Lazy Maintenance of Materialized Views

Jingren Zhou
Microsoft Research
jrzhou@microsoft.com

Per-Ake Larson
Microsoft Research
palarson@microsoft.com

Hicham G. Elmongui *
Purdue University
elmongui@cs.purdue.edu

ABSTRACT

Materialized views can speed up query processing greatly but they have to be kept up to date to be useful. Today, database systems typically maintain views *eagerly* in the same transaction as the base table updates. This has the effect that updates pay for view maintenance while beneficiaries (queries) get a free ride! View maintenance overhead can be significant and it seems unfair to have updates bear the cost.

We present a novel way to *lazily* maintain materialized views that relieves updates of this overhead. Maintenance of a view is postponed until the system has free cycles or the view is referenced by a query. View maintenance is fully or partly hidden from queries depending on the system load. Ideally, views are maintained entirely on system time at no cost to updates and queries. The efficiency of lazy maintenance is improved by combining updates from several transactions into a single maintenance operation, by condensing multiple updates of the same row into a single update, and by exploiting row versioning. Experiments using a prototype implementation in Microsoft SQL Server show much faster response times for updates and also significant reduction in maintenance cost when combining updates.

1. INTRODUCTION

Materialized views transparently pre-compute joins and aggregations and, when applicable, can reduce query execution time greatly. Materialized views are widely used in data warehousing/decision support systems. To ensure a correct result, a materialized view must be up to date whenever it is accessed by a query. Most database systems achieve this by *eager maintenance* where all affected views are maintained as part of the update statement or the update transaction [1].

Under eager maintenance, the cost of maintaining a view is born entirely by updates while the beneficiaries of the view (queries) get a free ride! View maintenance overhead can be quite high when multiple views require maintenance, resulting in poor response times for updates.

Forcing updates to pay for view maintenance seems rather unfair and may also be inefficient if there are many small updates. To address this situation, some database systems also support *deferred maintenance* where maintenance of a view is delayed and takes place only when explicitly triggered by a user. This approach has the serious drawback that a query may see an out-of-date view and produce an incorrect result. Allowing the query optimizer to automatically use such views compromises correctness. The use of materialized views is no longer automatic and transparent to users. Query issuers have to know what views are used by a query, how they are maintained and whether they are, or need to be, up to date at execution time.

Ideally we would like a solution that both relieves the burden of view maintenance from updates and retains the property that queries always see up-to-date views. *Lazy view maintenance* introduced in this paper achieves these two seemingly contradictory goals.

Under lazy maintenance, updates do not maintain views but just store away enough information so that affected views can be maintained later. Actual maintenance is done by low-priority jobs running when the system has free cycles available. If the system has enough free cycles and a view is maintained before it is needed by queries, neither updates nor queries pay for view maintenance! If a view is not up to date when needed by a query, it is transparently brought up to date before the query is allowed to access it. In this case, the first beneficiary of the view "pays" for all or part of the view's maintenance by experiencing a delay. However, it only pays to maintenance of views that it uses and not for maintenance of other views affected by an update.

Lazy maintenance allows updates to complete faster so locks are released sooner, which reduces the frequency of lock contention, lock conflicts and transaction aborts. This is particularly important for updates that affect highly-aggregated views because they tend to have higher rates of lock conflicts.

We exploit row versioning for lazy view maintenance, which greatly simplifies the maintenance operation. Maintenance cost can also be reduced by merging maintenance tasks, which allows them to be processed more efficiently. We also introduce a new "Condense" operator and optimization rules to prune out redundant updates of the same row as early as

*Work performed while visiting Microsoft Research. The author is also affiliated with Alexandria University, Egypt.

[1]Throughout the paper, unless otherwise stated, the term "update" is used generically and refers to any kind of modification operation (insert, delete, and update).

VLDB '07, September 23-28, 2007, Vienna, Austria.

possible. The savings can be substantial as we will show in a later section. We also gain flexibility to schedule maintenance tasks, for example, they can be prioritized so that more frequently referenced views are maintained first.

The main contributions in this paper are as follows.

1. We introduce a new approach to maintaining materialized views that relieves updates of view maintenance while still ensuring that queries see only up-to-date views. The process is *transparent* to applications.
2. We exploit row versioning to obtain simple and efficient maintenance expressions.
3. We reduce the cost of view maintenance by merging multiple maintenance tasks for a view and by eliminating redundant updates of the same row.
4. We exploit system free cycles to maintain views using low-priority background jobs. If a view is not up to date when a query needs it, the query waits while the view is immediately brought up-to-date.
5. A prototype implementation in SQL Server 2005 and extensive experiments demonstrate the feasibility and benefits of our approach.

The rest of the paper is organized as follows. We first outline an overview of our approach in Section 2. We present basic maintenance algorithms and consider combining multiple update transactions together for the maintenance purpose in Section 3. In Section 4, we introduce a new "Condense" operator and optimization rules to generate more efficient maintenance plans. Extensive experimental results on Microsoft SQL Server 2005 are provided in Section 5. We further discuss various issues in Section 6. We survey related work in Section 7 and conclude in Section 8.

2. SOLUTION OVERVIEW

We begin with an overview of our design in this section and describe individual components in more detail in subsequent sections. Our design relies on being able to read old versions of base table rows. Exploiting row versioning allows for simpler and more efficient maintenance expressions.

To assist the reader in understanding our approach, we provide a brief description of row versioning and snapshot isolation in Microsoft SQL Server 2005. Although our prototype is built on SQL Server, the techniques are applicable to any other system with support for row versioning.

2.1 Background Information

SQL Server 2005 includes a *version store* that stores old versions of records and makes it possible to read a table as of an earlier point time. The version store is designed for short-term transient versioning and not for persistent versioning. It is used for multiple purposes, for example, to implement online index build and snapshot isolation [3]. Here, we focus on the functionality needed for lazy view maintenance. The description is slightly simplified for ease of understanding; a detailed discussion of the implementation is out of the scope of the paper.

For versioning purposes, each transaction is assigned a unique transaction sequence number (`TXSN`) when the transaction begins and a commit sequence number (`CSN`) when it commits. These sequence numbers are monotonically increasing and drawn from a single counter. Within a transaction, each statement is also assigned a unique statement number (`STMTSN`).

Each record in the version store contain version information, such as which transaction (`TXSN`) and which statement (`STMTSN`) created the version. The version store maintains a *version chain* for each record of a table, view, or index. By following the version chain of a record, the system can recover any old version of the record that is still available.

The key feature of the version store is that given a transaction sequence number `TXSN` and a statement number `STMTSN`, the version store can return record versions as of either the *beginning* or the *end* of this transaction/statement. If a record is inserted after the statement, it is not visible.

The version store keeps track of all active transactions and what versions they need. A version has to be kept only until all transactions that may require it have terminated. Older versions are automatically reclaimed by a garbage collector.

Snapshot isolation [3] guarantees that all reads within a transaction will see a consistent snapshot of the database. The main benefit is that read-only transactions do not block update transactions, which is particularly important for long-running queries. In SQL Server, snapshot isolation is implemented using the version store.

Even though our approach is applicable for other isolation levels, the details of lazily maintaining a view depend on the isolation level of the originating transaction. For ease of presentation, we assume that all transactions run under snapshot isolation in this paper.

2.2 System Overview

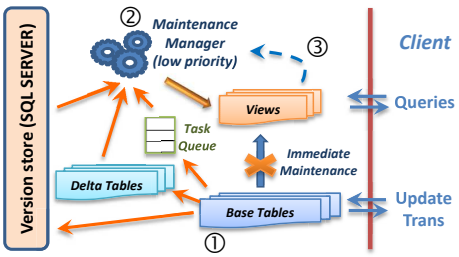


Figure 1: System Overview

Figure 1 shows our overall system design for lazy view maintenance. We first describe the individual components and then explain the overall procedure. Our design is primarily driven by efficiency considerations. We postpone comparison with alternative approaches to Section 6.

Delta Tables. Execution of an insert, delete, or update statement against a base table produces a stream of delta rows. The delta stream is then transformed into a *split delta stream* with an additional *action column* `Action`. Each delta row in the split delta stream encodes what change was made to a (uniquely identified) row of the target base table. The action column indicates if the delta row represents an insert, delete, or update of a row. In a split delta stream, an update is represented by two delta rows, one containing the old values with action "delete", the other containing the new values with action "insert". A more detailed description of delta streams can be found in [7].

For each base table we create a corresponding *delta table* which stores the *split* delta stream for its base table. Rows in the delta table include two additional columns, the transaction sequence number `TXSN` and the statement sequence number `STMTSN` that indicate which transaction and statement produced the delta row. These two columns are used

when building maintenance expressions as explained in Section 3. A delta table is clustered on columns `TXSN`, `STMTSN`, `Action`, plus the primary key columns of its base table.

Maintenance Task. When an update transaction commits, a lazy maintenance task is generated for each materialized view referencing a table that was updated by the transaction. When the task executes later on, its effect on the view should be the same as it would be if the task were executed immediately as part of the update transaction.

A maintenance task specifies which view needs to be maintained, the set of updated base tables, the transaction sequence number (`TXSN`) and the commit sequence number (`CSN`) of the originating transaction, at what statement (`STMTSN`) to begin (explained later), and current status of the task (*pending*, *inprogress* or *completed*). `TXSN` and optional `STMTSN` are used to locate the correct delta rows and versions of base tables when lazy maintenance is actually performed later. `CSN` is used to decide the task maintenance order. Task status is used by the maintenance manager to schedule and track individual tasks. More details about the maintenance algorithm are provided in Section 3.

Maintenance Manager. This component keeps track of active view maintenance tasks and what database versions and delta streams are needed. It is also responsible for constructing view maintenance jobs (from maintenance tasks) and scheduling them.

To be able to quickly find all maintenance tasks for a given view, the manager maintains a hash table containing an entry for each materialized view with active maintenance tasks. Each entry has a linked list containing the maintenance tasks of the view. The list is sorted in an increasing order on commit sequence number.

To be able to determine what versions and delta streams are still needed, the maintenance manager keeps track of the status of all maintenance tasks generated by an update transaction. When they have all completed, neither the database versions required by those tasks nor the delta streams generated by the transaction are needed any longer. The manager maintains a list of pending view maintenance tasks for each update transactions. All tasks generated by a transaction have completed when the list is empty or contains only completed tasks. In addition, the update transactions are inserted into a hash table to allow quick access based on the transaction sequence number. Using these data structures, the manager makes sure all versions required for lazy maintenance are kept in the version store and are properly released once maintenance is done. The manager also periodically creates low-priority jobs to delete obsolete delta streams from delta tables.

Task Table. Maintenance tasks are also stored persistently in a global view-maintenance task table. The table is used for recovery purposes only, not for normal processing. A maintenance task is added to this table as part of the transaction that generated it and deleted as part of the transaction that performs the maintenance. We discuss recovery strategy in Section 6.

2.2.1 Update Transactions

Consider an update statement modifying a base table R that is referenced by a number of materialized views. Eager maintenance updates all materialized views that reference R immediately after the update statement. In the case of lazy maintenance, as shown in Figure 1, *Step* ①, view maintenance is skipped [2]. Instead, enough information is saved so the affected views can be updated later. The *split* delta stream produced by the update statement is appended to the corresponding delta table. Versioning is enabled so that when the update is applied to R, the old version of each modified row is stored in the version store.

An update transaction may contain multiple update statements. The transaction internally records which table is modified by which statement and which views are affected. Each update statement reports its own information at the end of its execution.

When the update transaction commits, maintenance tasks are constructed based on the information reported during execution. One maintenance task is generated per affected materialized view. The tasks are then passed on to the maintenance manager and also written to the persistent task table. If the update transaction aborts, no information is saved and no maintenance tasks are constructed.

2.2.2 Lazy Maintenance

Lazy maintenance is shown as *Step* ② in Figure 1. The manager wakes up every few seconds. If there are no pending maintenance tasks or the system is currently busy, it goes back to sleep. Otherwise, it decides what views to maintain and, for each view, constructs a *low-priority background maintenance* job and schedules it. Maintenance jobs for the same view are always executed in the commit order of the originating transactions.

We explain how to generate maintenance expressions and how to schedule maintenance jobs in Section 3. The maintenance manager may combine multiple maintenance tasks for the same view into a larger job that can be executed more efficiently. During maintenance, delta streams from delta tables and appropriate versions of base tables from the version store are used.

When a maintenance job completes, it reports back to the maintenance manager. The manager then removes the completed tasks from its task list and releases any row versions and delta rows that are no longer required by the remaining pending tasks. As part of the maintenance transaction, the completed maintenance tasks are also deleted from the persistent task table. If a view is dropped, all of its pending tasks are removed. Altering a view is treated as dropping the view and creating a new one.

2.2.3 Query Execution

During query execution, shown as *Step* ③ in Figure 1, we have to make sure that all views used are up to date. Before a query plan begins execution, we check whether the views used by the plan have any pending maintenance tasks and whether they originate from transactions whose effects the query is supposed to see. Under snapshot isolation this means any transaction that committed before the current transaction began. If such tasks exist, the query asks the maintenance manager to schedule them immediately and then waits. The query resumes execution when the maintenance tasks have completed. We call this type of maintenance *on-demand maintenance*. We briefly describe the process here and discuss further considerations in Section 3.5.

[2]All indexes on the base table are updated eagerly. In this paper, we consider only lazy maintenance of views but our techniques are applicable to indexes as well.

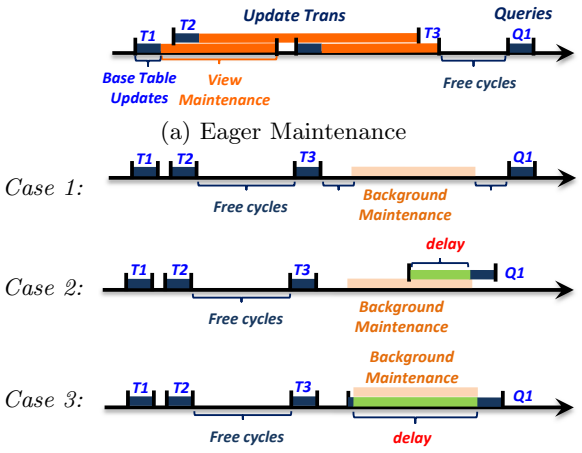


Figure 2: Response Time Benefits

If all views required by a query are up to date, there is virtually no delay in query execution. Note that the on-demand maintenance jobs are executed in separate transactions and commit before query execution resumes. Even if the query aborts, it does not trigger committed maintenance jobs to roll back.

A slightly more complex case occurs when, within the same transaction, we have update statements that affect views referenced by subsequent queries. The queries are supposed to see all changes made by prior update statements within the transaction so we have to update the views, but such in-transaction changes to the view cannot be made permanent because the transaction may abort.

We handle this case in the following way. Suppose we are about to begin execution of a query plan that uses a view V. We first request the maintenance manager to perform on-demand maintenance to bring view V up to date as of the beginning of this transaction. This part of maintenance is done in separate transactions so even if the current transaction fails, the effects of these maintenance jobs will not be rolled back. We then check whether the current transaction has updated any table that is referenced by view V. If it has, we maintain V by applying updates from this transaction to the view. This part of maintenance is executed within the current transaction so if the transaction later fails, all its effects on the view will be rolled back automatically.

2.3 Effect on Response Time

Lazy view maintenance is completely transparent to applications. Applications exploit materialized views in the same way as before and always see a state that is transactionally consistent with base tables. The only difference is in response time of updates and queries, which is the topic of this section. Suppose we have three updates followed by a query. All three updates affect a materialized view that is used by the query.

Under eager maintenance, shown in Figure 2(a), each update has to wait until view maintenance is done. If the affected views are expensive to maintain, update response times may be very slow. When the query arrives, the updates have completed and the view content is up to date so the query completes quickly.

Under lazy maintenance, shown in Figure 2(b), the response time of the updates is much improved. Suppose the system gets a chance to maintain the affected views after the three updates. By combining the three updates, the total time spent on maintaining the views is reduced. If the query arrives after lazy maintenance is done, shown in *Case 1*, its response time is the same as under eager maintenance. If the query arrives in the middle of lazy maintenance of a view that it needs, shown in *Case 2*, it is forced to wait until maintenance of that view is finished. Finally, if the query arrives immediately after updates and before the system has begun maintenance of the view, shown in *Case 3*, the query issues a on-demand maintenance request at the beginning and waits until it is finished. The total system response time for all the updates and the query is still improved over eager maintenance.

From this example, we see that lazy view maintenance consistently improves response time for updates and view maintenance overhead is fully or partially hidden from subsequent queries. How much of the overhead is hidden depends on how frequently the view's base tables are updated, how frequently the view is queried, and the overall load on the system.

3. MAINTENANCE ALGORITHMS

Lazy view maintenance is performed by executing a maintenance task. Maintenance tasks for the same view are executed strictly in the commit order of the originating transactions. This is enforced by comparing the commit sequence numbers (`CSN`) stored in the maintenance tasks. However, maintenance tasks for different views can be scheduled independently without any restriction on order. We discuss various scheduling strategies in Section 3.5.

3.1 Full and Partial Maintenance Tasks

As described earlier, a maintenance task specifies which view to maintain and the set of updated base tables. Lazy maintenance should produce the same effect on the view as if it had been maintained immediately in the originating transaction. This is achieved by retrieving appropriate base table versions from the version store and recovering delta streams from delta tables using the transaction sequence number (`TXSN`) of the originating transaction and, if necessary, the starting statement number (`STMTSN`).

Consider an update transaction T consisting of multiple statements and denote its transaction sequence number by T.`TXSN`. Suppose the third statement of T inserts rows ΔR into a table R and there is a view V that references R and another table S. If view V were maintained immediately after the insert statement, what version of S would the maintenance expression see? Because the transaction runs under snapshot isolation, it would see a version of S that includes all updates that committed *before* T started plus all updates of T *prior* to the current insert statement. Therefore, the lazy maintenance task should store `TXSN` $= T$.`TXSN` and `STMTSN` $= 3$. If the earlier statements in T did not update S, `STMTSN` is optional because the version of S seen by the third statement is the same as of the beginning of the transaction. In this case, the maintenance task includes changes from all update statements in the transaction. We call this type of maintenance tasks as "full maintenance tasks".

A more complicated case shows when including `STMTSN` is important. In the previous example, if the fourth statement of T references the view V, we need to maintain the view V up to the point of the fourth statement, including the third statement that inserts ΔR. Such maintenance be-

comes permanent once the transaction T commits. Suppose the fifth statement updates S. The corresponding lazy maintenance task should store `TXSN` $= T$.`TXSN` and `STMTSN` $= 5$. In this case, the statement number is crucial and tells us that all delta streams generated by statements prior to the fifth statement have already been applied to the view and only the remaining delta streams from the transaction need to be applied to complete maintenance. We call this type of maintenance tasks as "partial maintenance tasks".

3.2 Normalized Delta Streams

Before describing how to incrementally compute the view delta using delta tables and versioning, we present an important observation that forms the foundation of our solution.

Consider a transaction T containing a series of update statements that update tables R and S. Let ΔR^i, $i = 1, \cdots, n$ denote the split delta stream produced by the ith statement updating table R. Similarly, ΔS^i, $i = 1, \cdots, m$ denotes the split delta stream produced by the ith statement updating table S.

The update statements are processed in some order when the transaction executes, producing the delta streams in the same order. Suppose, for example, that the delta streams are produced in the order of $\Delta R^1, \Delta S^1, \Delta R^2, \Delta S^2, \cdots$. Denote the initial states (when the transaction begins) of R and S by R_0 and S_0 and the final states (when the transaction ends) by R_1 and S_1. If we apply the delta streams to the initial states R_0 and S_0 in the given order, the tables will end up in the final states R_1 and S_1.

Now we re-order the delta streams so that the R deltas occur first followed by the S deltas, $\Delta R^1, \Delta R^2, \cdots, \Delta R^n$, $\Delta S^1, \Delta S^2, \cdots, \Delta S^m$. If we apply the deltas in this order to R_0 and S_0, the tables still end up in exactly the same final states R_1 and S_1. Note that we did not change the ordering among the deltas for table R or the ordering of records within each delta stream. It is important that multiple delta rows that affect the same row are applied to the base table in the originating order. Otherwise, we could no longer guarantee that the tables end in the same final state.

Next we *concatenate* the R deltas and the S deltas, obtaining $\Delta R = \Delta R^1 + \Delta R^2 + \cdots + \Delta R^n$ and $\Delta S = \Delta S^1 + \Delta S^2 + \cdots + \Delta S^n$. Concatenation ensures that delta rows from ΔR^1 come before ΔR^2, and so on. This can be done by sorting the delta rows in ascending order on `TXSN` and `STMTSN`. The end result is as if there there had been two large update statements, producing delta stream ΔR and ΔS, respectively. If we apply ΔR to R_0 and then ΔS to S_0, the tables will end up in the same final states R_1 and S_1. The two deltas are applied to separate tables so it doesn't matter in what order we apply them. We summarize the observation below, but omit the proof due to space limitation.

Proposition 3.1 *Two sequences of delta streams affecting a set of base tables are* equivalent *if both produce the same final state when applied to the same initial state of the base tables. Any sequence of delta streams can be* normalized *to an equivalent sequence of delta streams consisting of one delta stream for each affected table.*

As discussed above, there are several sequences of delta streams that are equivalent to the original one generated when the query executes. Provided that we have a correct incremental view maintenance algorithm, we can choose any one among the equivalent sequences to use for maintenance.

For simplicity and efficiency, we always use the normalized sequence; it only requires a maintenance algorithm able to handle updates of one table and has the fewest delta streams.

3.3 Computing View Delta Streams

In this section, we focus on how to compute the delta stream ΔV to be applied to the view. As described in Section 2.2, delta rows in delta tables are stored together with three special columns `TXSN`, `STMTSN` and `Action`. When the chosen delta rows participate in incremental maintenance and joins with other tables, if necessary, the three special columns are carried along into the final computed ΔV. We discuss how to apply ΔV to the view using these special columns and some further optimizations in Section 4.

For simplicity, we derive expressions for an n-way join view $V = R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$. Maintaining an aggregated view can be done by first computing the delta of its non-aggregated part and then aggregating the delta rows before applying them to the view. During aggregation, the action column is used to decide whether to decrease or increase the value of aggregated columns. An additional count per group is maintained to decide when the group becomes empty and should to be deleted from the view. More details on maintaining aggregated views can be found in [7].

We start with the simple case when only one table, R_i, of the view has been updated, producing the concatenated delta stream ΔR_i. As shown already in [4], the view delta can then be incrementally computed as

$$\Delta V = R_1 \bowtie \ldots \bowtie R_{i-1} \bowtie \Delta R_i \bowtie R_{i+1} \bowtie \ldots \bowtie R_n$$

The rows of ΔR_i are stored in R_i's delta table. We know which transaction produced the delta so we can retrieve ΔR_i from the delta table by a simple selection query specifying the transaction's sequence number (`TXSN`). The selected delta may contain several delta streams, each one from a different update statement. The statement number (`STMTSN`) from the delta table indicates the update order. Note that $R_j (j \neq i)$ represents the version *before* the transaction. In fact, because only table R_i was updated, for all other tables, the versions before and after the update transaction are the same. We can retrieve either version from the version store.

Now consider the general case when a maintenance task represents a transaction with multiple update statements that modify m out of the view's n base tables. Each update statement updates only one table, of course, but different statements may update the same or different tables. Without loss of generality, we assume that base tables $R_1, \ldots, R_m$ are updated.

We maintain the view using the normalized delta streams $\Delta R_1, \Delta R_2, \cdots \Delta R_m$, where ΔR_i is the concatenation of the split delta streams from statements updating table R_i. Let R_i' denote the *after*-version of R_i, that is, at the end of the transaction after applying ΔR_i to R_i. As described earlier, ΔR_i can be retrieved from the corresponding delta table with the appropriate selection predicates on `TXSN`. The *before* and *after* versions of base tables R_i are also available because of the version store. By using the normalized delta streams, we can compute the view delta as

$$\begin{aligned}\Delta V = {} & \Delta R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n \bowtie \{1\} + \\ & R_1' \bowtie \Delta R_2 \bowtie R_3 \bowtie \ldots \bowtie R_n \bowtie \{2\} + \\ & \ldots + \\ & R_1' \bowtie \ldots \bowtie R_{m-1}' \bowtie \Delta R_m \bowtie \ldots \bowtie R_n \bowtie \{m\}\end{aligned}$$

A similar formula was introduced in [10], but here we present a different interpretation. We ignore the last join with a constant in each term for now.

The m base table deltas are applied one by one, in m steps. The view delta is computed as if the transaction had proceeded as follows. First all updates to base table R_1 are performed, producing the delta stream ΔR_1, and bringing the table to state R_1'. The first term in the expression computes the view delta needed to incorporate the effects of ΔR_1 into the view. Next all updates to base table R_2 are performed, producing ΔR_2, and bringing the table to state R_2'. The second term in the expression computes the view delta needed to incorporate the effects of ΔR_2 into the view. Note that the term must use R_1' because R_1 has already been updated. This pattern continues with one term for each updated base table until the m deltas are covered.

The final computed ΔV is the concatenation of the deltas from the m steps. We also add a *step sequence number* (`SSN`) to each row in the view delta. All rows generated by the ith term have `SSN` $= i$ (added by the last join in each term). The combination of `SSN`, `TXSN`, and `STMTSN` defines the order in which to apply the delta rows to the view (although in this case `TXSN` is the same for all delta rows). This order is important because different terms may generate delta rows for the same view row and we must ensure that they are applied in the right order. First all delta rows from the first term are applied, then all delta rows from the second term, and so on. For each term, the delta rows are applied in statement sequence order, which is the order of the original update statements.

The above equations are still valid for partial maintenance tasks but with adjustments for the fact that a prefix of the base table deltas has already been applied to the view. The statement sequence number of the first unprocessed statement is included in the task and we denote it by $Task$.`STMTSN`. The *before* version of a table needed is now the version at the beginning of statement $Task$.`STMTSN`. The normalized delta stream should include on deltas generated by statement $Task$.`STMTSN` or later. We can retrieve the required delta from the delta table by using a selection predicate that specifies not only the transaction sequence number but also a lower bound on the statement sequence number, $DeltaTable$.`STMTSN` $\geq Task$.`STMTSN`.

In summary, the maintenance expression computes one term for each updated table and concatenates the results. When computing the term containing ΔR_i, we replace ΔR_i with a selection on the corresponding delta table. When reading other base tables, we supply version hints to the table read operators, which then rely on the version store to return the appropriate version of each row.

Finally, all changes applied to the view will be tagged with the transaction sequence number of the original transaction. The net effect is that the view appears to be maintained by the original transaction, which was precisely the goal.

3.4 Combining Maintenance Tasks

For each update transaction, a maintenance task is created for each materialized view affected by the transaction's updates, regardless of the size of updates. When the maintenance manager decides or is required to maintain a view, several maintenance tasks may have accumulated in the view's task queue. The tasks for each view must be performed in the commit order of the originating transactions. The naive policy would be to process them one by one by issuing a separate maintenance job for each task. However, if each task only contains small updates, it is quite wasteful to apply them one by one. The more efficient way is to combine them into a single, larger maintenance job. Combining maintenance tasks has several benefits.

1. The combined task is executed once so we avoid multiple invocation of the same or similar maintenance expressions.
2. If different tasks update the same base table row or the same row in the view, the intermediate updates are redundant and can be eliminated. This reduces the size of the delta stream, which reduces the cost of evaluating the maintenance expression and applying the updates to the view. (This topic is covered in Section 4.)

Both full and partial maintenance task can be combined but our current prototype combines only full tasks; partial maintenance tasks are performed by themselves [3].

Suppose the materialized view V has a queue of l pending maintenance tasks that were generated by transactions $T_1, \ldots, T_l$ (in commit order), updating the set of base table $\mathcal{B}_1, \ldots, \mathcal{B}_l$ respectively. Assume transaction T_e has the smallest transaction sequence number, that is, T_e begins the earliest. We can treat transactions $T_1, \ldots, T_l$ as a single large transaction T_0 that starts at T_e.`TXSN`, ends at T_l.`CSN` and updates the set of base tables $\mathcal{B}_1 \cup \ldots \cup \mathcal{B}_l$.

The maintenance algorithm in the previous section can then be applied to this large transaction. Suppose table R_i is one of the updated tables. The ΔR_i is the concatenation of the R deltas from transactions $T_1, \ldots, T_l$ (in commit order). The before-version of a table is now the version before transaction T_e and the after-version includes the updates from all l participating transactions. Because we treat the l transactions as one big transaction, we need only two versions of base tables $R_i, \ldots, R_m$ plus their delta changes.

Similar to a single task, the combination of `SSN`, `TXSN`, and `STMTSN` defines the order in which to apply delta rows to the view. Note that the delta rows are applied in the order of transaction sequence number `TXSN`, which may be different from the commit `CSN` order of the originating transaction. It can be shown that this still produces a correct result. The reason is that, under snapshot isolation, two *committed* transactions whose commit order is different from their transaction begin order cannot have updated the same base table row. If they did, a conflict would have been detected and one of them would have been aborted. As a result, using the combination of `SSN`, `TXSN`, and `STMTSN` as the update order for the view produces the correct result.

Maintenance tasks against a view cannot always be combined because intermediate versions of the view are lost. By combining tasks from update transactions $T_1, \ldots, T_l$, the updates applied to the view will be tagged with the transaction sequence number of T_e so all changes appear to be made by transaction T_e and the view is brought to the state produced by T_l. When a new version of a row is created, the old version needs to be kept only if it may be read by an active transaction. If, at the point of lazy maintenance, there are

[3]Partial maintenance tasks can also be combined. All the originating transactions have successfully committed and, therefore, they have no conflicting base table updates. However, making sure that the correct versions of base tables are used during maintenance becomes somewhat more complex.

no active transactions or other pending maintenance tasks that may require an intermediate version of V, we can safely combine the l tasks.

3.5 Scheduling Maintenance Tasks

Background Scheduling. Lazy maintenance can be triggered when the system has free cycles. In this case, the maintenance manager can freely choose which materialized view(s) to maintain. Scheduling of view maintenance has multiple, somewhat conflicting goals. First, it is desirable to hide view maintenance from queries as much as possible to improve query response time. Second, maintenance should be performed as efficiently as possible. Third, it is important to minimize the resources consumed by pending maintenance tasks. Any scheduling policy represents a trade-off among these goals.

To hide view maintenance from queries, views could be assigned priorities based on how soon they are expected to be referenced by queries. The sooner the view is expected to be used, the sooner the view needs to be maintained. Future reference information can be estimated based on historical usage of the views.

If a view has multiple pending tasks, the manager must also decide whether and how many to combine into a single maintenance job. Combining tasks improves efficiency but could result in a long-running maintenance transaction, This could cause resource contention, which we also want to avoid. The choice should depend on the size of combined delta streams, the estimated cost of the maintenance operation, and the current system workload.

Pending maintenance tasks consume space for storing delta streams and old row versions. Cleanup both in the version store and delta tables proceeds linearly, always releasing the oldest data first. Hence, a single very old maintenance task can prevent younger data from being released. To avoid this, older maintenance tasks should be given higher priority.

Other considerations may also be important when designing a scheduling policy. For example, we may want to schedule maintenance of similar views together in order to exploit common subexpressions and achieve better buffer pool efficiency. For some applications, we may know that a view is used only at a certain time of the day, for example, to produce reports. In that case, all that matters is that the view is brought up to date before that time.

Maintenance jobs run as low-priority background jobs but one could further reduce their impact on system resources. In case of a sudden burst in system workload, maintenance jobs can be paused or even aborted to avoid slowing down the system. Similar to [6, 15], we can also perform maintenance tasks in two phases, a view delta computation phase and a view delta apply phase, in order to further limit resource contention.

Further consideration of scheduling policies for view maintenance is beyond the scope of this paper. Our current prototype uses a simple policy that gives the highest priority to the oldest pending tasks.

On-demand Scheduling. Lazy maintenance can also be triggered by a query. In this case, the views referenced by the query are maintained immediately. The maintenance manager must still decide whether and how to combine maintenance tasks. The maintenance job(s) inherit the same priority as the query.

A more interesting question is when it is possible to avoid maintaining a view even though it is referenced by a query. A view referenced by a query does not have to be brought up to date immediately if the pending updates do not affect the part of the view accessed by the query. It may be worthwhile to first check whether the pending maintenance tasks can cause a change in the view that is visible to the query. If not, the view does not have to be maintained immediately while still safely serving the query.

There are several ways to check. For example, we can project the query predicate onto each base table and scan the corresponding delta tables with the projected predicate [4]. If no scans return any tuples, we can safely deduce that the view content accessed by the query cannot be affected by the pending updates. This can be easily proven because it means that none of the m terms in the maintenance expression can produce a result affecting rows accessed by the query. However, this filtering operation can be as expensive as maintaining the view.

In either scheduling mode, the maintenance manager schedules one job at a time for one view. This is achieved by monitoring the task's status in the manager. There can be at most one task with the status of *inprogress* for each materialized view.

4. CONDENSING DELTA STREAMS

The computed ΔV from the previous section contains special columns `Action`, `SSN`, `TXSN`, and `STMTSN`. In this section, we first briefly describe how delta rows with action columns are applied in SQL Server and then present additional optimization techniques for lazy maintenance.

In SQL Server, a "StreamUpdate" operator consumes the view delta stream and applies the changes to the affected rows in the view, based on the action column `Action`. It is, of course, inefficient to update a row by first deleting the old one and then insert a new one with the same keys. Before applying updates to the view, the set of delta rows is sorted on the unique clustering keys and the action column to guarantee that changes to the same row are adjacent. The delete action has a value smaller than the insert action to ensure that delete rows precede insert rows. The sort is followed by a "Collapse" operator that simplifies the delta stream. If two adjacent delta rows have the same keys, the first one is a deletion and the second one an insertion with new values, then the two rows are replaced by a single update row with new values. The final "StreamUpdate" operator applies the changes to the view, that is, for each delta row, finds the corresponding target row in the view, if any, and performs the action specified in the delta row.

Using split delta streams and sorting on the action column guarantees optimal data locality for updates. We refer readers to [7] for more details.

4.1 The Condense Operator

Under eager maintenance, view maintenance is part of the execution plan for the update *statement* and each maintenance expression processes only one split delta stream. There can be at most two delta rows with the same keys in the final computed ΔV, which simplifies the logic of the "Collapse" operator.

Under lazy maintenance this no longer holds because a

[4]If the query has no predicate on a base table, the projected predicate on the table is always true.

maintenance task may include delta streams from multiple statements. Two update statements updating the same base table may affect the same base table row. If so, they will also affect the same row in the view. Furthermore, even updates that affect different rows in different base tables may end up affecting the same row in the view. As a result, we may have more than two delta rows with the same keys in the final computed delta stream. When combining multiple maintenance tasks, this becomes even more common.

As described earlier, the order in which delta rows are applied is crucial to achieve the correct result. However, it is clearly not efficient to apply many changes to a target row one by one. For example, we do not care about the intermediate modifications; all that matters is the final state of the row.

To skip unnecessary changes, we introduce a new "Condense" operator and apply it to the sorted view delta stream. The stream must be sorted on the unique clustering keys of the view plus the update order, which is the combination of `SSN`, `TXSN` and `STMTSN`, and finally `Action`. The sorting ensures that all changes to the same view row are grouped together in the correct update order. The action column is included to ensure that deletion of a row (if any) occurs before an insertion originating from the same statement. The "Condense" operator takes the sorted delta stream as the input and produces a condensed delta stream by, in essence, discarding intermediate changes to rows.

		First delta row in the group	
		Insert	Delete
Last delta row in the group	Insert	Output last delta row	*Full condense:* • output an update delta row *Partial condense:* • output first and last delta rows
	Delete	Output nothing	Output last delta row

Table 1: "Condense" Operator

The "Condense" operator is an aggregation operator, somewhat similar to duplicate elimination. For each group of delta rows with the same values of the unique clustering keys, it outputs at most one delta row for a "Full Condense" operator or two delta rows for a "Partial Condense" operator, covered in the next section. The output depends only on the first and the last delta row, summarized in Table 1.

It is important to note that before lazy maintenance is performed, all base table updates have been successfully completed, with no constraint violations. Otherwise, the update transaction would have rolled back and there would be no lazy maintenance job for it. Specifically, an insert operation implies that there is no row with the same unique keys before.

4.2 Partial Condense

So far, we have described the "Condense" operator applied right before the sorted delta stream is applied to the view. This reduces a large number of redundant updates to the view. However, this does not limit redundant computation caused by multiple updates to the same base table row.

For example, consider a materialized view $V = R \bowtie S$. When combining multiple updates to the same base table R, if all the updates change only one row, we do not care about any *intermediate* version of that row for maintaining the view under certain conditions. The maintenance operation may only require the *first* and *last* delta rows, join them with S and apply the changes to the view V. In this case, it clearly makes sense to push the "Condense" operator down through the joins or partially condense the delta stream from the delta table of R. This reduces the number of tuples participating in joins and can reduce the cost of computing the view delta significantly.

In fact, partially condensing the delta stream of R can be viewed as replacing all update statements in the step with an equivalent update process, which contains only one update statement that changes R from the *before* version directly to the *after* version.

We introduce a "Partial Condense" operator, which performs the same as a "Condense" operator except that when the first delta row is a delete and the last delta row is an insert, it outputs two delta rows because the deleted row and the inserted row may affect different view rows.

A "Condense" operator is analogous to a "GroupBy" operator in a sense that the "Condense" operator groups on the sorted columns. Therefore, we can emulate all the optimization rules for aggregation, such as push aggregation down through joins, partial/global aggregation, etc.

Due to space limitation, we do not present all the optimization rules for "Condense" and "Partial Condense" operators; instead, we use a few examples to illustrate some important transformation rules *in the context of lazy maintenance*. The underlying principle is that *delta rows are condensable if they are guaranteed to affect the same view row.*

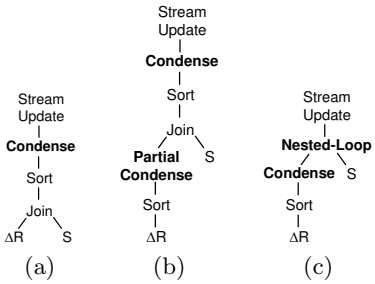
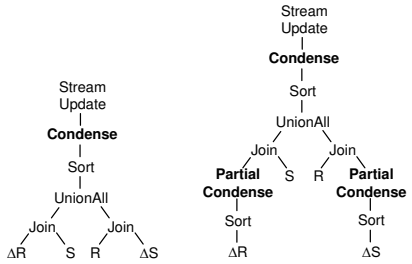


Figure 3: Transformations When Updating R

We first consider a simple case when only table R is updated in the maintenance task. Figure 3(a) shows the maintenance plan with a "Condense" operator on top. In fact, we can first sort ΔR on the unique keys of base table R plus columns `TXSN`, `STMTSN`, and `Action` and then partial condense the result. The "Partial Condense" operator removes all the intermediate versions of the delta rows. The maintenance plan is shown in Figure 3(b). If the unique clustering keys of the view is also the unique keys of the base table R, we can push the "Condense" operator through the join, as shown in Figure 3(c). In this case, the join needs to be a nested-loop join so that the order of the condensed tuples is maintained for the following "StreamUpdate" to consume.

When both R and S are updated in the maintenance task, following the algorithm in Section 3, Figure 4(a) shows the original maintenance plan. We leave out the version information for simple presentation. Note that each term only contains one delta stream. As shown in Figure 4(b), we can surely sort individual delta streams on their unique keys plus columns `TXSN`, `STMTSN`, and `Action` and then partial condense the sorted result, respectively. Again, if there are many redundant updates, the saving of partial condensing can be dramatic.



(a) (b)

Figure 4: Transformations When Updating R and S

Similar optimization can also be applied to maintenance expressions with aggregations. All the transformations are considered by the optimizer as alternatives. The optimizer should evaluate them in a cost-based fashion and choose the cheapest one.

5. EXPERIMENTAL RESULTS

We have prototyped lazy maintenance of materialized views in Microsoft SQL Server 2005. To demonstrate the benefits of lazy view maintenance, we compare response times for updates and queries in various situations. Lazy maintenance is performed in the background and the overhead may be fully hidden from users. Nevertheless, we also compare the time spent on lazy maintenance and compare it with the time required for eager maintenance to measure the overhead of lazy maintenance. Finally, we examine the worst-case overhead of using lazy maintenance instead of eager maintenance.

All experiments were performed on a workstation with a Pentium D 3.2 GHz processor, 1GB of memory and two 160GB disks, running Windows XP. All queries were against a 1GB version (SF=1) of the TPC-H database with cold buffer pool. For lazy maintenance, all updates and queries ran under snapshot isolation. Snapshot isolation imposes a small overhead for maintaining snapshots. For eager maintenance we ran all updates and queries under read committed isolation (the default), which may slightly bias the comparison in favor of eager maintenance.

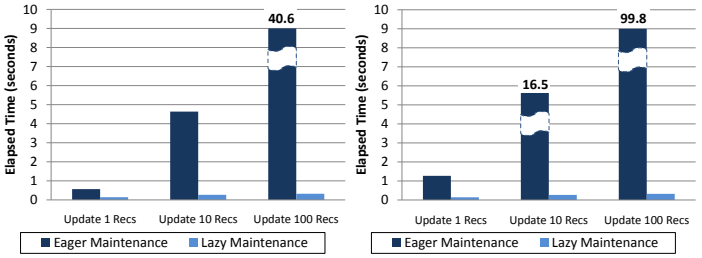
5.1 Update Response Time

Our first experiment demonstrates the improvement in update response time when using lazy view maintenance. We defined a highly aggregated materialized view V_1 that joins lineitem, orders, customer, and nation and aggregates by nation and market segment.

V_1:
```
create view V1 as
  select n_name, c_mktsegment, count(*) as totalcnt,
    sum(l_extendedprice) as totalprice,
    sum(l_quantity) as totalquantity,
  from customer, orders, lineitem, nation
  where c_custkey = o_custkey and o_orderkey = l_orderkey
    and n_nationkey = c_nationkey
  group by n_name, c_mktsegment
```

When an application updates customer information, for example, nation key or market segment, view V_1 is affected and needs to be maintained.

We compare update response time for lazy maintenance with that for eager maintenance. Figure 5(a) shows three scenarios in which we update 1, 10, and 100 records using a single update statement. With lazy maintenance, update response time is reduced to virtually nothing. The system



(a) V_1 (b) $V_1 + V_2$

Figure 5: Update Response Time

returns immediately after the customer table has been updated, delaying the expensive view maintenance. The more expensive the view maintenance, the greater the improvement in response time. For the case of updating 100 customer records, the update response time is improved by factor of 125.

What if more than one view is affected by the updates? We created another view V_2 that precomputes information for orders where the customer and the supplier come from different nations.

V_2:
```
create view V2 as
  select s_name, c_name, c_mktsegment, ps_comment, ...
  from customer, orders, lineitem, supplier, partsupp
  where c_custkey = o_custkey and o_orderkey = l_orderkey
    and l_suppkey = ps_suppkey and l_partkey = ps_partkey
    and ps_suppkey = s_suppkey and s_nationkey <> c_nationkey
```

We repeated the experiments with the same update statements. Figure 5(b) shows the update response time when both views are present. Similar to the previous experiment, the update response time under lazy maintenance is almost nothing. [5] Compared with the previous experiment, the update response time for eager maintenance increased significantly because additional view maintenance has to be done as part of the update transaction. Under lazy maintenance the update response time is virtually unchanged by addition of a second view.

In summary, lazy maintenance can reduce update response time by orders of magnitude because updates no longer have to wait for views to be maintained. Under lazy maintenance the update response time depends only on the cost of updating base tables and storing delta streams and not on the number and complexity of views affected.

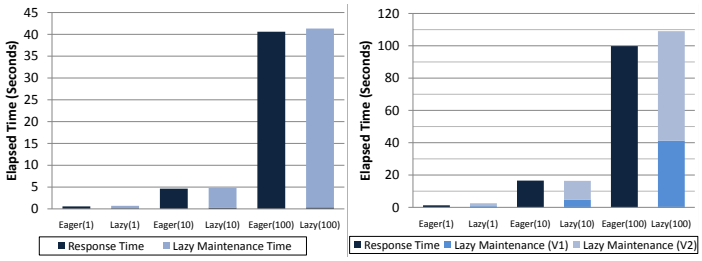
5.2 Maintenance Cost

When lazy maintenance is done in the background, the overhead may be completely hidden from users. Even so, we need to verify that lazy maintenance can be done efficiently and at a cost comparable to that of eager maintenance.

We measure the system time required for lazy maintenance of each view. The sum of the update response time and the time spent on lazy maintenance indicates the total amount of work spent on the update. Figure 6 shows the total amount of work for the same two experiments in the previous section. The update response time under lazy maintenance is too small to be visible in the figure.

The total amount of work under lazy maintenance is slightly higher, but still comparable to that of eager maintenance. The additional overhead is primarily caused by adding the

[5]We also run experiments with more than one table updated in a transaction, for example, updating both customer and lineitem. The reduction in update response time is still orders of magnitude.



(a) V_1 (b) $V_1 + V_2$

Figure 6: Breakdown of Total Time Spent on Update (including view maintenance)

delta streams into the customer delta table and by saving old versions of the updated customer rows. We delay a more detailed discussion of overhead until Section 5.5.

In the rest of this section, we present results with only one view V_1 present. The results with more views were similar.

5.3 Query Response Time

Under eager maintenance, a query can exploit a view for free since it has already been maintained. However, update transactions are slowed down by view maintenance so they keep locks on the affected views longer, which may force queries and other updates to wait. In this sense, materialized views are not completely free to queries.

Under lazy maintenance, query response time depends on when the query arrives. Before execution begins, the query first checks with the maintenance manager if the requested view is up to date. If not, the query waits until all pending and in-progress maintenance of the view is completed.

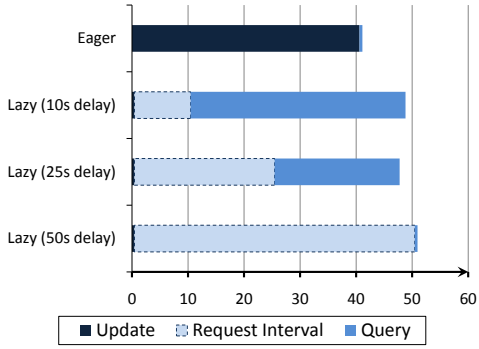


Figure 7: Query Response Time

In this experiment, we issued a 100-row update of the customer table followed by a query exploiting view V_1. In the eager maintenance case, we issued the query immediately after the update statement returned. In the lazy maintenance cases, we issued the query 10, 25, or 50 seconds after the update statement.

Figure 7 compares query response times for different scenarios. Again, the update response time under lazy maintenance is too small to be visible in the figure. When the query is issued after lazy maintenance has completed, for example 50 seconds after the update in the figure, neither the update nor the query pay the cost for maintaining the view! The maintenance cost is hidden from the application. Of course, if the query is issued soon enough, there may be some overlap between view maintenance and query execution. Still, the total response time for the update and the query is faster than that under eager maintenance.

In fact, when queries are much more frequent than updates, only the first query after an update may be delayed; the ones arriving later still use the view for free.

5.4 Multiple Updates

To demonstrate the benefits of combining maintenance task, we ran an experiment with 100 small update statements. Each statement ran in a separate transaction.

We consider two update scenarios. In the first scenario, each update statement randomly updates 1-10 consecutive rows in the customer table. The updated rows are scattered all over the table. The second scenario is similar, except the updates are highly skewed, uniformly scattered among the first 100 rows in the customer table.

For each update scenario, we report total system time spent for both eager maintenance and lazy maintenance, shown in Figure 8. Under lazy maintenance, we compare two maintenance strategies: (1) maintain update transactions one after another; (2) merge the 100 update transaction by combining their delta streams and maintain the view *once*. In the latter case, as described in Section 4.2, we apply a "Partial Condense" operator after reading the delta stream from the customer delta table.

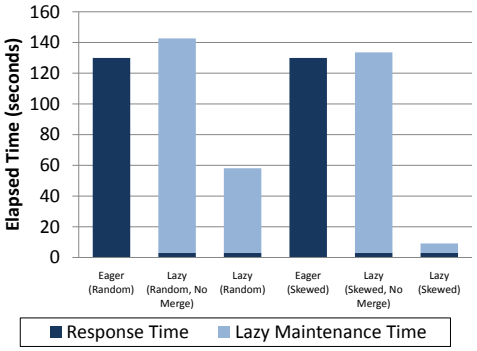


Figure 8: Merge Multiple Updates

In both update scenarios, the update response time is greatly reduced. The maintenance time is also significantly reduced by combining maintenance tasks. In the random update scenario, the reduced maintenance time is mainly because we avoid invoking and initializing maintenance plans 100 times individually. In the skewed update scenarios, the partial `condense` operator reduces the update rows from almost 100,000 rows to roughly 100 rows, which cuts unnecessary computation tremendously. In the end, maintenance time is reduced more than 13X compared with eager maintenance.

5.5 Overhead for Lazy Maintenance

As demonstrated before, lazy maintenance reduces the total response time of updates and queries, even when queries arrive soon after the updates. If we compare the total system time spent on updates, lazy maintenance may be slightly more expensive than eager maintenance, though the extra costs are likely to be hidden from applications.

Compared with eager maintenance, lazy maintenance has a few extra steps. We store delta streams into delta tables and reread them during maintenance and insert maintenance tasks into the persistent task table. After maintenance, we have to clean the delta tables and the task table and remove obsolete information. We also have to maintain version information during updates and exploit version information during maintenance. The overhead of these extra

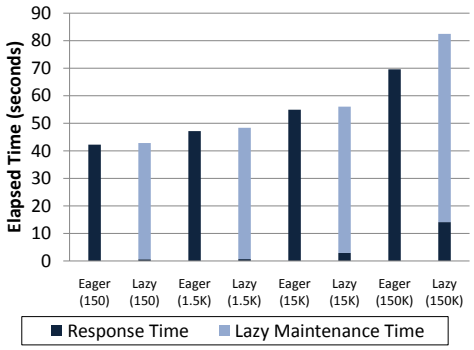


Figure 9: Overhead for Lazy Maintenance

steps is more noticeable with large delta streams.

We repeated the previous experimental setup but increased the number of customer rows that were updated. Figure 9 shows the total amount of work for updating 150, 1.5K, 15K, and 150K customer rows using a single statement, respectively. As the size of updates increases, the update response time of lazy maintenance also increases because more data has to be stored into the delta table and the version store. However, the update response time is still much better than for eager maintenance. The total time of the update and lazy maintenance is higher than that for eager maintenance with large delta streams. Nevertheless, some (or all) of lazy maintenance cost may be hidden from applications.

6. FURTHER CONSIDERATIONS

In this section, we discuss various issues on lazy maintenance and compare other alternative approaches.

6.1 When to Use Lazy Maintenance?

We demonstrated the benefits of lazy maintenance experimentally in the last section. However, we do not recommend replacing eager maintenance with lazy maintenance for *all* materialized views. Each maintenance approach has its benefits and drawbacks and which approach is better for a particular view depends on the application.

Generally speaking, the choice of maintenance strategy for a materialized view depends on the following factors.

- The ratio of updates to queries and how soon queries follow after updates.
- The size of updates (number of rows affected by each update), relative to the view maintenance cost.

Eager maintenance is suitable for materialized views whose base tables are seldom updated and the updates are likely to be followed immediately by queries. It is also suitable for views where the input delta streams tend to be large but maintenance cost is relatively low. On the other hand, lazy maintenance is suitable for views with more frequent small updates and whose maintenance costs are relatively high.

6.2 Recovery and Error Handling

The system may crash while there are still pending maintenance tasks. As part of normal recovery, the persistent task table is recovered and the effects of any in-flight maintenance jobs are undone. Based on the information in the task table, we can rebuild the view maintenance manager and determine what delta streams and versions are needed by the remaining maintenance tasks. The required parts of the delta tables and the version store can then be rebuilt from the database log.

Is it possible that a lazy maintenance task cannot be completed? In SQL Server a materialized view must have a unique clustering key. If this uniqueness constraint is not implied by constraints on the view's base tables, a violation may not be detected until view maintenance time and maintenance fails. This is an inherent problem for all deferred or asynchronous view maintenance algorithms [6, 15] and not unique to our approach. This can be handled by either disallowing lazy maintenance for such views or marking a view as unusable if a violation is detected.

6.3 Alternative Approaches

To compute the change to a view, any deferred, incremental maintenance algorithm needs base table deltas and old versions of base tables. We explicitly store table deltas and rely on an existing version store to deliver the required versions of base tables. We chose this solution for efficiency but there are other alternatives.

On a system with a version store, base table deltas can be handled in three different ways. First, they can be explicitly stored in delta tables (also called table logs). This is an efficient and commonly used solution. Second, they can be extracted from the recovery log. This could very expensive though because the delta rows for a particular transaction are not clustered in the log. Even if we know exactly where they are in the log, simply gathering them could still be expensive. Third, base table deltas can be recovered from the version store. This could also very expensive because the rows updated by a transaction are not clustered in the base table and the version store.

Similarly, on a system with a version store, access to old versions of base tables can also be provided in various ways. First, they can be accessed using the version store. This is precisely what a version store is built for, so this is likely to be an efficient solution. Second, an old version of a base table can be reconstructed from the current version of the table by "undoing" changes made by all transactions that occurred later than the target transaction. The resulting expressions are rather complex and potentially expensive to compute, even when the required base table deltas can be retrieved efficiently from delta tables.

7. RELATED WORK

Issues related to design, exploitation and maintenance of materialized views have received considerable attention in the research community for the last two decades. Materialized views have been adopted in all major commercial database systems, including Oracle [2], IBM DB2 [16, 12], and Microsoft SQL Server [8].

Many incremental view maintenance algorithms have been proposed and studied in [4, 10, 9, 13, 12, 5, 11], all in the context of eager maintenance. They all used the *update delta* paradigm where a set of change tuples (insertions or deletions) is computed and then applied to the materialized view. Eager compensation as a view maintenance technique was proposed and used in [18, 19, 1] for the case when base tables may be distributed across several systems. Another concurrency control technique, based on multi-versioning, was presented in [14] in order to reduce resource contention.

Deferred or asynchronous view maintenance was proposed in [6] and [15], though with different goals. Colby et al. [6] proposed algorithms for deferred view maintenance in order to minimize view downtime. Their algorithms only brought

the view up-to-date to the point of deferred maintenance while our algorithm can bring the view up to any point in time. Salem et al. [15] introduced an algorithm that performs incremental view maintenance as a series of small and asynchronous steps. The goal was to limit contention between the refresh process and concurrent operations. However, as illustrated by experiments, dividing maintenance into many small steps can be quite inefficient and combining small maintenance tasks improves efficiency.

Both papers focused on algorithmic issues and provided none or very limited experimental results. Their algorithms relied on auxiliary tables similar to our delta tables but did not exploit versioning, resulting in rather complex and potentially expensive maintenance expressions. Both papers proposed decomposing view maintenance into separate propagation and apply phases. This idea can also be applied to lazy maintenance.

Oracle supports materialized views with different refresh options and modes. Some, but not all, types of views can be maintained incrementally (fast refreshable); others are recomputed completely on refresh. View logs (delta tables) have to be explicitly defined in order to use fast refresh. Views can be refreshed either *on commit*, that is, at commit time of a update transaction, or *on demand*, when a user explicitly calls a refresh procedure. As far as we have been able to find out, views are not maintained *during* a transaction, which means that a view cannot be used automatically by a query if it is affected by an earlier update in the same transaction.

There has been much research on optimizing maintenance of a set of views. We refer readers to the latest work [17] for a complete list of references. Under lazy maintenance, the same techniques can be applied when scheduling maintenance for multiple views together. The details are beyond the scope of this paper.

8. CONCLUSION

In this paper, we proposed a lazy approach to maintain materialized views. Unlike eager maintenance, we separate view maintenance from update transactions. View maintenance is performed either when the system has free cycles or when a query references the view. Lazy maintenance of materialized views greatly improves update response time while still allowing queries to use materialized views safely. The whole process is totally transparent to applications. View maintenance overhead can be significantly reduced by condensing delta streams and/or combining multiple maintenance tasks.

All or most of the maintenance cost can be hidden from applications. In the common scenario when queries are much more frequent that updates, most queries get the full benefit of materialized views for free and none or only a few queries may experience a delay while a view is maintained.

Although this paper deals only with materialized views, the approach of lazy maintenance can be applied to indexes and other auxiliary data structures as well. Another direction for future work is to incorporate probabilistic query delay into the plan cost when choosing a lazily maintained view during optimization.

9. REFERENCES

[1] D. Agrawal, A. E. Abbadi, A. Singh, and T. Yurek. Efficient view maintenance at data warehouses. In *Proceedings of SIGMOD Conference*, 1997.

[2] R. G. Bello, K. Dias, A. Downing, J. J. F. Jr., J. L. Finnerty, W. D. Norcott, H. Sun, A. Witkowski, and M. Ziauddin. Materialized views in oracle. In *Proceedings of VLDB Conference*, 1998.

[3] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. In *Proceedings of SIGMOD Conference*, 1995.

[4] J. A. Blakeley, P.-Å. Larson, and F. W. Tompa. Efficiently updating materialized views. In *Proceedings of SIGMOD Conference*, 1986.

[5] S. Chen and E. A. Rundensteiner. Gpivot: Efficient incremental maintenance of complex rolap views. In *Proceedings of ICDE Conference*, 2005.

[6] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *Proceedings of SIGMOD Conference*, 1996.

[7] C. A. Galindo-Legaria, S. Stefani, and F. Waas. Query processing for sql updates. In *Proceedings of SIGMOD Conference*, 2004.

[8] J. Goldstein and P.-Å. Larson. Optimizing queries using materialized views: A practical, scalable solution. In *Proceedings of SIGMOD Conference*, 2001.

[9] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *Proceedins of SIGMOD Conference*, 1995.

[10] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of SIGMOD Conference*, 1993.

[11] P.-Å. Larson and J. Zhou. Efficient maintenance of materialized outer-join views. In *Proceedings of ICDE Conference*, 2007.

[12] W. Lehner, R. Sidle, H. Pirahesh, and R. Cochrane. Maintenance of automatic summary tables. In *Proceedings of SIGMOD Conference*, 2000.

[13] I. S. Mumick, D. Quass, and B. S. Mumick. Maintenance of data cubes and summary tables in a warehouse. In *Proceedings of SIGMOD Conference*, 1997.

[14] D. Quass and J. Widom. On-line warehouse view maintenance. In *Proc. of SIGMOD Conference*, 1997.

[15] K. Salem, K. Beyer, B. Lindsay, and R. Cochrane. How to roll a join: Asynchronous incremental view maintenance. In *Proc. of SIGMOD Conference*, 2000.

[16] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, and M. Urata. Answering complex sql queries using automatic summary tables. In *Proceedings of SIGMOD Conference*, 2000.

[17] J. Zhou, P.-Å. Larson, J.-C. Freytag, and W. Lehner. Efficient exploitation of similar subexpressions for query processing. In *Proceedings of SIGMOD Conference*, 2007.

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

[19] Y. Zhuge, H. Garcia-Molina, and J. L. Wiener. The strobe algorithms for multi-source warehouse consistency. In *Proceedings of Conference on Parallel and Distributed Information Systems*, 1996.