base tables before the update. The changes to the view are computed with the incremental query $\triangle MU$, given below (the symbols $\triangle R$ and $\triangle S$ denote bags of tuples inserted into tables R and S):

$$\pi_{R.A}(R \bowtie (\triangle S)) \cup \pi_{R.A}((\triangle R) \bowtie S) \cup \pi_{R.A}((\triangle R) \bowtie (\triangle S))$$

To be consistent with SQL semantics, we assume that all operators have multiset (bag) semantics. Using this equation, the incremental insert to MU can be calculated correctly as $\{[a_1],[a_1]\}$. Now suppose that the same equation is evaluated in a post-update state, i.e., after the tuples $[a_1,b_2]$ and $[b_2,c_2]$ have been inserted in R and S. Then $\triangle MU$ would incorrectly evaluate to $\{[a_1],[a_1],[a_1],[a_1]\}$. □

Example 1.3 We present another example that shows how the state bug can lead to wrong answers other than incorrect multiplicities. Consider a view U defined as $R - S$. Let $R = \{[a],[b],[c]\}$ and $S = \{[c],[d]\}$. In the current state, U is materialized in a table MU that contains tuples $[a]$ and $[b]$. Let t be a transaction that deletes the tuple $[b]$ from R and inserts it into S. Then the algorithms of [QW91, GL95] that extend [BLT86, Han87] to the full relational and bag algebra calculate the delete bag for the view using the following equation (the symbols ∇R, and ∇MU denote bags of tuples deleted from tables R and MU):

$$\nabla MU = (\nabla R - S) \cup (\triangle S \cap R) = (\{[b]\} - S) \cup (\{[b]\} \cap R).$$

Note that it is irrelevant which semantics (set or bag) we use as no duplicates are present in any of the tables before or after the transaction. If ∇MU is evaluated in the pre-update state, the result is $\{[b]\}$ and then MU becomes $\{[a]\}$ (which is correct). However, the same expression for ∇MU evaluated in the post-update state, *after* transaction t is applied, yields $\nabla MU = \{\}$, which means that MU is not updated and keeps the incorrect tuple $[b]$! □

In the past, the same algorithm has been used in both pre-update and post-update states. However, the state bug has been avoided either by assuming availability of pre-update base tables in the post-update state, or by considering only restricted classes of views and updates. The first approach is illustrated in [Han87], where *differential tables* are maintained on base tables that contain the *suspended* updates that have not actually been applied to the database state. One problem with this approach is that it slows down the evaluation of all queries over base tables.

As an example of the second approach, [ZG+95] investigates view maintenance in a warehousing environment. Their algorithms comprise a standard view maintenance part and a compensating part. The view maintenance part is based on the pre-update algorithm of [BLT86, Han87], but is applied in the post-update state. The state bug is not encountered since their solution imposes restrictions that require (1) updates to change only *one* table, and (2) view definitions to be SPJ queries without self-joins. Their algorithms would yield incorrect results if these restrictions were relaxed.

Other papers dealing with deferred maintenance [KR87, LH+86, SP89] have considered even smaller (select-project) classes of views. Select-project views are self-maintainable [GJM96] in the sense that such views can be maintained without looking at base tables. Consequently, the issue of pre-update state vs. post-update state of base tables is irrelevant for maintaining select-project views.

Contribution 2: Our second contribution is to derive algorithms for view maintenance in the post-update state that avoid the state bug. These algorithms work for the full multiset algebra and permit insertions and deletions to any number of tables.

Paper Outline: After introducing the notation and basic concepts in Section 2, we present, in Section 3, a framework that casts the problem of view maintenance as that of maintaining database invariants. Four different scenarios are discussed – one for the immediate update of materialized views and three variations on deferred maintenance. In Section 4, we exploit a duality between pre-and post-update states to arrive at incremental algorithms that avoid the state bug and work for a large class of updates and views. Section 5 presents algorithms for solving the three scenarios of deferred view maintenance described in Section 3. We present refresh policies that use these algorithms and solve the problem of minimizing view downtime. Related work is discussed in Section 6. All proofs can be found in the full paper [CG+96].

2 Preliminaries

2.1 The bag algebra, $\mathcal{BA}$

A bag (or multiset) X is like a set, except that multiple occurrences of elements are allowed. An element x is said to have multiplicity n in the bag X if X contains exactly n copies of x. The notation $x \in X$ means that x has multiplicity $n > 0$ in X, and $x \notin X$ means that x has multiplicity 0 in X.

A database schema is a collection of base table names $\{R_1, \ldots, R_n\}$. A *database state* is a mapping from table names $\{R_1, \ldots, R_n\}$ to finite bags of tuples. We write $R_i(s)$ to denote the value of table R_i in the state s.

Our query language will be the bag algebra of [GM93, LW93], restricted to flat bags (bags of tuples, i.e., no bag-valued attributes). Let p range over quantifier-free predicates, and A range over sets of attribute names. $\mathcal{BA}$

expressions are generated by the following grammar.

$$
\begin{array}{rcll}
Q & ::= & \phi & \text{empty bag} \\
 & | & \{x\} & \text{singleton bag} \\
 & | & R & \text{table name} \\
 & | & \sigma_p(Q) & \text{selection} \\
 & | & \Pi_A(Q) & \text{projection} \\
 & | & \epsilon(Q) & \text{duplicate elimination} \\
 & | & Q_1 \uplus Q_2 & \text{additive union} \\
 & | & Q_1 \dot{-} Q_2 & \text{monus} \\
 & | & Q_1 \times Q_2 & \text{cartesian product}
\end{array}
$$

We will use the symbols Q, Q_1, Q_2, E, and F to denote $\mathcal{BA}$ expressions, which will usually be called *queries*. If s is a database state and Q is a query, then $Q(s)$ denotes the multiset resulting from evaluating Q in the state s.

The only operation that may require explanation is monus. If x occurs n times in Q_1 and m times in Q_2, then the number of occurrences of x in $Q_1 \dot{-} Q_2$ is the maximum of 0 and $n - m$. The SQL `EXCEPT` operator is different from monus in that Q_1 `EXCEPT` Q_2 eliminates *all* tuples that occur in Q_2, no matter what their multiplicity. The `EXCEPT` operation can be defined in our bag language as

$$Q_1 \texttt{ EXCEPT } Q_2 \stackrel{\text{def}}{=} \Pi_1(\sigma_{1=2}(Q_1 \times (\epsilon(Q_1) \dot{-} Q_2))).$$

We include monus in the algebra because it cannot be defined using `EXCEPT` and the rest of $\mathcal{BA}$. This follows from the characterization of interdefinability of the operations of $\mathcal{BA}$ in [GM93, LW93].

We will also use the operations $Q_1 \min Q_2$ (minimal intersection) and $Q_1 \max Q_2$ (maximal union) that create bags in which the multiplicity of any tuple is the minimum (maximum) of its multiplicities in Q_1 and Q_2. These can be defined in $\mathcal{BA}$ as $Q_1 \min Q_2 \stackrel{\text{def}}{=} Q_1 \dot{-} (Q_1 \dot{-} Q_2)$ and $Q_1 \max Q_2 \stackrel{\text{def}}{=} Q_1 \uplus (Q_2 \dot{-} Q_1)$.

For arbitrary queries Q_1 and Q_2 we use the notation $Q_1 \equiv Q_2$ to mean that for all database states s, $Q_1(s) = Q_2(s)$. The notation $Q_1 \subseteq Q_2$ means that for all database states s, $Q_1(s)$ is a subbag of $Q_2(s)$.

2.2 Transactions

Transactions $\mathcal{T}$ are functions from states to states. If s is a database state, then $\mathcal{T}(s)$ is the state resulting from the execution of transaction $\mathcal{T}$ in state s. $Q(\mathcal{T}(s))$ represents the value of query Q after $\mathcal{T}$ is executed in state s.

We consider *abstract transactions* defined with the notation $\mathcal{T} = \{R_1{:=}Q_1, \ldots, R_n{:=}Q_n\}$, abbreviated as $\mathcal{T} = \{R_i{:=}Q_i\}$. When $\mathcal{T}$ is executed in state s, then the value of R_i in state $\mathcal{T}(s)$ becomes $Q_i(s)$. That is, $\mathcal{T}$ executed in state s has the effect of simultaneously replacing the contents of each R_i with the result of evaluating query Q_i in state s.

Since we only consider view maintenance in response to insertions and deletions into base tables caused by a transaction, we will consider only *simple transactions* $\mathcal{T}$ of the form

$$\{R_1{:=}(R_1 \dot{-} \nabla R_1) \uplus \triangle R_1, \ldots, R_n{:=}(R_n \dot{-} \nabla R_n) \uplus \triangle R_n\}.$$

In other words, the value of R_i in state $\mathcal{T}(s)$ is $((R_i \dot{-} \nabla R_i) \uplus \triangle R_i)(s)$. This is without loss of generality since any abstract transaction can be transformed to an equivalent simple transaction.

2.3 Logs and differential tables

A log $\mathcal{L}$ is a collection of auxiliary base tables $\blacktriangledown R_1$, $\blacktriangle R_1$, …, $\blacktriangledown R_n$, $\blacktriangle R_n$. Suppose that database states are ordered and $s_p \leq s_c$, where s_p represents a state of the database that existed before the database entered state s_c. Informally, think of s_p as a *past* state and s_c as the *current state*. A log $\mathcal{L}$ *records the transition from state s_p to the state s_c*, written $s_p \stackrel{\mathcal{L}}{\rightarrow} s_c$, if, for each table R_i,

$$R_i(s_p) = ((R_i \dot{-} \blacktriangle R_i) \uplus \blacktriangledown R_i)(s_c).$$

That is, log $\mathcal{L}$ records all deletions ($\blacktriangledown R_i$) from and insertions ($\blacktriangle R_i$) into each table R_i that comprise the transition from state s_p to state s_c. Note that in order to compute the past value of R_i from the value of R_i in the current state, we must *delete* the bag that was inserted and *insert* the bag that was deleted. A similar technique is used in [CW91] with transition tables, which can be thought of as transient logs.

Our notion of logs is not the same as that of differential tables introduced in [SL76]. The tables B, A, and D are differential tables for table R if $R = (B \dot{-} D) \uplus A$. In this approach, every "base table" R is treated as a virtual table (view). Tables D and A can be thought of as suspended deletions and insertions, while B represents an "old" value of the table R. In contrast, our notion of a log assumes that the changes have been applied to the base tables.

A word about our use of white triangles (∇ and $\triangle$) and black triangles ($\blacktriangledown$ and $\blacktriangle$). The white triangles represent changes specified by the transactions, or changes computed from those specified in the transactions. The black triangles represent changes in the log or changes computed from the log.

2.4 Substitutions

We will denote general substitutions with the notation $\eta = [Q_1/R_1, \cdots, Q_n/R_n]$. The notation $\eta(Q)$ denotes the query that results from simultaneously replacing every occurrence of R_i in Q by Q_i. For example, if η is $[\epsilon(R_2)/R_1, \sigma_q(R_1)/R_2]$ and Q is $\sigma_p(R_1 \times R_2)$, then $\eta(Q)$ is $\sigma_p(\epsilon(R_2) \times \sigma_q(R_1))$.

The next subsection will make use of two substitutions $\widehat{\mathcal{T}}$ and $\widehat{\mathcal{L}}$ that are derived from simple transactions $\mathcal{T}$ and logs $\mathcal{L}$ as:

$$[((R_1 \dot{-} \nabla R_1) \uplus \triangle R_1)/R_1, \ldots, ((R_n \dot{-} \nabla R_n) \uplus \triangle R_n)/R_n]$$

and

$$[((R_1 \dot{-} \blacktriangle R_1) \uplus \blacktriangledown R_1)/R_1, \ldots, ((R_n \dot{-} \blacktriangle R_n) \uplus \blacktriangledown R_n)/R_n]$$

2.5 Past and future queries

Past and future queries are the key concepts of view maintenance as they allow us to compute the value of a query in a state that is different from the current one.

Definition 1 (Past and Future Queries):

1. Suppose s_p is a state that precedes state s_c. A query PQ is a *past-query* at state s_c for a query Q at s_p if $Q(s_p) = PQ(s_c)$. Informally, we can evaluate a past-query PQ in the current state in order to determine the value that Q had in an earlier state.

2. A query FQ is called a *future-query* at state s_p for Q at state s_c if $FQ(s_p) = Q(s_c)$. We call FQ a *future-query* for Q with respect to a transaction $\mathcal{T}$ if for every database state s we have $FQ(s) = Q(\mathcal{T}(s))$. That is, if the database is currently in state s, then we can evaluate FQ in order to determine the "future" value that query Q will have in the state immediately after $\mathcal{T}$ is executed. □

Transactions and logs can be used to compute future- and past-queries. If $\mathcal{T}$ is a simple transaction, then $\text{FUTURE}(\mathcal{T}, Q)$ defined as

$$\widehat{\mathcal{T}}(Q) \equiv Q((R_1 \dot{-} \nabla R_1) \uplus \triangle R_1, \ldots, (R_n \dot{-} \nabla R_n) \uplus \triangle R_n)$$

is a future-query for $Q(R_1, \ldots, R_n)$ with respect to $\mathcal{T}$. Indeed, for any state s, $(\text{FUTURE}(\mathcal{T}, Q))(s) = Q(\mathcal{T}(s))$.

If $\mathcal{L}$ is a log from state s_p to state s_c, then the values of R_i at s_p can be computed from the values of R_i at s_c and the log as $R_i(s_p) = ((R_i \dot{-} \blacktriangle R_i) \uplus \blacktriangledown R_i)(s_c)$. Therefore, $\text{PAST}(\mathcal{L}, Q)$ defined as

$$\widehat{\mathcal{L}}(Q) \equiv Q((R_1 \dot{-} \blacktriangle R_1) \uplus \blacktriangledown R_1, \ldots, (R_n \dot{-} \blacktriangle R_n) \uplus \blacktriangledown R_n)$$

is a past-query, at state s_c for Q at state s_p. That is, $Q(s_p) = (\text{PAST}(\mathcal{L}, Q))(s_c)$.

In summary, future-queries allow us to *anticipate* state changes, while past-queries allow us to *compensate* for changes that have already been made.

3 View Maintenance Scenarios

In what follows, the view V is defined by a query Q and materialized in the table MV. A materialized view is said to be consistent with its definition in state s if $Q(s) = MV(s)$.

Any correct solution to the immediate view maintenance problem must guarantee that the contents of the view table MV always be consistent with the definition of the view V. In other words, the formula $Q \equiv MV$ is an *invariant* that should hold in all database states. Any solution to the immediate view maintenance problem must then employ some method of augmenting user transactions with the updates to table MV needed to maintain this invariant.

This section demonstrates that the same approach can be used to characterize deferred view maintenance problems. We use database invariants to specify three deferred view maintenance scenarios. For each invariant, we specify algorithms for transforming user transactions into ones that maintain the invariant. These invariants are more complex than the immediate case since they must relate table MV to query Q as well as auxiliary tables. Unlike the immediate case, the deferred scenarios also require additional algorithms for refreshing view tables as well as for propagating changes to auxiliary tables. For each scenario considered, we explain the main idea behind the associated view maintenance algorithms. The details of the algorithms will be given in Section 5.

3.1 Database invariants

First, we need to introduce some terminology. For formula α and database state s, the notation $s \models \alpha$ means that α holds in state s. Given formulas α, β, and a transaction $\mathcal{T}$, we will use the *Hoare triple* $\{\alpha\}\mathcal{T}\{\beta\}$ (see [Gri81]) to assert that for every state s, if $s \models \alpha$, then $\mathcal{T}(s) \models \beta$. A transaction $\mathcal{T}$ is said to be *safe for* α if $\{\alpha\}\mathcal{T}\{\alpha\}$. That is, if α holds in a given state, then it will hold in the state after $\mathcal{T}$ is executed.

We assume that the database tables are partitioned into *external* tables that can be changed by user transactions (user-defined base tables) and *internal* tables that are used to store and support materialized views (such as MV, log tables, and view differential files). User transactions are not allowed to directly update internal tables.

A formula is called a *database invariant* if it is guaranteed to hold in every state. We shall denote database invariants by $\mathbb{INV}_*$ where the index $*$ specifies a named scenario for view maintenance. Given an invariant $\mathbb{INV}_*$ and a user transaction $\mathcal{T}$, it cannot be expected that $\mathcal{T}$ will be safe for $\mathbb{INV}_*$. Thus, each scenario requires an algorithm for transforming any user transaction $\mathcal{T}$ into a transaction $\mathsf{makesafe}_*[\mathcal{T}]$ that is safe for $\mathbb{INV}_*$. This transaction should have the same behavior as $\mathcal{T}$ on external tables. Hence, $\mathsf{makesafe}_*[\mathcal{T}]$ will augment $\mathcal{T}$ with changes to internal tables.

The scenarios describing deferred maintenance will also require various auxiliary functions to *refresh* view tables. For each $\mathbb{INV}_*$, we will define a transaction $\mathsf{refresh}_*$ such that $\{\mathbb{INV}_*\}\mathsf{refresh}_*\{Q \equiv MV\}$.

3.2 Immediate maintenance

We review the immediate update scenario in order to facilitate comparison with the deferred scenarios.

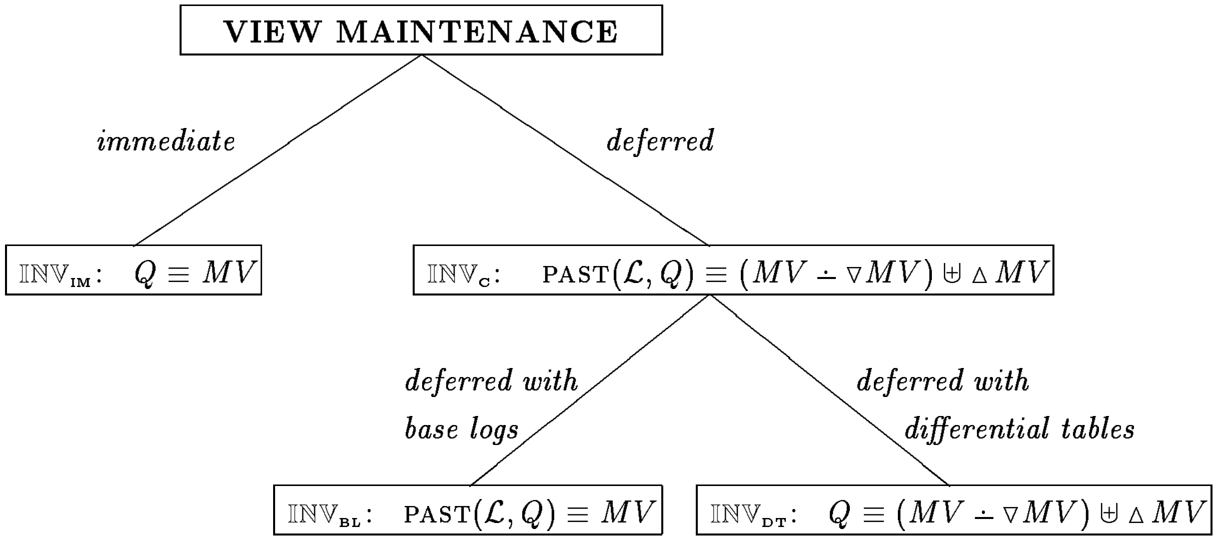

Figure 1: Invariants for view maintenance

Suppose that we require that the table MV always be consistent with its definition. As noted, this amounts to declaring the formula below to be a database invariant.

$\mathbb{INV}_{\text{IM}}$ $$\boxed{Q \equiv MV}$$

The literature on immediate view maintenance [BLT86, CW91, GL95, QW91, SI84] presents various approaches to converting any transaction $\mathcal{T}$ to a transaction $\mathsf{makesafe}_{\text{IM}}[\mathcal{T}]$ that is guaranteed to maintain $\mathbb{INV}_{\text{IM}}$. The method of choice is to produce *incremental queries*, $\triangledown(\mathcal{T}, Q)$ and $\vartriangle(\mathcal{T}, Q)$, such that augmenting $\mathcal{T}$ with

$$MV{:=}(MV \dot{-} \triangledown(\mathcal{T}, Q)) \uplus \vartriangle(\mathcal{T}, Q)$$

correctly maintains the view. Note that the incremental queries are typically evaluated in the state *before* the updates of $\mathcal{T}$ have been applied.

Although incremental queries can avoid the work of recomputing Q from scratch, their evaluation can still impose a large per-transaction overhead.

3.3 Deferred maintenance with base logs

Suppose that the table MV is allowed to become inconsistent with the definition of view V. This means that the content of table MV is equal to the value of Q in some *past* state when MV was last refreshed or was initialized. Suppose that log $\mathcal{L}$ records the changes made to base tables that make up the transition from this past state to the current state. This scenario can be captured with the invariant

$\mathbb{INV}_{\text{BL}}$ $$\boxed{\text{PAST}(\mathcal{L}, Q) \equiv MV}$$

Note that if the log is empty, then the view table is consistent since in this case $Q \equiv \text{PAST}(\mathcal{L}, Q)$. A solution to this scenario involves defining the transformation $\mathsf{makesafe}_{\text{BL}}[.]$ that maintains the invariant and a function $\mathsf{refresh}_{\text{BL}}$ that brings the view up-to-date.

For any user transaction $\mathcal{T}$, $\mathsf{makesafe}_{\text{BL}}[\mathcal{T}]$ must do two things: 1) execute $\mathcal{T}$, and (2) correctly extend the log $\mathcal{L}$ in order to maintain the invariant. This imposes little overhead on each transaction since we only need to record the changes made to base tables.

The refresh function must satisfy the specification $\{\mathbb{INV}_{\text{BL}}\}\mathsf{refresh}_{\text{BL}}\{Q \equiv MV\}$. In a manner similar to the immediate case, we could formulate incremental queries, $\blacktriangledown(\mathcal{L}, Q)$ and $\blacktriangle(\mathcal{L}, Q)$, such that the transaction

$$MV{:=}(MV \dot{-} \blacktriangledown(\mathcal{L}, Q)) \uplus \blacktriangle(\mathcal{L}, Q)$$

correctly refreshes the table MV. Unlike the immediate case, these incremental queries must be evaluated in a post-update state that reflects the changes recorded in log $\mathcal{L}$. In Section 4, we present a technique for computing incremental queries for post-update states.

We should expect that in most cases this incremental approach will be much less expensive than recomputing Q from scratch. However, the computation of the incremental queries still may be costly, which implies a high refresh time.

3.4 Deferred maintenance with differential tables for views

Many applications require a low refresh time. One way to minimize view downtime is to precompute the changes necessary for refreshing table MV and store them in "differential tables." This scenario can be captured with the invariant

$$\mathbb{INV}_{\text{DT}} \qquad \boxed{Q \;\equiv\; (MV \dot{-} \nabla MV) \uplus \triangle MV}$$

where ∇MV and $\triangle MV$ are the differential tables that maintain the changes needed to bring the view table up-to-date. Another way of saying this is that the differential tables record the difference of the past value of Q (stored in MV) and its current value. Note that if the differential tables are empty, then the view table MV is consistent.

The refresh function in this case applies the differential tables to MV,

$$MV{:=}(MV \dot{-} \nabla MV) \uplus \triangle MV,$$

and empties them. If the differential tables contain exactly the *net change* needed to refresh MV (that is, $\nabla MV \subseteq MV$ and $\nabla MV \min \triangle MV \equiv \phi$), then this represents the *minimal* possible refresh time for MV.

However, as in the immediate update case, the per-transaction overhead for maintaining the invariant may be high since $\mathsf{makesafe}_{\text{DT}}[\mathcal{T}]$ must maintain correct values for ∇MV and $\triangle MV$.

3.5 Deferred maintenance with differential tables and base logs

One of our goals is to present a new solution to the deferred view maintenance problem that provides (1) a fast refresh algorithm, and (2) low per-transaction overhead for maintaining auxiliary information.

Our solution combines the last two approaches. We maintain *both* a log $\mathcal{L}$ on base tables *and* a pair of differential tables, ∇MV and $\triangle MV$, for the view table MV. The combined invariant is

$$\mathbb{INV}_{\text{C}} \qquad \boxed{\text{PAST}(\mathcal{L}, Q) \;\equiv\; (MV \dot{-} \nabla MV) \uplus \triangle MV}$$

To understand this scenario, it helps to keep in mind three different states: (1) a past state s_p such that the table MV is consistent with Q in state s_p, (2) the current database state s_c, and (3) an intermediate state s_i, with $s_p \leq s_i \leq s_c$. The log $\mathcal{L}$ records the transition from s_i to s_c. That is, in this scenario the log is used to maintain the *view differential tables* (∇MV and $\triangle MV$), and records the changes to the base tables made since the last refresh of the differential tables (in state s_i). If the differential tables are applied to the view table MV to refresh it, then the contents of the table MV will correspond to the value that Q had in state s_i, when the log was initialized. That is, updating MV using the differential tables gives us the value of the past query for Q, $\text{PAST}(\mathcal{L}, Q)$.

The transaction $\mathsf{makesafe}_{\text{C}}[\mathcal{T}]$ is essentially the same as $\mathsf{makesafe}_{\text{BL}}[\mathcal{T}]$ — it only needs to update the log in order to maintain invariant $\mathbb{INV}_{\text{C}}$. The refresh function for this scenario must satisfy the specification $\{\mathbb{INV}_{\text{C}}\}\mathsf{refresh}_{\text{C}}\{Q \;\equiv\; MV\}$. In addition, this scenario suggests two auxiliary transactions: a transaction $\mathsf{propagate}_{\text{C}}$, that propagates to the differential tables the changes recorded in the log $\mathcal{L}$, and a transaction $\mathsf{partial\_refresh}_{\text{C}}$, that partially refreshes the view table by applying the differential tables. These transactions have the specifications:

$$\{\mathbb{INV}_{\text{C}}\}\ \mathsf{propagate}_{\text{C}}\ \{Q \;\equiv\; (MV \dot{-} \nabla MV) \uplus \triangle MV\},$$

$$\{\mathbb{INV}_{\text{C}}\}\ \mathsf{partial\_refresh}_{\text{C}}\ \{\text{PAST}(\mathcal{L}, Q) \;\equiv\; MV\}.$$

By decoupling incremental computation from *both* $\mathsf{refresh}_{\text{C}}$ and $\mathsf{makesafe}_{\text{C}}[\mathcal{T}]$, these auxiliary transactions will allow us to achieve our goal of low refresh time while simultaneously obtaining low per-transaction overhead. A more detailed discussion is presented in Section 5. Here we are only interested in a formal specification of this scenario.

Figure 1 summarizes the four invariants that describe different scenarios for view maintenance. Note that both the $\mathbb{INV}_{\text{BL}}$ and $\mathbb{INV}_{\text{DT}}$ scenarios can be considered as special cases of the $\mathbb{INV}_{\text{C}}$ scenario.

4 Exploiting Duality

As mentioned in the previous section, the method of choice for solving the immediate view maintenance problem involves finding incremental queries $\nabla(\mathcal{T}, Q)$ and $\triangle(\mathcal{T}, Q)$ such that the operation

$$MV{:=}(MV \dot{-} \nabla(\mathcal{T}, Q)) \uplus \triangle(\mathcal{T}, Q)$$

will correctly update the materialized view, provided that the queries $\nabla(\mathcal{T}, Q)$ and $\triangle(\mathcal{T}, Q)$ are evaluated in the *pre-update* database state. This amounts to solving for $\nabla(\mathcal{T}, Q)$ and $\triangle(\mathcal{T}, Q)$ in the equation

$$(1) \qquad \text{FUTURE}(\mathcal{T}, Q) \;\equiv\; (Q \dot{-} \nabla(\mathcal{T}, Q)) \uplus \triangle(\mathcal{T}, Q)$$

since table MV is assumed to contain the current value of Q and we wish to update MV to contain the value that Q will have in the future, after $\mathcal{T}$ is executed. An example of such an algorithm for the bag algebra can be found in [GL95].

Now let us turn to the simple case of deferred maintenance. Suppose that MV was initialized or last refreshed at state s_p and the database is currently in state s_c. Suppose that $\mathcal{L}$ is a log from s_p to s_c. In order to incrementally refresh MV we want to find two queries $\blacktriangledown(\mathcal{L}, Q)$ and $\blacktriangle(\mathcal{L}, Q)$ such that the operation

$$MV{:=}(MV \dot{-} \blacktriangledown(\mathcal{L}, Q)) \uplus \blacktriangle(\mathcal{L}, Q)$$

will correctly update the materialized view.

Note that these incremental queries must be evaluated in the *post-update* database state that reflects all of the changes recorded in $\mathcal{L}$. Finding such incremental queries amounts to solving for $\blacktriangledown(\mathcal{L}, Q)$ and $\blacktriangle(\mathcal{L}, Q)$ in the equation

$$(2) \qquad Q \equiv (\text{PAST}(\mathcal{L}, Q) \dot{-} \blacktriangledown(\mathcal{L}, Q)) \uplus \blacktriangle(\mathcal{L}, Q)$$

since table MV is assumed to contain the past value of Q and we wish to update MV to contain the current value of Q.

Can we use the same algorithm for the pre- and post-update states? As we have indicated (see Section 1.2), this cannot be done directly without producing incorrect results. There is, however, a natural *duality* between future- and past-queries that can be exploited to solve this problem. Recall from Section 2.5 that both of these queries are formed as substitution instances,

$$\text{FUTURE}(\mathcal{T}, Q) \stackrel{\text{def}}{=} \widehat{\mathcal{T}}(Q), \qquad \text{PAST}(\mathcal{L}, Q) \stackrel{\text{def}}{=} \widehat{\mathcal{L}}(Q)$$

and that each query is formed by replacing every occurrence of a base table name R_i with a query of the form $(R_i \dot{-} D_i) \uplus A_i$. However, the roles of insertions and deletions are reversed since future-queries anticipate the changes that a transaction will make, while past-queries compensate for changes that have already been made.

Suppose that η is a substitution (see Section 2.4), and suppose that we have a method for constructing queries $\text{DEL}(\eta, Q)$ and $\text{ADD}(\eta, Q)$ such that

$$(3) \qquad \eta(Q) \equiv (Q \dot{-} \text{DEL}(\eta, Q)) \uplus \text{ADD}(\eta, Q).$$

Algorithms that produce the queries $\text{DEL}(\eta, Q)$ and $\text{ADD}(\eta, Q)$ are called *differential algorithms* (terminology is from [Pai84]). Solving Equation (1) is then simply a matter of defining $\nabla(\mathcal{T}, Q)$ and $\triangle(\mathcal{T}, Q)$ as

$$\nabla(\mathcal{T}, Q) \stackrel{\text{def}}{=} \text{DEL}(\widehat{\mathcal{T}}, Q), \qquad \triangle(\mathcal{T}, Q) \stackrel{\text{def}}{=} \text{ADD}(\widehat{\mathcal{T}}, Q).$$

Solving (2) for $\blacktriangledown(\mathcal{L}, Q)$ and $\blacktriangle(\mathcal{L}, Q)$ is not quite so simple. First, applying Equation (3) with $\eta = \widehat{\mathcal{L}}$ results in

$$\text{PAST}(\mathcal{L}, Q) \equiv \widehat{\mathcal{L}}(Q) \equiv (Q \dot{-} \text{DEL}(\widehat{\mathcal{L}}, Q)) \uplus \text{ADD}(\widehat{\mathcal{L}}, Q).$$

Now in order to solve Equation (2) we must "cancel" the incremental queries. We can do this using the following lemma.

Lemma 1 (cancellation) *Suppose that N, O, I, and D are queries. If $N \equiv (O \dot{-} D) \uplus I$, then $O \equiv (N \dot{-} I) \uplus (O \min D)$.* □

Applying this lemma to the above equation yields

$$Q \equiv (\text{PAST}(\mathcal{L}, Q) \dot{-} \text{ADD}(\widehat{\mathcal{L}}, Q)) \uplus (Q \min \text{DEL}(\widehat{\mathcal{L}}, Q)).$$

Therefore, (2) can be solved by defining the queries $\blacktriangledown(\mathcal{L}, Q)$ and $\blacktriangle(\mathcal{L}, Q)$ as

$$\begin{aligned} \blacktriangledown(\mathcal{L}, Q) &\stackrel{\text{def}}{=} \text{ADD}(\widehat{\mathcal{L}}, Q) \\ \blacktriangle(\mathcal{L}, Q) &\stackrel{\text{def}}{=} Q \min \text{DEL}(\widehat{\mathcal{L}}, Q). \end{aligned}$$

4.1 Incremental computation

Note that the query $\blacktriangle(\mathcal{L}, Q) \stackrel{\text{def}}{=} Q \min \text{DEL}(\widehat{\mathcal{L}}, Q)$ could be simplified to $\text{DEL}(\widehat{\mathcal{L}}, Q)$ if we knew that $\text{DEL}(\widehat{\mathcal{L}}, Q) \subseteq Q$. This is related to the "minimality" conditions of [GL95, QW91]. These conditions limit the number of unnecessary tuples produced by the incremental change queries.

The minimality constraints typically imposed on solutions to $\eta(Q) \equiv (Q \dot{-} \text{DEL}(\eta, Q)) \uplus \text{ADD}(\eta, Q)$ are

(a) $\text{DEL}(\eta, Q) \subseteq Q$: Only tuples actually in Q are in the deleted bag.

(b) $\text{DEL}(\eta, Q) \min \text{ADD}(\eta, Q) \equiv \phi$: No tuple is deleted and then reinserted.

The design of differential algorithms to compute $\text{DEL}(\eta, Q)$ and $\text{ADD}(\eta, Q)$ then involves a choice of imposing none of these constraints, or of imposing one of the three possible combinations of them. A solution meeting condition (a) will be called *weakly minimal*, while a solution meeting both conditions (a) and (b) will be called *strongly minimal.* In this paper, we present algorithms that produce weakly minimal solutions, for which the following simpler equations hold:

$$\begin{aligned} \blacktriangledown(\mathcal{L}, Q) &\stackrel{\text{def}}{=} \text{ADD}(\widehat{\mathcal{L}}, Q) \\ \blacktriangle(\mathcal{L}, Q) &\stackrel{\text{def}}{=} \text{DEL}(\widehat{\mathcal{L}}, Q). \end{aligned}$$

We will assume that every substitution $\eta = [Q_1/R_1, \cdots, Q_n/R_n]$ has a *factored* form. That is, every query Q_i is of the form $(R_i \dot{-} D_i) \uplus A_i$. Note that (1) if $\eta = \widehat{\mathcal{T}}$, then $D_i = \nabla R_i$ and $A_i = \triangle R_i$, and (2) if $\eta = \widehat{\mathcal{L}}$, then $D_i = \blacktriangle R_i$ and $A_i = \blacktriangledown R_i$.

A factored substitution is called *weakly minimal* if $D_i \subseteq R_i$. Note that any factored substitution η can be transformed into an equivalent weakly or strongly minimal substitution.

A simple transaction is called weakly minimal if $\widehat{\mathcal{T}}$ is a weakly minimal substitution. Similarly, a log $\mathcal{L}$ is called weakly minimal if $\widehat{\mathcal{L}}$ is a weakly minimal substitution. This amounts to declaring that $\blacktriangle R_i \subseteq R_i$ is a database invariant, for each table R_i. As we will see in the next section, care must be taken to guarantee that these invariants are maintained.

Figure 2 presents our algorithm for calculating $\text{DEL}(\eta, Q)$ and $\text{ADD}(\eta, Q)$ for weakly minimal substitutions. When Q is ϕ or $\{x\}$, then $\text{DEL}(\eta, Q) \equiv \text{ADD}(\eta, Q) \equiv \phi$. This algorithm is derived from the same change propagation rules for the bag algebra that were used in [GL95] to derive a strongly minimal algorithm. The functions $\nabla(\mathcal{T}, Q)$, $\triangle(\mathcal{T}, Q)$, $\blacktriangledown(\mathcal{L}, Q)$, $\blacktriangle(\mathcal{L}, Q)$ can be derived straightforwardly from Figure 2. For example, the equation for $\text{DEL}(\eta, E \dot{-} F)$ in Figure 2 gives rise to the equation

$$\nabla(\mathcal{T}, E \dot{-} F) \stackrel{\text{def}}{=} (\nabla(\mathcal{T}, E) \uplus \triangle(\mathcal{T}, F)) \min (E \dot{-} F),$$

Q	$\textsc{Del}(\eta, Q)$
R_i	D_i, where $\eta(R_i) = (R_i \dot{-} D_i) \uplus A_i$
$\sigma_p(E)$	$\sigma_p(\textsc{Del}(\eta, E))$
$\Pi_A(E)$	$\Pi_A(\textsc{Del}(\eta, E))$
$\epsilon(E)$	$\epsilon(\textsc{Del}(\eta, E)) \dot{-} (E \dot{-} \textsc{Del}(\eta, E))$
$E \uplus F$	$\textsc{Del}(\eta, E) \uplus \textsc{Del}(\eta, F)$
$E \dot{-} F$	$(\textsc{Del}(\eta, E) \uplus \textsc{Add}(\eta, F)) \min (E \dot{-} F)$
$E \times F$	$(\textsc{Del}(\eta, E) \times \textsc{Del}(\eta, F)) \uplus (\textsc{Del}(\eta, E) \times (F \dot{-} \textsc{Del}(\eta, F))) \uplus ((E \dot{-} \textsc{Del}(\eta, E)) \times \textsc{Del}(\eta, F))$

Q	$\textsc{Add}(\eta, Q)$
R_i	A_i, where $\eta(R_i) = (R_i \dot{-} D_i) \uplus A_i$
$\sigma_p(E)$	$\sigma_p(\textsc{Add}(\eta, E))$
$\Pi_A(E)$	$\Pi_A(\textsc{Add}(\eta, E))$
$\epsilon(E)$	$\epsilon(\textsc{Add}(\eta, E)) \dot{-} (E \dot{-} \textsc{Del}(\eta, E))$
$E \uplus F$	$\textsc{Add}(\eta, E) \uplus \textsc{Add}(\eta, F)$
$E \dot{-} F$	$((\textsc{Add}(\eta, E) \uplus \textsc{Del}(\eta, F)) \dot{-} (F \dot{-} E)) \dot{-} ((\textsc{Del}(\eta, E) \uplus \textsc{Add}(\eta, F)) \dot{-} (E \dot{-} F))$
$E \times F$	$(\textsc{Add}(\eta, E) \times \textsc{Add}(\eta, F)) \uplus (\textsc{Add}(\eta, E) \times (F \dot{-} \textsc{Del}(\eta, F))) \uplus ((E \dot{-} \textsc{Del}(\eta, E)) \times \textsc{Add}(\eta, F))$

Figure 2: Mutually Recursive functions $\textsc{Del}(\eta, Q)$ and $\textsc{Add}(\eta, Q)$.

as well as its dual equation

$$\blacktriangle(\mathcal{L}, E \dot{-} F) \stackrel{\text{def}}{=} (\blacktriangle(\mathcal{L}, E) \uplus \blacktriangledown(\mathcal{L}, F)) \min (E \dot{-} F).$$

Theorem 2 (Correctness of Differentiation) *For any query Q and any weakly minimal substitution η,*

(a) $\eta(Q) \equiv (Q \dot{-} \textsc{Del}(\eta, Q)) \uplus \textsc{Add}(\eta, Q)$,

(b) $\textsc{Del}(\eta, Q) \subseteq Q$. □

It can be verified that our post-update algorithm gives the correct answers in the examples presented in Section 1.2.

One of the reasons that we chose to use a weakly minimal solution in this paper is that the expressions are somewhat less complicated than for other solutions, and the algorithm can be seen as a generalization of [BLT86, Han87] to the full bag algebra $\mathcal{BA}$.

It should be emphasized that the issue of minimality of incremental algorithms is completely *orthogonal* to the problem of maintaining views in a deferred manner. Any abstract transaction can be transformed into an equivalent (weakly or strongly) minimal simple transaction, and the same is true for logs. The algorithms in Figure 2, and those of Section 5.1 could be modified to maintain any combination of the minimality conditions (a) and (b), including no minimality constraints at all. For example, in order to produce a strongly minimal solution, one could use the strongly minimal incremental algorithm presented in [GL95], and then modify the algorithms in Figure 3 by enforcing strong minimality.

4.2 How the state bug has been avoided

There are two ways of directly using the pre-update algorithm in the post-update state. The first is exemplified by [Han87], where differential tables are used to suspend the application of changes to database tables. In other words, updates are not actually applied but simply stored in differential tables. Past values of base tables are directly available and do not have to be computed. In this way, the pre-update algorithm will give the correct result. However, this approach is not sufficiently general since it assumes that all database tables are implemented with differential tables. This assumption may be unrealistic in many applications.

The second method can be explained with this observation:

Remark 1 *For certain* restricted *classes of views and updates, the equations derived by the pre-update and post-update algorithms produce the same results upon evaluation in the post-update state.*

For example, it can be shown that if Q is an SPJ query without self-joins, $\mathcal{T}$ is a weakly minimal transaction that inserts into and/or deletes from a single table R, and log $\mathcal{L}$ records only the changes of one such transaction $\mathcal{T}$, then $\triangledown(\mathcal{T}, Q) \equiv \blacktriangledown(\mathcal{L}, Q)$ and $\triangle(\mathcal{T}, Q) \equiv \blacktriangle(\mathcal{L}, Q)$.

If these restrictions are relaxed even slightly (i.e., an SPJ query is allowed to have self-joins, or multiple tables are updated), then it is easy to find examples of views and/or updates for which the pre-update algorithms of [BLT86, GL95, Han87, QW91] will give incorrect results if the incremental queries are evaluated in the post-update state.

5 Algorithms and Policies

This section presents algorithmic solutions for the three scenarios of deferred view maintenance described in Section 3. Each set of algorithms can be used to implement a wide range of view update *policies*. By a *policy* we mean a scheme by which the refresh functions are actually invoked for a given view. For example, in the simple scenario defined by invariant $\mathbb{INV}_{\text{BL}}$, the function $\mathsf{refresh}_{\text{BL}}$ could be invoked (1) only on demand by a user, (2) whenever the table MV is queried, or (3) in a periodic way. The section ends with a presentation of two policies for the $\mathbb{INV}_{\text{C}}$ scenario that can be used to minimize view downtime.

5.1 Algorithms

Figure 3 presents algorithmic solutions for the three scenarios of deferred view maintenance described in Section 3. The notation $\mathcal{L}{:=}\phi$ is used to abbreviate the operations needed to empty log tables ($\blacktriangledown R_1{:=}\phi, \ldots, \blacktriangle R_n{:=}\phi$). If $\mathcal{T}_1$ and $\mathcal{T}_2$ are transactions, then $\mathcal{T}_1 + \mathcal{T}_2$ denotes the transaction that has the same behavior as performing the operations of $\mathcal{T}_1$ and $\mathcal{T}_2$ simultaneously. That is, we may view $\mathcal{T}_1 + \mathcal{T}_2$ as performing $\mathcal{T}_1$ and $\mathcal{T}_2$ in a way that operations in $\mathcal{T}_1$ do not see the effect of operations in $\mathcal{T}_2$, and vice versa.

These high-level algorithms are built from two main components: the pre- and post-update differential algorithms presented in Section 4, and a method for composing two sequential updates into a single update. The latter is provided by the following lemma.

Lemma 3 (Weakly Minimal Composition) *Suppose that O, I_1, I_2, D_1 and D_2 are queries such that $D_1 \subseteq O$ and $D_2 \subseteq (O \dot{-} D_1) \uplus I_1$. Let $D_3 \stackrel{\text{def}}{=} D_1 \uplus (D_2 \dot{-} I_1)$ and $I_3 \stackrel{\text{def}}{=} (I_1 \dot{-} D_2) \uplus I_2$. Then*

(a) $(((O \dot{-} D_1) \uplus I_1) \dot{-} D_2) \uplus I_2 \equiv (O \dot{-} D_3) \uplus I_3$,

(b) $D_3 \subseteq O$.

As an example, we show how $\mathsf{propagate}_{\text{C}}$ from Figure 3 is derived. Equation (2) in Section 4 tells us that

$$Q \equiv (\text{PAST}(\mathcal{L}, Q) \dot{-} \blacktriangledown(\mathcal{L}, Q)) \uplus \blacktriangle(\mathcal{L}, Q),$$

and invariant $\mathbb{INV}_{\text{C}}$ tells us that

$$\text{PAST}(\mathcal{L}, Q) \equiv (MV \dot{-} \triangledown MV) \uplus \vartriangle MV.$$

This implies that

$$Q \equiv (((MV \dot{-} \triangledown MV) \uplus \vartriangle MV) \dot{-} \blacktriangledown(\mathcal{L}, Q)) \uplus \blacktriangle(\mathcal{L}, Q).$$

By the composition lemma, we then get

$$Q \equiv (MV \dot{-} (\triangledown MV \uplus (\blacktriangledown(\mathcal{L}, Q) \dot{-} \vartriangle MV))) \uplus ((\vartriangle MV \dot{-} \blacktriangledown(\mathcal{L}, Q)) \uplus \blacktriangle(\mathcal{L}, Q)).$$

5.2 Correctness

As discussed in Section 4.1 our solutions will impose the following *minimality invariants*, in addition to the invariants described in Figure 1. In the two cases that use a log $\mathcal{L}$, we require that the invariants $\blacktriangle R_i \subseteq R_i$ be maintained. In the two cases that use differential tables we will require that the invariant $\triangledown MV \subseteq MV$ be maintained.

The following lemma tells us that the transactions of Figure 3 correctly extend the log and maintain the minimality invariants.

Lemma 4 *Suppose that $\mathcal{L}$ is a weakly minimal log, $s_p \stackrel{\mathcal{L}}{\to} s_c$, and $\mathcal{T}$ is a weakly minimal transaction. Then $s_p \stackrel{\mathcal{L}}{\to} (\mathsf{makesafe}_{\text{BL}}[\mathcal{T}])(s_c)$. Furthermore, the transaction $\mathsf{makesafe}_{\text{BL}}[\mathcal{T}]$ is safe for $\blacktriangle R_i \subseteq R_i$ for each table R_i, and the transactions $\mathsf{makesafe}_{\text{DT}}[\mathcal{T}]$ and $\mathsf{propagate}_{\text{C}}$ are safe for $\triangledown MV \subseteq MV$.* □

The following theorem tells us that our algorithms meet the specifications given in Section 3.

Theorem 5 *The algorithms of Figure 3 are correct. That is, every transaction $\mathsf{makesafe}_*[\mathcal{T}]$ is safe for $\mathbb{INV}_*$ for $*$ being BL, DT and C. The refresh transactions are correct:*

$$\{\mathbb{INV}_*\}\mathsf{refresh}_*\{Q \equiv MV\}$$

In addition, the following holds:

$$\{\mathbb{INV}_{\text{C}}\}\ \mathsf{propagate}_{\text{C}}\ \{Q \equiv (MV \dot{-} \triangledown MV) \uplus \vartriangle MV\}$$
$$\{\mathbb{INV}_{\text{C}}\}\ \mathsf{partial\_refresh}_{\text{C}}\ \{\text{PAST}(\mathcal{L}, Q) \equiv MV\}$$ □

5.3 Minimizing view downtime

The two transactions, $\mathsf{propagate}_{\text{C}}$ and $\mathsf{partial\_refresh}_{\text{C}}$, of the $\mathbb{INV}_{\text{C}}$ scenario allow for a very rich set of maintenance policies. We now present two policies for that scenario and describe how they minimize view downtime.

Policy 1: Every k time units, the transaction $\mathsf{propagate}_{\text{C}}$ is invoked to propagate changes from the log $\mathcal{L}$ to the differential tables, $\triangledown MV$ and $\vartriangle MV$. Every m time units ($m > k$), the view table MV is brought up-to-date using $\mathsf{refresh}_{\text{C}}$.

Policy 2: The use of $\mathsf{propagate}_{\text{C}}$ is the same as that in Policy 1. Every m time units ($m > k$), the view table MV is *partially refreshed* using $\mathsf{partial\_refresh}_{\text{C}}$.

With both policies, per-transaction overhead is minimized since $\mathsf{makesafe}_{\text{C}}[\mathcal{T}]$ only adds the work required to update the log tables. Policy 1 can be expected to have a refresh time much lower than that of the $\mathbb{INV}_{\text{BL}}$ scenario. This is because much of the work of computing incremental changes has already been done during periodic propagation. Policy 2 has the least downtime

$$\underline{\mathbb{INV}_{\text{BL}}\text{: } \textsc{past}(\mathcal{L}, Q) \equiv MV}$$

$$\mathsf{makesafe}_{\text{BL}}[\mathcal{T}] = \left\{ \begin{array}{l} \blacktriangledown R_i := \blacktriangledown R_i \uplus (\triangledown R_i \dot{-} \blacktriangle R_i), \\ \blacktriangle R_i := (\blacktriangle R_i \dot{-} \triangledown R_i) \uplus \triangle R_i \end{array} \right\} + \mathcal{T}$$

$$\mathsf{refresh}_{\text{BL}} = \{MV{:=}(MV \dot{-} \blacktriangledown(\mathcal{L}, Q)) \uplus \blacktriangle(\mathcal{L}, Q), \quad \mathcal{L} := \phi\}$$

$$\underline{\mathbb{INV}_{\text{DT}} : Q \equiv (MV \dot{-} \triangledown MV) \uplus \triangle MV}$$

$$\mathsf{makesafe}_{\text{DT}}[\mathcal{T}] = \left\{ \begin{array}{l} \triangledown MV := \triangledown MV \uplus (\triangledown(\mathcal{T}, Q) \dot{-} \triangle MV), \\ \triangle MV := (\triangle MV \dot{-} \triangledown(\mathcal{T}, Q)) \uplus \triangle(\mathcal{T}, Q) \end{array} \right\} + \mathcal{T}$$

$$\mathsf{refresh}_{\text{DT}} = \{MV{:=}(MV \dot{-} \triangledown MV) \uplus \triangle MV, \quad \triangledown MV{:=}\phi, \quad \triangle MV{:=}\phi\}$$

$$\underline{\mathbb{INV}_{\text{C}} : \textsc{past}(\mathcal{L}, Q) \equiv (MV \dot{-} \triangledown MV) \uplus \triangle MV}$$

$$\mathsf{makesafe}_{\text{C}}[\mathcal{T}] = \mathsf{makesafe}_{\text{BL}}[\mathcal{T}]$$

$$\mathsf{propagate}_{\text{C}} = \left\{ \begin{array}{l} \triangledown MV := \triangledown MV \uplus (\blacktriangledown(\mathcal{L}, Q) \dot{-} \triangle MV), \\ \triangle MV := (\triangle MV \dot{-} \blacktriangledown(\mathcal{L}, Q)) \uplus \blacktriangle(\mathcal{L}, Q), \\ \mathcal{L} := \phi \end{array} \right\}$$

$$\mathsf{partial\_refresh}_{\text{C}} = \mathsf{refresh}_{\text{DT}}$$

$$\mathsf{refresh}_{\text{C}} = \begin{array}{l} (\mathsf{propagate}_{\text{C}} \text{ followed by } \mathsf{partial\_refresh}_{\text{C}}) \quad \text{or} \\ (\mathsf{partial\_refresh}_{\text{C}} \text{ followed by } \mathsf{refresh}_{\text{BL}}) \end{array}$$

Figure 3: Deferred View Maintenance Algorithms

since it merely applies the precomputed differential tables to the view table. Policy 2 refreshes the view to a state that is at most k time units out-of-date. This policy is appropriate for applications that can tolerate data that is slightly out-of-date (assuming k is small).

One can minimize view downtime further by removing, from $\triangledown MV$ and $\triangle MV$, tuples that exist in both $\triangledown MV$ and $\triangle MV$. Such a solution would be generated by using strong minimality (Section 3.4), and requires a strongly minimal analog of Lemma 3.

Example 5.4 Again we consider the retail application of Section 1.1. Suppose that we use the $\mathbb{INV}_{\text{C}}$ scenario for the materialized view MV, and maintain logs on the changes to the `sales` table. In this example, the refresh period is 24 hours ($m = 24$). Suppose that propagation is done every hour ($k = 1$).

Using Policy 1, we can expect the downtime to be much smaller than it would be in the $\mathbb{INV}_{\text{BL}}$ scenario, since the log would contain at most an hour's worth of changes rather than a day's worth. The refresh of Policy 2 results in a view table that is no more than one hour out-of-date, and has the minimal downtime. □

Of course, there are many possible variations on these two policies. For example, rather than using a fixed interval k, the transaction $\mathsf{propagate}_{\text{C}}$ could be invoked asynchronously whenever any free cycles are available. Similarly, $\mathsf{refresh}_{\text{C}}$ or $\mathsf{partial\_refresh}_{\text{C}}$ could be invoked only when a user queries the view.

6 Related Work

Several incremental view maintenance algorithms for immediate maintenance have been proposed [BLT86, GL95, Han87, QW91]. These algorithms are based on the assumption that access to the pre-update base tables is available. Equations that involve both pre-update and post-update base tables are presented in [CW91, GMS93]. In [CW91], the incremental changes are computed in the post-update state. The pre-update state of a table is computed from its post-update state and from the transition tables that contain update information. Our future queries are similar to the `when`-clause of [GHJ96].

Research related to deferred view maintenance has focussed on two main issues: (a) computing the changes to the view and (b) applying the changes to the view. The work on computing updates has involved issues such as the types of auxiliary information needed to compute incremental changes, and detecting relevant updates. All of this work, however, has been done in the context of restrictive classes of views. Database snapshots were proposed in [AL80] as a means of providing access to old database states and also as a way of optimizing the performance of large, distributed databases. An algorithm for determining the changes that should be made to snapshots (restricted to select-project views over base tables) is presented in [LH+86]. Techniques for maintaining update logs to allow efficient detection of relevant updates to select-project views are given in [KR87] and [SP89]. Deferred maintenance for select-join views is implemented in ADMS [RK86].

Issues related to the process of applying the computed updates to the view have been studied in [SR88] and [AGK95]. The problem of determining the optimal refresh frequency, based on queueing models and parameterization of response time and processing cost constraints, has been investigated in [SR88]. View refresh strategies based on different priorities for transactions that apply computed updates to a view and transactions that read a view are presented in [AGK95]. While our paper is also concerned with the issue of balancing the costs of refresh with the constraints of other transactions, the focus is on high-level algorithms for incremental maintenance based on the various methods of keeping auxiliary information to achieve this balance. A comparison of view processing techniques based on non-materialization, and immediate and deferred view maintenance is presented in [Han87]. The algorithms for deferred maintenance used in that paper are based on future updates and hypothetical tables.

7 Future work

There are many directions for future work. For example, are there algorithms to refresh only those parts of a view needed by a given query? How should log information be stored so that the work done by $\mathsf{makesafe}_{\mathrm{BL}}[\mathcal{T}]$ is minimal, and independent of the number of views supported? What are the problems related to concurrency control in the presence of materialized views?

Acknowledgments. We would like to thank Teradata/Walmart Support Group and Dave Belanger for the initial discussions that led to this work, and Dan Lieuwen and the referees for helpful comments.

References

[AGK95] B. Adelberg, H. Garcia-Molina, and B. Kao. Applying update streams in a soft real-time database system. In *SIGMOD'95*.

[AL80] M. Adiba and B. Lindsay. Database snapshots. In *VLDB'80*.

[BLT86] J. Blakeley, P. Larson, and F. Tompa. Efficiently Updating Materialized Views. In *SIGMOD'86*.

[CG+96] L. Colby, T. Griffin, L. Libkin, I. Mumick, and H. Trickey. Algorithms for deferred view maintenance. *Bell Labs Tech. Memo*, 1996.

[CW91] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *VLDB'91*.

[GHJ96] S. Ghandeharizadeh, R. Hull, and D. Jacobs. Heraclitus: Elevating deltas to be first class citizens in a database programming language. *ACM TODS*, to appear.

[GJM96] A. Gupta, H. Jagadish, and I. Mumick. Data integration using self-maintainable views. In *EDBT'96*.

[GL95] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *SIGMOD'95*.

[GM93] S. Grumbach and T. Milo. Towards tractable algebras for bags. In *PODS'93*.

[GM95] A. Gupta and I. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE DE Bulletin*, 18(2):3–19, 1995.

[GM96] A. Gupta and I. Mumick, editors. *Materialized Views*. MIT Press, 1996. To be published.

[GMS93] A. Gupta, I. Mumick, and V. Subrahmanian. Maintaining views incrementally. In *SIGMOD'93*.

[Gri81] D. Gries. *The Science of Programming*. Springer, 1981.

[Han87] E. Hanson. A performance analysis of view materialization strategies. In *SIGMOD'87*.

[KR87] B. Kähler and O. Risnes. Extended logging for database snapshots. In *VLDB'87*.

[LH+86] B. Lindsay, L. Haas, C. Mohan, H. Pirahesh, and P. Wilms. A snapshot differential refresh algorithm. In *SIGMOD'86*.

[LW93] L. Libkin and L. Wong. Some properties of query languages for bags. In *DBPL'93*.

[Pai84] R. Paige. Applications of finite differencing to database integrity control and query/transaction optimization. In H. Gallaire, J. Minker, and J. Nicolas, editors, *Advances in Database Theory*, 1984.

[QW91] X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *IEEE TKDE*, 3(3):337–341, 1991.

[RK86] N. Roussopoulos and H. Kang. Principles and techniques in the design of ADMS$\pm$. *IEEE Computer*, 19:19–25, 1986.

[SI84] O. Shmueli and A. Itai. Maintenance of Views. In *SIGMOD'84*.

[SL76] D. Severance and G. Lohman. Differential files: Their application to the maintenance of large databases. *ACM TODS*, 1(3):256–267, 1976.

[SP89] A. Segev and J. Park. Updating distributed materialized views. *IEEE TKDE*, 1(2):173–184, 1989.

[SR88] J. Srivastava and D. Rotem. Analytical modeling of materialized view maintenance. In *PODS'88*.

[ZG+95] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *SIGMOD'95*.