ACM SIGMOD Conference, May 1997, pp.405–416

Supporting Multiple View Maintenance Policies

Latha S. Colby*
Red Brick Systems
colby@redbrick.com

Akira Kawaguchi†
Columbia University
akira@cs.columbia.edu

Daniel F. Lieuwen
Bell Labs, Lucent Technologies
lieuwen@research.bell-labs.com

Inderpal Singh Mumick
AT&T Laboratories
mumick@research.att.com

Kenneth A. Ross†
Columbia University
kar@cs.columbia.edu

Abstract

Materialized views and view maintenance are becoming increasingly important in practice. In order to satisfy different data currency and performance requirements, a number of view maintenance policies have been proposed. Immediate maintenance involves a potential refresh of the view after every update to the deriving tables. When staleness of views can be tolerated, a view may be refreshed periodically or (on-demand) when it is queried. The maintenance policies that are chosen for views have implications on the validity of the results of queries and affect the performance of queries and updates. In this paper, we investigate a number of issues related to supporting multiple views with different maintenance policies.

We develop formal notions of consistency for views with different maintenance policies. We then introduce a model based on view groupings for view maintenance policy assignment, and provide algorithms, based on the viewgroup model, that allow consistency of views to be guaranteed. Next, we conduct a detailed study of the performance aspects of view maintenance policies based on an actual implementation of our model. The performance study investigates the trade-offs between different maintenance policy assignments. Our analysis of both the consistency and performance aspects of various view maintenance policies are important in making correct maintenance policy assignments.

1 Introduction

Materialized views are becoming important for providing viable solutions to problems in applications related to billing, retailing, decision support, data warehousing and data integration [GM95, Mum95, ZGHW95, HZ96]. A materialized view is like a data cache; the main reason for defining and storing a materialized view is to increase query performance. However, view maintenance imposes a penalty on update transactions. In order to minimize this penalty, different policies for view maintenance have been proposed, depending on the read/update transaction mix and on the need for queries to see current data. Three common policies are: (1) Immediate Views: The view is maintained immediately upon an update to a base table, as a part of the transaction that updates the base table. Immediate maintenance allows fast querying, at the expense of slowing down update transactions. (2) Deferred Views [RK86, AGK95, AKG96, CGL+96]: The view is maintained by a transaction that is separate from the update transactions and is typically invoked when the view is queried. Deferred maintenance thus leads to comparatively slower querying than immediate maintenance, but allows faster updates. (3) Snapshot Views [AL80, LHM+86]: The view is maintained periodically, say once a day or once a week, by an asynchronous process. Snapshot maintenance allows fast querying and updates, but queries can read data that is not up-to-date with base tables.

Motivation: When trying to implement materialized views, one has to decide what maintenance policy one should adopt. A system may require all three policies for different views: An application has to make a view immediate if it expects a very high query rate and/or real-time response requirements. For example, in a cellular billing application, the balance due is a view on cellular call data, and can be used to block future calls. Clearly, this view must be immediately maintained. However, immediate maintenance is not scalable with respect to the number of views, so a system cannot define many immediate views. Deferred and snapshot maintenance are scalable with respect to the number of views, so it is often desirable to define most views as deferred or snapshot. If one can tolerate stale data, or needs a stable data set, then snapshot views are probably the best choice. If it is important that a view be up-to-date with base tables, but if the query response time is not critical, then a deferred view is appropriate. The following example motivates the need for different view maintenance policies:

EXAMPLE 1.1 Consider a membership based retailing company (such as Price Club, Sam's Club, or BJ's) with a database of sales and return transactions from several stores. Tables `sales`, `customer`, `supplier`, and `supplies` are maintained in the database. The `sales` table contains the de-

*The work of L. Colby was performed while at Bell Labs.

†The work of A. Kawaguchi and K. Ross was performed while visiting AT&T Laboratories and Bell Laboratories, and was partially supported by a grant from the AT&T Foundation, by a David and Lucile Packard Foundation Fellowship in Science and Engineering, by a Sloan Foundation Fellowship, and by an NSF Young Investigator award.

tailed transaction data. The `customer` and `supplier` tables contain information about customers and suppliers, respectively. The `supplies` table contains information about items supplied to a store by each supplier. The following materialized views are built.

- `CustReturns`: Defined as the join between `customer` and `sales` transactions that are marked as returns. This table may be queried by stores when processing a return and needs to be current.
- `TotalItemReturns`: Total amount and number of returns for each item (aggregate over `CustReturns`). This view is used for decision support.
- `LargeSales`: Customers who have made single purchases of more than $1,000 (join between `customer` and `sales`). This view is used for decision support, marketing, and promotions.
- `ItemStoreStock`: For each item and store, the total number of items in stock in the store (join over aggregates over `sales` and `supplies`). This is used to trigger re-stocking decisions and is queried frequently.
- `ItemSuppSales`: Total sales for each item, supplier pair (aggregate over join of `sales`, `supplies` and `supplier`). The view is used for decision support.
- `ItemProfits`: Contains the total profits for each item category (aggregate over join of `ItemSuppSales` and `supplies`). This view is also used for decision support.

The views can be defined in SQL, but the definitions are not important for our purpose. Let us consider the desired maintenance policies for each view. `CustReturns` should provide up-to-date results since it is used for making return decisions. However, queries to this view are likely to be relatively infrequent, and the view could be maintained only when it is queried (deferred maintenance). `TotalItemReturns` and `LargeSales` are used for decision-support and marketing, need a stable version of data, and can be maintained periodically (snapshot maintenance). `ItemStoreStock` is monitored frequently and is used to trigger re-stocking decisions. It thus needs to be maintained using an immediate maintenance policy. `ItemSuppSales` and `ItemProfits` are used for decision support, and can be maintained periodically—say once a day.

■

The choice of a policy for each view is not as straightforward as the above example may indicate. If the total system throughput is of concern, converting an immediate view into a deferred view can sometimes help. For example, it may turn out that the immediate maintenance of `ItemStoreStock` is too expensive, and throughput can be made acceptable by doing deferred maintenance instead. Also, the maintenance policy of one view cannot be chosen independently of the policies of related views. For example, if two views have snapshot maintenance policies but with different refresh cycles and these views are used to derive a third view, this third view can reflect an inconsistent state of the world.

Most of the research on materialized views has focused on high level incremental algorithms for updating materialized views efficiently when the base tables are updated [BC79, SI84, BLT86, QW91, CW91, GMS93, GL95, LMSS95]. Efficient data structures for supporting incremental view maintenance in the presence of multiple views were described in [KR87, SP89, Rou91]. However, global issues in supporting many views with different policies have not been explored in detail.

Summary of Contributions: This paper investigates a number of issues related to supporting multiple views with different maintenance policies. The maintenance policies that are chosen for views have implications on the validity of the results of queries and affect the performance of queries and updates. In designing an application based on views with different maintenance schemes, it is important that the designer be aware of the consequences of the maintenance policy selections on the correctness and performance of the application. To this end, we study both consistency and performance issues related to supporting many views with different maintenance policies.

We develop formal notions of consistency for views with *immediate*, *deferred*, and *snapshot* maintenance policies. In order to provide consistent information in a system allowing multiple views with different maintenance policies, we develop a model based on the notion of viewgroups. A viewgroup is a collection of views that are required to be mutually consistent. We show that certain combinations of the different maintenance policies and viewgroups cannot guarantee consistency or are otherwise not meaningful. We identify the legal combinations, and give an algorithm to maintain the views in a manner that guarantees consistency. The viewgroup model is also important in bounding the amount of computation resulting from any given transaction by confining the effects of the transaction to a portion of the database.

In order to understand the impact of different maintenance policies on the performance of queries and updates, we conduct a detailed performance analysis based on an actual implementation. The trade-offs between different maintenance policy assignments are studied by measuring refresh times, update times, query response times, total elapsed times and number of refreshes. The experimental study gives a very good understanding of the costs of each maintenance policy, and shows that an application can gain performance advantages by using multiple maintenance policies.

Paper Outline: We develop our materialized view model for supporting multiple views with different maintenance policies in Section 2. The overall architecture and algorithms for supporting view maintenance in such a model is described in Section 3. An overview of our implementation goals and methodology is given in Section 4. Section 5 gives results from performance experiments. Sections 6 and 7 describe related work and conclusions, respectively.

2 A Model for Supporting Materialized Views

In this section, we investigate the impact on the validity of query results of allowing views with different maintenance policies. An important issue when defining multiple maintenance policies is consistency between different materialized views. We will define consistency formally in Section 2.1. For now, let us informally review the consistency expectations for each maintenance policy.

Immediate views have to be consistent with the tables they are defined over, as they exist in the current state. Deferred views need not be consistent, but queries over deferred views have to be answered as if views are consistent, and consistency is typically achieved by making the deferred view consistent at query time. The process of re-establishing consistency is called *refresh*. Snapshot views are required to be consistent with the state of the deriving tables that existed at the time of the last refresh. When we use views to define other views, and have different consistency requirements for these, it is easy to get confused about what state each materialized view is meant to represent. Note that the consistency discussion here is on the desired relationship between states of materialized views and base tables in a central system with one user. A related problem of concurrency control, to ensure that consistency is achieved in the presence of concurrent transactions, is discussed in [KLM+97]. Notions of consistency for views in distributed environments have been presented in [ZGHW95] and [HZ96].

To identify related views that should be consistent with each other, and to localize the maintenance activity within a small set of views, we place base and view tables into *viewgroups*. A viewgroup consists of a set of tables, along with logs holding the changes to the tables. A viewgroup should have the property that there is some past (or current) state s_j of base tables such that all views in the viewgroup are consistent with the state s_j of the base tables. In other words, the result of evaluating a query over any materialized view in a viewgroup must be the same as evaluating the equivalent query over base tables in state s_j. Since this state is common to all views in the viewgroup, a query that involves multiple views in the viewgroup will return a consistent answer. In addition, viewgroups should be isolated in the sense that (1) maintenance of a view in a viewgroup should not trigger maintenance or updates in another viewgroup, and (2) it should be possible to answer a query without looking outside the queried viewgroup.

In the rest of this section, we formally define consistency requirements, specify the properties that we want views and viewgroups to satisfy, and develop a scheme for assigning maintenance policies to views, and for assigning views to viewgroups.

2.1 Notation and Consistency

A table that is not defined as a view is called a *base table*. A *materialized view* V is a table that is defined through a query Q over some set of tables $\{R_1, ..., R_k\}$, denoted as

$$V \stackrel{def}{=} Q(R_1, ..., R_k).$$

$R_1, \ldots, R_k$ are called the *referenced* (or parent) tables of V, and may be base tables or materialized view tables. A materialized view is labeled as *immediate*, *deferred*, or *snapshot* to indicate the maintenance policy. A *virtual view* is a view that is not materialized. For simplicity of presentation, we do not discuss virtual views in the rest of this paper.

Definition 2.1 (View-Dependency Graph:): The dependency graph G of a view V is a graph with a node for each table referenced in the view definition, a node labeled V, and a directed edge from the node for each referenced table to the node for V. The dependency graph shows how the view is derived from base tables and/or other views.

The view-dependency graph $\mathcal{G}$ of a database schema is the union of the dependency graphs for all the views in the schema. The view-dependency graph shows how all the views in the schema are derived from each other and from base tables. ■

EXAMPLE 2.1 View-Dependency Graph: Figure 1 illustrates a view-dependency graph. The B_i are base tables, and the V_j are views. Ignore the dashed lines and their labels for now. V_8 is a view that may be defined with a SQL statement such as "`CREATE VIEW` V_8 `AS (SELECT` $\star$ `FROM` B_3, V_7 `WHERE` *Predicate*)". B_3 and V_7 are the referenced tables. The edge from V_7 to V_8 means that V_7 is referenced in the definition of view V_8. We say that V_7 is a *parent* of V_8, and that V_8 is a *child* of V_7. In this formulation, ancestors of a view V represent tables used to derive view V. Descendants of V represent views derived from V. ■

States, Values, and Materializations: Informally, the *value* of a table R is the bag of tuples that must be returned for the query "`SELECT` $\star$ `FROM` R". In other words, the value of a table is the *observed result* of querying the table. The *materialization* of a table R is the bag of tuples that are actually stored in the database for the table.

A database *state* s is a mapping from each (base and view) table R in the database schema to the value of R, denoted $R(s)$, and the materialization of R, denoted $R_M(s)$.

$$s : R \rightarrow (R(s), R_M(s))$$

The current database state is denoted by s_{curr}. We assume that states are temporally ordered.

For base tables, the value $R(s)$ is equal to the bag of tuples stored for relation R (*i.e.* the materialization $R_M(s)$). For immediate and snapshot views, the value $R(s)$ is required to be equal to the materialization $R_M(s)$. (For snapshot views, both $R(s)$ and $R_M(s)$ remain unchanged as s varies, until s reaches a state in which a new snapshot is taken.) For deferred views, the materialization and value can be different. For a deferred view $V \stackrel{def}{=} Q(R_1, ..., R_k)$, the value $V(s)$ is the collection of tuples that we want to be returned by the query "`SELECT` $\star$ `FROM` V". Since a deferred view is expected to be maintained at the time of querying, the value $V(s)$ of a deferred view can be formally defined as: $V(s) = Q(R_1(s), ..., R_k(s))$, where $R_i(s)$ is the value of the referenced table R_i in state s. The materialization $V_M(s)$ can be different from the value $V(s)$ in most states, but refresh would be required to make the two equal at the time of querying.

We now define consistency between a view table, its parents, and the base tables.

Definition 2.2 (Reference and Base Consistency): Let $V \stackrel{def}{=} Q(R_1, \ldots, R_k)$ be a materialized view, with $R_1, \ldots, R_k$ as the referenced tables. Let $\{B_1, \ldots, B_m\}$ be the base tables from which V is ultimately derived. In each current state s_{curr}, the view V is said to be *reference consistent* with state s_j if $V(s_{curr}) = Q(R_1(s_j), \ldots, R_k(s_j))$. Similarly, the materialization of view V is said to be *reference consistent* with state s_j ($s_j \leq s_{curr}$) if $V_M(s_{curr}) = Q(R_1(s_j), \ldots, R_k(s_j))$.

Let Q' be the query obtained from query Q by substituting each referenced view by its defining query until all the referenced tables are base tables. In state s_{curr}, the view table V is said to be *base consistent* with state s_j if $V(s_{curr}) = Q'(B_1(s_j), \ldots, B_m(s_j))$. Similarly, the materialization of view V is said to be *base consistent* with state s_j ($s_j \leq s_{curr}$) if $V_M(s_{curr}) = Q'(B_1(s_j), \ldots, B_m(s_j))$.

By definition, base tables are base and reference consistent with the current state. ■

From Definition 2.2 and the definition of value $V(s)$ for deferred views, it follows that a deferred view is always reference consistent with the current state. However, whether

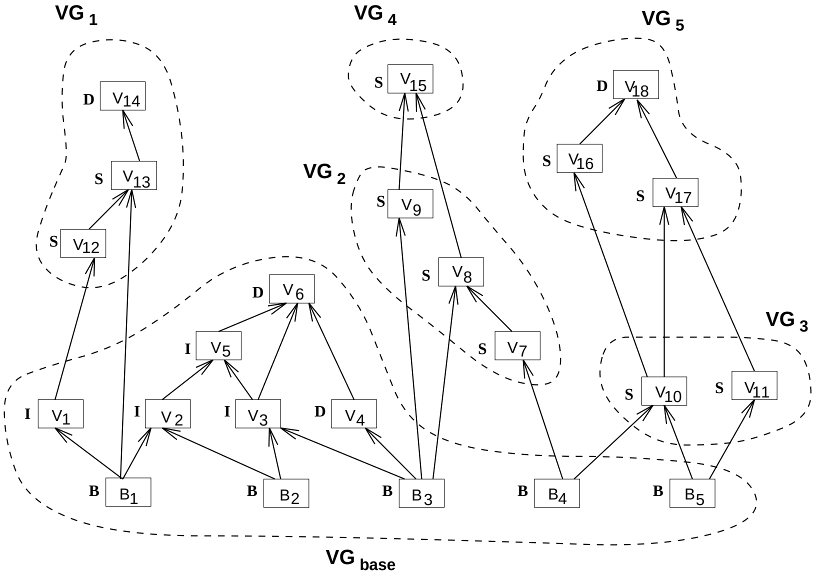


Figure 1: An example of a view-dependency graph

or not V is base consistent with any state at all depends on whether or not its parents are base consistent.

In any given state s_{curr}, there can be several database states with which a view is base consistent. For example, suppose that a view is defined as a (duplicate preserving) projection over a base table. Now suppose that the view is base consistent with the current state and that an update is made to a tuple of the base table on a column that is not included in the view definition. The view will be base consistent with both the old and new states. However, there is a unique "intended" state, s, with which a view is base consistent. We will call this the *effective state* of the view (in state s_{curr}), denoted $\mathsf{effective}(V, s_{curr})$. This is, essentially, the state of the base tables that was propagated into the view by the last refresh operation. In Section 3 we describe how this state is determined.

2.2 Views and Viewgroups

We now state the consistency requirements for views and characterize the different maintenance policies:

Properties of Materialized Views:
Let $V \stackrel{def}{=} Q(R_1, ..., R_k)$ be a materialized view. Then, in any state s_{curr}, the view V must satisfy the following properties:

P1 If V is an immediate view then V is reference consistent with s_{curr}. Further, the materialization of V is equal to the value of V (i.e., $V_M(s_{curr}) = V(s_{curr})$).

P2 If V is a deferred view, then V is reference consistent with s_{curr}, and there exists a past state s_j ($s_j \leq s_{curr}$) such that the materialization of V is reference consistent with state s_j. Further, immediately after any query, the materialization of V must be equal to the value of V.

P3 If V is a snapshot view then there exists a past state s_j ($s_j \leq s_{curr}$) such that V is reference consistent with state s_j. Further, the materialization of V is equal to the value of V.

P4 There exist states s_i and s_j, such that $s_i \leq s_j \leq s_{curr}$, s_i is the effective state of V, i.e., $s_i = \mathsf{effective}(V, s_{curr})$, V is base consistent with state s_i, and V is reference consistent with state s_j.

P5 (a) There exist states s_j, $s_{j_1}, \ldots, s_{j_k}$, such that $s_j \leq s_{j_1} \leq s_{curr}, \ldots, s_j \leq s_{j_k} \leq s_{curr}$, the materialization of V is reference consistent with state s_j, and the materializations of $R_1, \ldots, R_k$ are reference consistent with states $s_{j_1}, \ldots, s_{j_k}$ respectively. (b) For any two states, $s_i < s_k$, $\mathsf{effective}(V, s_i) \leq \mathsf{effective}(V, s_k)$.

The first three properties characterize the different view maintenance policies in terms of reference consistency. The fourth property says that the view must also be consistent with some previous state of the base tables (the effective state) from which the view is ultimately derived. The fifth property requires that the materialization of a view follow those of its referenced tables and that effective states can only progress forward. In other words, we cannot skip refreshing one view before refreshing its descendants and a refresh cannot make a view more stale in a new state.

Viewgroup: A viewgroup $\mathcal{VG}$ is a subset of nodes in the view-dependency graph that represents closely related views satisfying the following properties:

Properties of Viewgroups:

P6 The set of all viewgroups forms a partition over the set of nodes in the view-dependency graph. Recall that a partition over a set S is a set of subsets $\{S_1, ..., S_p\}$ of S such that (i) for each pair of sets S_i and S_j, $S_i \cap S_j = \phi$ if $i \neq j$, (ii) each $S_i \neq \phi$, and (iii) $\bigcup_{i=1}^{p} S_i = S$.

P7 A query on a view in a viewgroup $\mathcal{VG}$ can be answered without querying any other viewgroup.

P8 Maintenance of a view in a viewgroup $\mathcal{VG}$ cannot be triggered by any operation in another viewgroup.

P9 (Viewgroup Consistency): In any state s_{curr}, for each viewgroup $\mathcal{VG}$, there exist past database states s_i

and s_j such that $s_i \leq s_j \leq s_{curr}$, all the views $V \in \mathcal{VG}$ are reference consistent with state s_j and are base consistent with state s_i, and $s_i = \mathsf{effective}(V, s_{curr})$ for all $V \in \mathcal{VG}$. The state s_i is also called the effective state of $\mathcal{VG}$ and is denoted as $\mathsf{effective}(\mathcal{VG}, s_{curr})$.

Properties **P7** and **P8** ensure that viewgroups identify modules within which we can do maintenance and querying without affecting other modules. The modularization is important for a system to support a large number of views. Property **P9** is important to make sure that queries can access multiple tables within a viewgroup and expect to see data consistent with some state of the system.

2.3 Constraints on Maintenance Policies and Viewgroups:

Satisfying properties **P1-P9** for a schema means that we cannot arbitrarily choose maintenance policies and viewgroups for each view that we define. We list the design rules that constrain the assignment of maintenance policies and viewgroups. See [CKL+96] for a detailed explanation of the reasons for these design rules. The maintenance policy can be represented by a label (I for immediate, D for deferred, and S for snapshot) on each node of the view-dependency graph, and a viewgroup can be represented by a set of nodes from the view-dependency graph. The design rules can thus be described using the view-dependency graph.

Rule 1 *Each view can be assigned to exactly one viewgroup.*

Rule 2 *Immediate and deferred views must belong to the same viewgroup as their parents.*

Rule 3 *All snapshot views in a viewgroup must have the same refresh cycle.*

Rule 4 *Snapshot views cannot occur in the same viewgroup as base tables.*

Rule 5 *A viewgroup can be derived from at most one other viewgroup.*

Rule 6 *Deferred views cannot have children in other viewgroups.*

In addition to the above design rules required to satisfy properties **P1-P9**, certain assignments of maintenance policies are semantically equivalent to more easily understood assignments. We choose to disallow such assignments.

Rule 7 *An immediate view cannot have a deferred view or a snapshot view as a parent.*

Rule 8 *A snapshot view cannot have a deferred view as a parent.*

Rule 9 *All base tables must be placed in a single viewgroup. (We will call this viewgroup $\mathcal{VG}_{base}$.)*

From **Rules 1-9** we can further infer that (1) all immediate views must be placed in the base viewgroup $\mathcal{VG}_{base}$, and (2) the viewgroup graph is a tree rooted at $\mathcal{VG}_{base}$. The viewgroup graph has a node for each viewgroup, and an edge from viewgroup $\mathcal{VG}_1$ to $\mathcal{VG}_2$ if there is an edge in the view-dependency graph from a view in viewgroup $\mathcal{VG}_1$ to a view in viewgroup $\mathcal{VG}_2$.

Figure 1 shows an example of a legal assignment of views to viewgroups and maintenance policies. The dashed lines represent viewgroups. The labels `I`, `D` and `S` denote immediate, deferred and snapshot views, respectively.

EXAMPLE 2.2 Consider the views of Example 1.1. The maintenance policy assignments of all views except one satisfy **Rules 1-9**. However, having `CustReturns` as a deferred view violates **Rule 8**, since a snapshot view `TotalItemReturns` is defined over `CustReturns`. We have three options for getting a legal assignment: (1) make the view `CustReturns` an immediate view (if we want it to always contain current data), or (2) make `CustReturns` a snapshot view (if we can tolerate old data), or (3) redefine `TotalItemReturns` as an aggregate over the join of `customer` and `sales`. ■

2.4 Declaring Views and Viewgroups

A user creates a viewgroup with a Data Definition Language (DDL) statement indicating the name of the viewgroup and its refresh frequency. The base viewgroup is built-in, and all base tables are automatically placed in the base viewgroup. Each materialized view is declared with its maintenance policy. Each snapshot view must be assigned to a viewgroup by the user; all views in a viewgroup are refreshed together. An immediate or deferred view need not be explicitly assigned to the viewgroup: It is assigned to the viewgroup of its parent tables automatically by the system (**Rule 2**). The system checks the view definition, maintenance policy assignment, and the viewgroup assignment to ensure that **Rules 1-9** are satisfied. If not, the view definition is rejected. Finally, each view is materialized upon creation.

3 Maintaining Views and Viewgroups

So far, we have introduced the concept of a viewgroup, and described a model for ensuring the consistency of multiple materialized views. In this section, we describe in more detail how view maintenance takes place in our implementation, and how the "effective state" of a viewgroup is chosen.

3.1 View Maintenance Architecture

View maintenance requires additional work to be done in response to updates. When a table has an immediate view defined on it, the extra work that has to be done in response to an update on the table is refreshing the immediate view. If a table has a deferred or a snapshot view, the extra work is recording the updates to the table in logs.

We maintain a special log per table for the explicit purpose of view maintenance. The log data structures and algorithms used to interface with the logs are realized in the Ode database system. Section 4 describes our storage model and implementation.

The function that takes an update transaction and executes the transaction as well as the required additional work is called `makesafe`, as in [CGL+96], and the function that changes the view is called `refresh`. Figure 2 illustrates our implementation architecture.

3.2 Algorithms for Maintaining Consistency

There are three types of events that can cause changes to the contents of a view; (a) updates to a base table, (b) a query involving a deferred view, and (c) explicit or periodic refresh of a viewgroup containing snapshot views. (Additions and deletions of views are not discussed here; however, it must be noted that a view is materialized when it is defined.) We summarize the actions to be taken on each of these events.

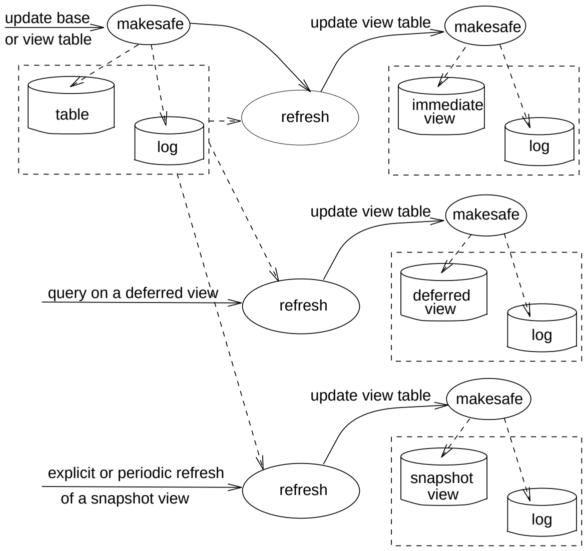


Figure 2: View maintenance architecture

An update to a table will result in a call to `makesafe` (Algorithm 3.1) which, in addition to the table update, updates logs (if the table has any deferred or snapshot views defined on it), and refreshes any immediate views defined on the table.

Algorithm 3.1 (makesafe)
Input: Update command U.
Result: Consistent database after execution of U.
Method:
if U is the update of base tables
  for each table T to be updated **do**
    update T;
    update logs if T has `deferred` or `snapshot` children;
  endfor
  $B \leftarrow$ updated base tables;
  done $\leftarrow$ false;
  $\mathcal{D} \leftarrow$ views in $\mathcal{VG}_{base}$ that are descendants of B;
  while done is false **do**
    $D \leftarrow$ some `immediate` view in $\mathcal{D}$ whose parents are not in $\mathcal{D}$;
    if no such D
      done $\leftarrow$ true;
    else
      refresh(D); remove D from $\mathcal{D}$;
  endwhile
else /* U is the update of a view V */
  update V;
  update logs if V has `deferred` or `snapshot` children;
◇

Algorithm 3.2 (refresh)
Input: View V to be refreshed.
Result: V is refreshed and its logs are updated.
Method:
  compute incremental changes to V;
  generate update command U to update V;
  makesafe(U);
◇

The `refresh-group` operation (Algorithm 3.3) is called on a (non-base) viewgroup in response to an explicit request by the user to refresh the viewgroup, or due to an asynchronous refresh of the viewgroup at the assigned refresh interval. The result of the operation is a refresh of all snapshot views in the viewgroup. A `refresh-group` operation proceeds in a breadth-first manner where a view is refreshed only after all of its parent views (that need to be refreshed) are refreshed.

Algorithm 3.3 (refresh-group)
Input: Viewgroup $\mathcal{VG}$ to be refreshed.
Result: All snapshot views in the group are refreshed.
Method:
  $B \leftarrow$ views in parent viewgroup from which views in $\mathcal{VG}$ are derived;
  done $\leftarrow$ false;
  $\mathcal{D} \leftarrow$ views in $\mathcal{VG}$ that are descendants of B;
  while done is false **do**
    $D \leftarrow$ some `snapshot` view in $\mathcal{D}$ whose parents are not in $\mathcal{D}$;
    if no such D
      done $\leftarrow$ true;
    else
      refresh(D); remove D from $\mathcal{D}$;
  endwhile
◇

A query on a deferred view will result in a call to the procedure `refresh-deferred-view` (Algorithm 3.4). The algorithm first checks if the view needs to be refreshed, to prevent multiple deferred views with a common deferred parent from causing multiple refreshes of the parent, as well as to avoid refreshing a view for which the underlying data has not changed. The test to determine if a view needs to be refreshed is described in Section 4, If the view needs to be refreshed, the procedure `refresh-deferred-view` recursively calls itself on each deferred parent view, and then refreshes the view.

Algorithm 3.4 (refresh-deferred-view)
Input: Deferred view V
Result: Recursive refresh of deferred views subtree.
Method:
    if V needs to be refreshed
  for each parent A of V **do**
    if view-type of A is `deferred`
      refresh-deferred-view(A);
  endfor
  refresh(V);
◇

3.3 Effective State and Correctness

In Section 2.2, we presented various consistency requirements for views and viewgroups. There can be several database states with which a view is base consistent. However, there must exist some unique "intended" state with which a view or viewgroup is base consistent. This state is called the effective state, and it must move forward in time. The effective state is determined as follows.

1. The effective state of $\mathcal{VG}_{base}$ is s_{curr}.

2. The effective state of any other viewgroup $\mathcal{VG}$ is the effective state of its parent viewgroup $\mathcal{VG}'$ in state s', i.e., $\mathsf{effective}(\mathcal{VG}, s_{curr}) = \mathsf{effective}(\mathcal{VG}', s')$, where s' is the state in which the last refresh-group of $\mathcal{VG}$ was performed.

In other words, the effective state of a viewgroup is the current database state if the viewgroup is the base viewgroup. Otherwise, it is the effective state of the parent group that was "propagated" into the viewgroup by the most recent `refresh-group` operation.

We are now ready to state the following (a formal proof will be given in the full paper):

Consider a system permitting materialized views and viewgroups to be defined according to the design rules of Section 2.3 and using the maintenance functions described in Section 3.2. Let the effective state be determined as above. Then,

- A non-base viewgroup is reference consistent with the state in which it was last refreshed.
- Properties **P1-P9** are satisfied.

4 Implementation Goals and Overview

The focus of the previous two sections was on the modeling aspects of view maintenance. The viewgroup model is important not only for defining and achieving consistency of queries but also for containing the amount of computation that is done by any given transaction. The focus of the next section is on the performance aspects of view maintenance. In this section, we give an overview of the implementation on which the experimental study was based.

When implementing our view maintenance and viewgroup algorithms, we had a number of goals in mind. We wanted to be able to answer the following questions:

Feasibility: Is it feasible to support multiple maintenance policies in a single system?

Scalability: Can multiple views be materialized and maintained in scalable manner?

Performance: Can an application gain a performance advantage by using multiple maintenance policies?

Tradeoffs: What are the performance trade-offs between different maintenance policies, and between incremental maintenance and recomputation itself?

Cost: How expensive is each maintenance policy for multiple views?

In this section, we highlight some of the important aspects of the implementation. We chose the Ode database system [AG89] as the implementation vehicle for materialized views. This choice was made since we have expertise and access to the source code for Ode, allowing us to experiment with special data structures for storing logs. Ode is an object-oriented database; its data manipulation language is O++, a variant of C++ with persistence and transactions. It also offers relational features through the SWORD interface [MRS93]. The ideas used in the implementation are equally applicable to commercial relational systems (e.g. Oracle, Sybase).

Tables: A table (base or materialized view) is realized by a *collection* class in Ode. A collection instantiation returns a *descriptor* (or handle) that is used to reference a *materialization* of tuples. A tuple is an Ode object. The materialization has a cluster of active tuples, representing the current state of the table, and a cluster of inactive tuples, that can be used to compute the pre-update state of the table. The `insert()` function creates a new tuple with the given values, and inserts it into the active cluster. The `remove()` function removes a tuple from the active cluster, and places it in the inactive cluster. The inactive tuple must stay in the inactive cluster until the effect of its removal is propagated to all views defined on the table, after which it can be garbage collected. The `replace()` function updates an existing tuple, and stores the pre-update values in a newly created inactive tuple. Index structures can be built on the tuples in the active cluster.

Log Data Structure: A *log* is maintained for each table that has a deferred or snapshot view defined over it. The log is physically independent of the table materialization, and there is only one log for each table regardless of the number of the views it derives. The problem of maintaining a single log for a table that has multiple views (with different refresh times) defined on it was also considered in [SP89], [KR87], and [CG96]. Our log structures are based on the general principles presented in those papers.

We create one log entry per update operation. Each log entry has an operation flag and the *oid*[1] of the tuple in the materialization to which the operation was applied. For `replace()`, the log entry contains a second *oid* pointing to the pre-update value of the replaced tuple in the inactive cluster. We also have a separate table containing a collection of tuples, each of which represents an edge of the view dependency graph. These edge tuples are added when a new (materialized) view is defined in the system, and are modified every time a view is maintained. An edge tuple for a pair (B, V) contains a pointer to the last log entry for B that was used to maintain V (if V is a deferred or snapshot view).

Computing Incremental Changes: The implemented incremental maintenance algorithm is based on the counting algorithm of [BLT86, GMS93]. The algorithm is slightly enhanced to incorporate replace operations. To use the counting algorithm efficiently, we found it important to be able to access both the pre- and post-update states of the tables [CM96, HZ96], and to avoid redundant computations caused by deletions when maintaining a join of two or more deriving tables. The *net* base table changes are determined by iterating through the portion of the log containing the relevant changes and collapsing and/or eliminating redundant changes (such as pairs of inserts and deletes).

The `makesafe` operation (Algorithm 3.1) traverses a view dependency graph to refresh all immediate views in a topological order. Since the cost of computing the topological order is non-negligible (about one second per refresh operation in the preliminary experiments), we *pre-compute* the topological order whenever an immediate view is added (deleted) to (from) the system, and store it in the descriptor of the base table. This way `makesafe` will refresh immediate views by simply following the pre-computed order.

Quick Maintenance Check: The deferred view maintenance policy requires checking whether a view needs to be refreshed before each query on the view. However, the cost of testing whether a view needs to be refreshed is non-negligible. To minimize this cost, we designed a quick main-

[1] A `ROWID` would be used instead of an *oid* in an implementation on a relational system.

tenance check for deferred views with a simple time-stamp technique.

In each viewgroup, a deferred view is rooted at a subtree containing only other deferred views as internal nodes and snapshot, immediate, or base tables as leaves (with the arrows in the reverse of the direction in the view dependency graph). For each deferred view, we store pointers to the leaves of its deferred view subtree. The `makesafe` algorithm is extended to place a time-stamp on the table being updated. In `refresh-deferred-view` (Algorithm 3.4), the deferred view's time-stamp and the time-stamps of the leaf tables (described above) are compared to detect if any one of the leaves has a more recent time-stamp.

5 Performance Study

This section describes an experimental performance study on top of the disk-based Ode<EOS> database system. The experiments compare the performance of the different maintenance policies in the presence of multiple materialized views. We first investigate the performance of the incremental refresh algorithm against full recomputation. We then compare the behavior of the different maintenance policies in response to different transaction mixes. Lastly, we verify that the implementation supports multiple views with different maintenance policies in a scalable manner.

All experiments were run in single user mode on a Sun UltraSparc 64 MB RAM machine running Solaris 2.5. The database was kept on a local 4.2 GB SPARC storage UniPack disk drive attachment to eliminate NFS delays.

5.1 Experimental Setup

We build two databases containing base tables and materialized views, run 1,000 transactions against each database, and gather various statistics (e.g., total elapsed time, average query response time, average update time) for the set of 1,000 transactions.

Databases: Figure 3 illustrates the base tables and views in one of the two experimental databases. The schema has four base tables and seven views. Tuples in the base tables are 300 bytes long; view tuples are 50 bytes long. Two of the materialized views (P-view1 and P-view2) are Projection views, three (SPJ-view1, SPJ-view2, and SPJ-view3) are Select-Project-Join views over the base tables, and the last two (SPJ-view12 and SPJ-view23) are SPJ views over the view tables, as illustrated in the figure.

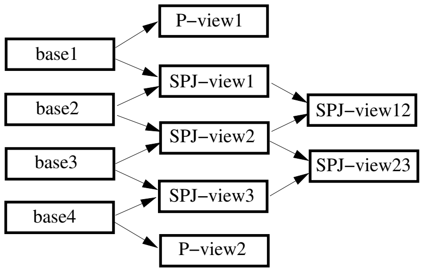


Figure 3: Experimental database schema

Before each experiment, each base table is initialized with 100,000 tuples of uniformly-distributed and randomly generated data, and each view is materialized. (Experiments with 1,000,000 tuples per base table produced similar results.) For each base table, join column values are randomly chosen from the domain (0, 100,000). Thus, each SPJ view is roughly the size of a base table (e.g., 100,000 tuples). B+tree indices are built on the join attributes of all base tables and views. These indices improve both query and (incremental) view maintenance performance.[2]

Transactions: A program produces a stream of transactions, each of which either queries or updates the database. A *query transaction* contains only `display` operations on a randomly chosen view, while an *update transaction* contains either `insert`, `remove`, or `replace` operations on a randomly chosen base table. The `replace` operation updates non-indexed fields of the chosen base table. The *update ratio* of the transaction stream is the number of update transactions divided by the total number of transactions in the stream. For instance, if the stream contains 750 updates and 250 queries, then the update ratio is 0.75.

Each transaction contains between one and eight operations over the same table. Thus, a query transaction reads between one and eight tuples matching randomly chosen values from a single view table. We run experiments with nine different update ratios (0, 0.05, 0.1, 0.25, 0.5, 0.75, 0.9, 0.95, 1), and measured the total response time, the average query/update response time, and other statistics.

5.2 Performance of Incremental Maintenance

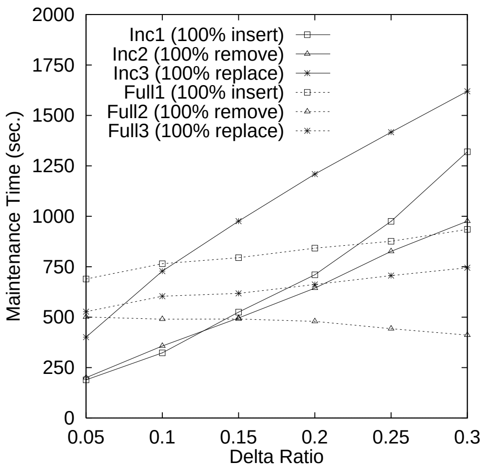
The first set of experiments investigates the basic performance aspects of the incremental maintenance algorithm.

Purpose of experiment: It is well-known that incremental refresh outperforms full refresh if the size of the incremental change set (delta) is relatively small; however, what "small" is depends on the implementation logic of refresh, the nature of the updates, and also on the complexity of query definitions. Here, we wish to investigate the performance of incremental refresh versus full recomputation for an SPJ view under different types of updates (insertions, deletions, or replacements).

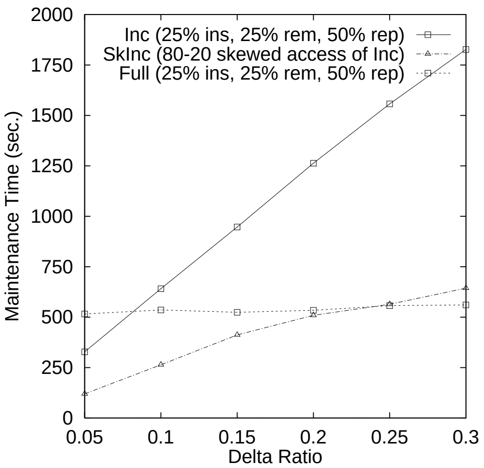
Method of experiment: We use only a subset of the schema of Figure 3 (base1, base2, and the SPJ-view1) for this experiment, and execute only update transactions (i.e., update ratio = 1.0). Both base tables are updated uniformly, though we choose several different types of update transaction sets such as `insert` only, `remove` only, `replace` only, and a mixture of these operations. We also execute a transaction set with skewed access that localizes 80% of the updates to 20% of the base tuples.

We define the *delta ratio* as the fraction of base tuples that are updated (number of updated tuples divided by total number of tuples in base1 and base2 tables). We measure the time taken to maintain the view at different values of the delta ratio using (1) Incremental maintenance (labeled `Inc` in Figure 4), and (2) Full recomputation (labeled `Full` in Figure 4). A full recomputation re-materializes the view by discarding the old contents, recomputing the view tuples, and rebuilding indices on them.

[2] An index on a view degrades the performance of full recomputation, since the old index contents are thrown away, and a new index must be built.



(a) Incremental maintenance vs. full recomputation



(b) Effect of skew on incremental maintenance

Figure 4: Incremental and full refresh response time at various delta rates

Analysis: Graph (a) in Figure 4 shows the performance of the incremental maintenance for join views:

1. When only `inserts` into the base tables occur (`Inc1` and `Full1` in the graph), incremental refresh is superior to full refresh until each base table has become about 23% larger (i.e., after 46,000 insertions total).
2. When only `removes` from the base tables occur (`Inc2` and `Full2` in the graph), incremental refresh remains better until about 15% of the tuples are removed from each base table (i.e., 30,000 deletions total).
3. When only `replaces` to the base tables occur (`Inc3` and `Full3` in the graph), incremental refresh performs better only up to around 7% replacement in each base table (i.e., 14,000 tuple replacements total).

Graph (b) in Figure 4 illustrates the benefit of skewed updates on the performance of incremental maintenance. The line labeled `Inc` gives the incremental maintenance time for a transaction stream with 25% inserts, 25% removes, and 50% replaces uniformly distributed across the data set. The line labeled `SkInc` gives the incremental maintenance time for a transaction stream with a similar mix of operations, but with a skew based on the 80-20 rule – 80% of the transactions update 20% of the data. The third line labeled `Full` gives the full recomputation time, which is the same for both uniform and skewed updates. We see that incremental maintenance performs much better when the updates are skewed, outperforming full recomputation until 24% of the data has been updated. When updates are skewed, the same base tuples may be updated several times, and log trimming reduces these several updates into a single update. Consequently, incremental maintenance works with a smaller number of net changes and is more efficient compared to a case where updates are uniformly distributed.

5.3 Comparison of Maintenance Policies

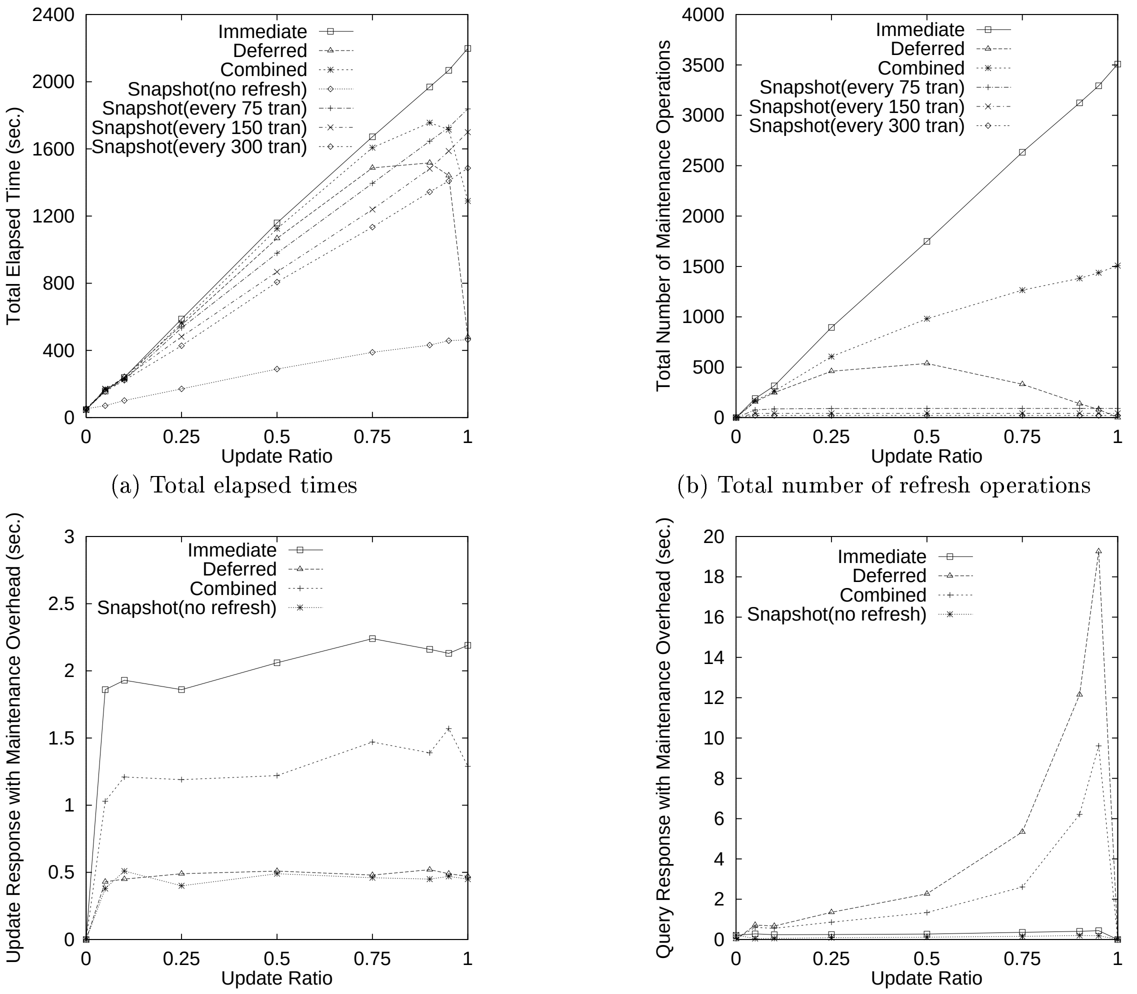
In this subsection, we compare the performance of different maintenance policies in the presence of multiple views.

Purpose of experiment: We wish to see the performance characteristics of each view refresh policy under a variety of update and query ratios. We expect to demonstrate that deferred maintenance leads to faster update times and slower query times than immediate maintenance. We would also like to determine which policy leads to a better overall elapsed time.

Method of experiment: We experiment with the following assignments of maintenance policies: (1) `Immediate`, (2) `Deferred`, (3) `Snapshot(No Refresh)` meaning that all views are snapshots, but are not refreshed during the experiment, (4) `Snapshot(75 Tran)`, `Snapshot(150 Tran)`, and `Snapshot(300 Tran)`, meaning that all views are snapshots, and are refreshed after every 75th, 150th and 300th transaction respectively,[3] and (5) `Combined`, meaning that the views SPJ-view12 and SPJ-view23 use the deferred maintenance policy, while all other (first layer) views use the immediate maintenance policy. For each maintenance policy, we vary the update ratio of the transaction stream, and measure the elapsed time, the number of refresh operations, the update response time, and the query response time.

Analysis: Graph (a) in Figure 5 shows the elapsed times to complete 1,000 transactions under different maintenance policies. The line for `Snapshot (no refresh)` gives the time required to execute the transactions without any maintenance overhead – the elapsed time increases with the update ratio simply because updates are more expensive than queries. A major observation is that the immediate maintenance policy's cost increases linearly as the update ratio increases, while the deferred and combined maintenance policies show smaller elapsed times at higher update rates. Note that the sudden drop in the total elapsed time at update ratio 1.0 for the deferred policy is due to the absence of queries (so that maintenance never occurs). For snapshot

[3] Note that we chose 75, 150, and 300 because they do not evenly divide 1,000. Otherwise, these policies would have to do maintenance for the 1,000th transaction. This would penalize snapshot relative to deferred, since snapshot would maintain all views, while deferred would be able to leave some views unmaintained.



(a) Total elapsed times

(b) Total number of refresh operations

(c) Update response times

(d) Query response times

Figure 5: Comparison of different maintenance policies

policies, the larger the update interval, the faster the snapshot maintenance policy runs since (1) some of the overheads are constant, and (2) log trimming is more effective with a larger number of log records.

Graph (b) in Figure 5 shows that the number of maintenance operations goes up linearly with the number of updates under immediate maintenance. The number of maintenance operations is roughly semi-elliptic for the deferred policy. When the number of read transactions goes down, so does the number of refreshes past a certain point (reaching zero when there are no read operations, since then the view is never maintained). As expected, the combined behavior is between that of immediate and deferred.

Graphs (c) and (d) respectively show the update and query response times, including the time to maintain the views when required. Graph (c) clearly shows that using the immediate policy significantly increases the cost of update transactions. On the other hand, Graph (d) shows that the average query response time for snapshot and immediate policies (both of which involve no maintenance work by queries) is significantly less than that for the deferred policy (which must do maintenance work as part of the query). The combined strategy reduces the query response time by shifting the maintenance work of the first level views to the update transactions — the average query response time goes down by about half.

5.4 Testing Scalability of Log Operations

We wished to verify that the logging overhead, which must be paid by update transactions, is *independent* of the number of deferred/snapshot views, so that the system can scale with increasing numbers of deferred/snapshot views. An experiment conducted by varying the number of views from 0 to 7, and measuring the update response time for different maintenance policies confirmed that the update response time was insensitive to the number of deferred and snapshot views. However, the update response time was found to increase linearly with the number of immediate views.

5.5 A second database schema

The experiments above were repeated with a linear tree shaped database schema (Figure 6). The linear tree schema is somewhat simpler, with the same base tables as in the earlier schema, but with the views defined in a linear tree form. The results were found to be similar to those reported above for the schema of Figure 3.

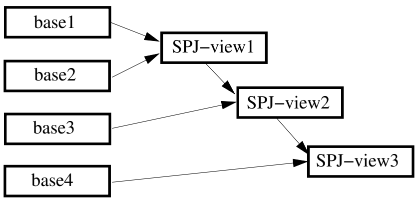


Figure 6: Linear tree schema

6 Related Work

ADMS [RK86, Rou91] was the first system to realize the importance of supporting multiple maintenance policies. However ADMS did not propose any model of consistency in the presence of multiple policies. They concentrated on the advantages of materializing a view using view-caches, which are join indices, rather than the view tuples themselves. An analytical and experimental study comparing the benefits of deferred incremental maintenance over re-computation using different join methods was presented. However, they did not compare the performance of the different maintenance policies. Consistency of views in distributed warehousing environments was studied in [ZGHW95] and [HZ96]. Our focus, on the other hand, is on the consistency aspects of views with different policies in the same system.

Another analytical performance study by [BM90] compared the use of join indices, fully materialized views, and recomputation for querying join views, and found that either method can perform better depending upon the update rate and join selectivity.

[Han87] presents an analytical performance comparison of answering queries over a *single* view by (1) recomputation, (2) an immediately maintained materialized view, and (3) a deferred materialized view. The study is done for a single SP view (recomputation is the best), a single SPJ (select-project-join) view (immediate maintenance is the best), and a single aggregate view (deferred maintenance is the best). In contrast, our performance study uses an actual implementation, and compares the maintenance policies for sets of materialized views. Also, deferred view maintenance is treated somewhat differently in [Han87], where it is assumed that the changes to the base tables are also deferred until the view is queried. An analytical study of optimal refresh policies, based on queuing models and parameterization of response time and cost constraints, is described in [SR88].

As one can see, many issues in the performance analysis of materialized views have yet to be explored. There have been isolated attempts to study certain narrow aspects, and there is a need for a comprehensive performance study using an actual implementation. Our study is a start in this direction, and is the first one to compare the impact of different maintenance policies for different views.

Snapshots were first proposed in [AL80]. Implementation techniques for snapshot views are described in [LHM+86, KR87, SP89]. These papers consider only SP (select-project) views. [LHM+86] focuses on detecting relevant changes to a snapshot based on update tags on base tables. [KR87] and [SP89] present techniques for maintaining logs and computing the net update to a view when multiple views share common parents. The log structures in our implementation are based on the ideas in [SP89]. Replication servers from Oracle and IBM essentially implement snapshot materialized views, where the snapshot view is materialized in a remote system.

Our implementation uses the counting incremental maintenance algorithm of [GMS93] for immediate, deferred, and snapshot maintenance. Several other incremental algorithms have been proposed [BLT86, CW91, QW91, GL95, CGL+96]. In [CGL+96], equations that compute incremental changes to a view using only the post-update state of tables are derived. The paper also proposes a high-level scheme for minimizing the time taken to refresh a view.

Strategies for updating a view based on different priorities for transactions that apply computed updates to a view and transactions that read a view, are presented in [AGK95]. Their experiments focus on that portion of the maintenance procedure that applies previously computed changes to a view.

Concurrency control problems and a serializability model to guarantee serializability in the presence of deferred views are discussed in [KLM+97]. The focus of that paper is on doing concurrency control when multiple transactions reading and updating tables are executing concurrently in the system.

7 Conclusions

In this paper we investigated issues related to supporting multiple materialized views with different maintenance policies. We considered the problem of supporting immediate, deferred, and snapshot materialized views in the same system. This problem is important because the different maintenance policies have different performance characteristics and consistency guarantees, and an application may need a mix of the characteristics for different materialized views. While researchers have looked at supporting each of these policies in isolation, simultaneous support for these has not been investigated in detail. A complete solution to the problem of supporting many views with different maintenance policies must include both correctness and performance aspects.

We first studied the impact on the validity of query results in a system supporting different policies. We developed formal notions of consistency for views with different policies and proposed a model based on viewgroups to collect together related, mutually consistent, views. The viewgroup concept is important in understanding the notion of consistency in the presence of multiple views, and to be able to support consistent querying over materialized views. Viewgroups also help place limits on the potential effects of a read or update transaction by isolating the resulting maintenance activities to a single viewgroup. We presented rules for assigning maintenance policies, and an algorithm to maintain views subject to the constraints.

Next, we studied the performance aspects of a system supporting different maintenance policies based on an actual implementation. The implementation used efficient data structures and algorithms to provide a viable and scalable system for supporting multiple views and policies. We conducted a detailed performance study to compare the tradeoffs between the different policies. We used two different databases for the experiments, and obtained similar results on both, giving us confidence in the soundness of our conclusions. In addition to demonstrating the feasibility and scalability of a system based on our model, we learned the following: (1) Immediate maintenance penalizes update transactions, while deferred maintenance penalizes read transactions. Combining immediate and deferred policies, wherein some views have immediate maintenance and others have deferred maintenance, has a balancing effect. (2) Doing deferred maintenance for two levels (a deferred view defined using another deferred view) degrades performance much more dramatically than when deferring maintenance for one level. (3) The number of levels of immediate and deferred views have more impact on performance than the breadth of the schema. (4) Immediate and deferred maintenance have similar throughput at low update ratios (that is, few updates and many queries). (5) Snapshot maintenance performs better than immediate or deferred maintenance (not surprisingly). (6) Rematerialization is a viable alternative to incremental maintenance once 10-20% of base tuples are changed. The cut-over point occurs more quickly when base tuples are updated than when tuples are inserted into or deleted from the base tables. This observation shows that there is scope for a better maintenance algorithm for updates than the currently published algorithms that treat updates as deletions followed by insertions

The experiments clearly illustrate that the different maintenance policies impact the system differently, and that one may need to choose different policies for different views depending upon the data currency requirements and the performance implications. The results of our investigations on both the consistency and performance aspects of various view maintenance policies are important in choosing the appropriate maintenance policies.

Acknowledgments We thank Robert Arlein for technical assistance in the development of the prototype system. We also thank Damianos Chatziantoniou, Shu-Wie Chen, Matt Greenwood, Pankaj Kulkarni, and Jun Rao, of Columbia University's database research group for valuable comments on an earlier version of this paper. Finally, we are grateful to the referees for their remarks.

References

[AG89] R. Agrawal and N. Gehani. Ode (object database and environment): the language and the data model. In *SIGMOD* 1989.

[AGK95] B. Adelberg, H. Garcia-Molina, and B. Kao. Applying update streams in a soft real-time database system. In *SIGMOD* 1995.

[AKG96] B. Adelberg, B. Kao, and H. Garcia-Molina. Database support for efficiently maintaining derived data. In *EDBT* 1996.

[AL80] M. Adiba and B. Lindsay. Database snapshots. In *VLDB* 1980.

[BC79] P. Buneman and E. Clemons. Efficiently monitoring relational databases. *ACM TODS*, 4(3):368–382, September 1979.

[BLT86] J. Blakeley, P. Larson, and F. Tompa. Efficiently Updating Materialized Views. In *SIGMOD* 1986.

[BM90] J. Blakeley and N. Martin. Join index, materialized view, and hybrid hash join: A performance analysis. In *Proc. Data Engineering*, 1990.

[CG96] L. Colby and T. Griffin. An algebraic approach to supporting multiple deferred views. In *Proc. Int'l Workshop on Materialized Views: Techniques and Applications*, Montreal, Canada, June 7 1996.

[CGL+96] L. Colby, T. Griffin, L. Libkin, I. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *SIGMOD* 1996.

[CKL+96] L. Colby, A. Kawaguchi, D. Lieuwen, I. Mumick, and K. Ross. Supporting Multiple View Maintenance Policies: Concepts, Algorithms, and Performance Analysis. AT&T Technical Memo.

[CM96] L. Colby and I. Mumick. Staggered maintenance of multiple views. In *Proc. Int'l Workshop on Materialized Views: Techniques and Applications*, Montreal, Canada, June 7 1996.

[CW91] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *VLDB* 1991.

[GL95] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *SIGMOD* 1995.

[GM95] A. Gupta and I. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Engineering Bulletin, Special Issue on Materialized Views and Data Warehousing*, 18(2):3–19, June 1995.

[GMS93] A. Gupta, I. Mumick, and V. Subrahmanian. Maintaining views incrementally. In *SIGMOD* 1993.

[Han87] E. Hanson. A performance analysis of view materialization strategies. In *SIGMOD* 1987.

[HZ96] R. Hull and G. Zhou. A framework for supporting data integration using the materialized and virtual approaches. In *SIGMOD* 1996.

[KLM+96] A. Kawaguchi, D. Lieuwen, I. Mumick, and K. Ross. View maintenance in nested data models. In *Proc. Int'l Workshop on Materialized Views: Techniques and Applications*, Montreal, Canada, June 7 1996.

[KLM+97] A. Kawaguchi, D. Lieuwen, I. Mumick, D. Quass, and K. Ross. Concurrency control theory for deferred materialized views. In *ICDT* 1997.

[KR87] B. Kähler and O. Risnes. Extended logging for database snapshots. In *VLDB* 1987.

[LHM+86] B. Lindsay, L. Haas, C. Mohan, H. Pirahesh, and P. Wilms. A snapshot differential refresh algorithm. In *SIGMOD* 1986.

[LMSS95] J. Lu, G. Moerkotte, J. Schu, and V. Subrahmanian. Efficient maintenance of materialized mediated views. In *SIGMOD* 1995.

[MRS93] I. Mumick, K. Ross, and S. Sudarshan. Design and implementation of the SWORD declarative object-oriented database system, 1993. Unpublished Manuscript.

[Mum95] I. Mumick. The Rejuvenation of Materialized Views. In *Proc. Int'l Conf. on Information Systems and Management of Data (CISMOD)*, Bombay, India, November 15-17 1995.

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

[RK86] N. Roussopoulos and H. Kang. Principles and techniques in the design of ADMS±. *IEEE Computer*, pages 19–25, December 1986.

[Rou91] N. Roussopoulos. The incremental access method of view cache: Concept, algorithms, and cost analysis. *ACM TODS*, 16(3):535–563, September 1991.

[SI84] O. Shmueli and A. Itai. Maintenance of Views. In *SIGMOD* 1984.

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

[SR88] J. Srivastava and D. Rotem. Analytical modeling of materialized view maintenance. In *PODS* 1988.

[ZGHW95] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *SIGMOD* 1995.