How To Roll a Join: Asynchronous Incremental View Maintenance

Kenneth Salem
Dept. of Computer Science
University of Waterloo
kmsalem@uwaterloo.ca

Kevin Beyer
Computer Sciences Dept.
University of Wisconsin
beyer@cs.wisc.edu

Bruce Lindsay
Roberta Cochrane
IBM Almaden Research Center
{bruce,bobbiec}@almaden.ibm.com

Abstract

Incremental refresh of a materialized join view is often less expensive than a full, non-incremental refresh. However, it is still a potentially costly atomic operation. This paper presents an algorithm that performs incremental view maintenance as a series of small, asynchronous steps. The size of each step can be controlled to limit contention between the refresh process and concurrent operations that access the materialized view or the underlying relations. The algorithm supports point-in-time refresh, which allows a materialized view to be refreshed to any time between the last refresh and the present.

1 Introduction

In a relational database, a view is a relation that is derived from other relations. Views, like other relations, can be queried, and new views may be derived from them. A view may be materialized by storing the view's tuples in the database. In many cases, a materialized view can be queried more efficiently than a non-materialized view because its tuples need not be re-derived. The more complex the view, and the more often that view is queried, the more benefit materialization can provide. Materialized views have many applications.[6]

Unless it is updated, a materialized view becomes stale when its underlying relations are modified. Updating a stale materialized view so that it reflects the current state of its underlying tables is called *refreshing* the view. Another option is *point-in-time* refresh, which updates the stale view to a specified intermediate time between its old state and the state reflected by the current underlying tables. Point-in-time refresh is valuable in many applications. However, when point-in-time refresh is performed, care must be taken to leave the materialized view in a transaction-consistent intermediate state.

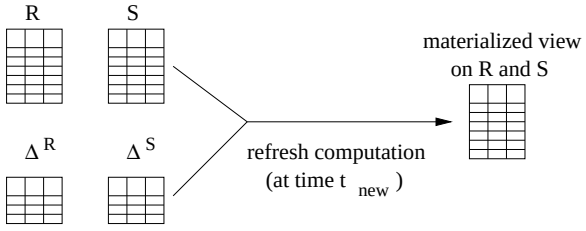

Figure 1: Incremental View Maintenance

Unless the underlying tables are very small, or change very quickly, it is usually desirable to refresh a materialized view incrementally. That is, rather than completely recomputing the view in the desired state, the change from the old state to the desired state is computed and then used to modify the old materialized view. For various classes of views, it is well-understood how to calculate an incremental change.[3, 7, 4] Figure 1 illustrates a refresh operation that updates a materialized view from time t_{old} to time t_{new}. The refresh operation uses the view's underlying tables, R and S, and the incremental changes (ΔR and ΔS) for those tables from t_{old} to t_{new} to refresh the view from t_{old} to t_{new}.

There are several problems with the incremental view maintenance techniques that are part of the current state of the art. First, the incremental refresh operation needs to be executed as an atomic transaction. It must see a consistent snapshot of the underlying tables. The transaction may be long-lived, resulting in contention between the refresh process and concurrent updates to the underlying tables, and between the refresh operation and concurrent reads of the materialized view. As the *refresh interval* (the amount of time between t_{old} and t_{new}) gets longer, as the view definition becomes more complex, and as the number of views to be maintained increases, this problem becomes worse.

Second, the refresh transaction needs to be synchronous with the refresh interval. That is, the transac-

MOD 2000, Dallas, TX USA

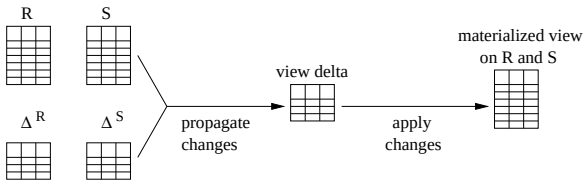

Figure 2: Propagate and Apply

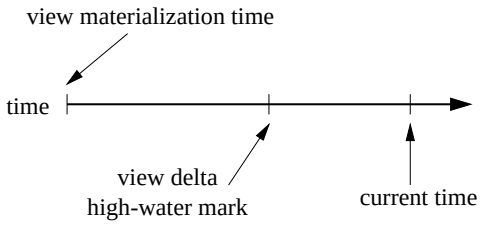

Figure 3: The Contents of the View Delta Table

tion must be performed at a specific time, usually t_{new}, because it needs to see R and S as they exist at that time. This precludes point-in-time refresh. It is not possible to decide at 8:00pm to refresh a materialized view from its 4:00pm state to its 5:00pm state, because at 8:00pm the underlying tables may no longer be as they were at 5:00pm. The decision to refresh the view must be made at 5:00pm and the refresh cost must be paid at 5:00pm, even though more resources may be available later when the load is lighter.

The long transaction problem can be addressed by using very small refresh intervals. For example, the materialized view can be refreshed within every transaction. However, this forces the materialized view to track the current time very closely. For some applications such close tracking is required, but for others it either impractical or undesirable, e.g., when the materialized view represents daily results. The use of short refresh intervals also does not address the synchronization problem. View maintenance costs are paid by each update transaction.

An orthogonal technique involves splitting the refresh computation into a *propagation* phase and an *apply* phase.[4] This is shown in Figure 2. During the propagation phase, changes to the view are computed and stored. Later, during the apply phase, the stored changes (the *view delta*) are used to update the materialized view. This approach breaks a single refresh transaction into separate propagate and apply transactions. It also partially addresses the synchronization problem: the application phase can be delayed, since the view delta can be stored. However, the propagation phase, which includes multiple join queries among the underlying tables and their changes, is still synchronous and atomic.

This paper's contribution is an incremental view maintenance technique called *rolling join propagation* that address both the long transaction problem and the synchronization problem. It has three significant features. First, view delta propagation is asynchronous. That is, the computation of the view delta for a time interval ending at time t_{new} takes place at some time after t_{new}. Second, view delta propagation is treated as a continuous process. The view delta is propagated using a series of small transactions, rather than a single large transaction. The size of each transaction can be be controlled. This control provides a means of limiting the contention between the propagation process and concurrent updates to the underlying tables. Third, the changes recorded in the view delta are *timestamped* to indicate when they should be applied to the view. Timestamps facilitate the control of propagation transaction sizes and enable point-in-time incremental maintenance.

The rolling propagation technique uses separate propagation and apply processes, as was shown in Figure 2. Aside from the usual producer/consumer synchronization, the two processes are completely independent. Either process, or both, can be suspended during periods of high system load, or for other reasons. At any given time, the view delta table contains a complete, timestamped view delta covering the interval from the view's current materialization time to a view delta high-water mark time. This is shown in Figure 3. The view delta may contain additional tuples as well. For example, it may contain partially-computed changes for the time between the high-water mark and the current time. However, the apply process can easily distinguish (and avoid) such tuples using their timestamps. Because the tuples are timestamped, the apply process can, at any time, use the view delta to roll the materialized view forward to any time point up to the view delta's high-water mark. This provides for point-in-time refresh of the materialized view.

2 Definitions and Assumptions

The database consists of a set of tables, which are multisets of tuples. Transactions cause the database to evolve over time. Transactions may insert, delete, and update tuples. An update is modeled as an insertion and a deletion.

A *delta table* describes changes made to another table (possibly a view). We will refer to the non-delta tables as *base tables* when it is necessary to distinguish them from delta tables. If R is a base table, Δ^R will be used to represent the delta table that describes R's changes. Δ^R has the same attributes as R plus two additional attributes: `count` and `timestamp`. A `count` value of $+n$ is used to represent the insertion of n copies of its associated tuple. A value of $-n$ represents the deletion

of n copies of the tuple. The `timestamp` represents the time of the insertion or deletion.

To maintain a uniform notation, each base table is considered to have implicit `count` and `timestamp` attributes. The implicit count associated with each tuple in the base table is +1. (Thus, a base table is represented as the insertion of one copy of each of its tuples into an empty table.) The implicit `timestamp` value is null. The `count` and `timestamp` attributes in base tables exist only for notational convenience. They are not represented explicitly in the database.

The rolling propagation algorithm is presented for select-project-join views, i.e., for views of the form $\pi(\sigma(R^1 \bowtie R^2 \bowtie \ldots \bowtie R^n))$. When it is convenient to do so, the explicit relational operators will be dropped, and the view definition will simply be denoted by $R^1R^2 \ldots R^n$. Although rolling propagation is presented for select-project-join views, it can be extended easily to accommodate views involving union. It can also be extended to accommodate select-project-join views with aggregation by using *summary delta tables*, as described in [8].

In addition to select, project, and join, the algorithm makes use of union and negation operators. The multiset union of tables R^1 and R^2 will be written $R^1 + R^2$. The negation operation, written $-R$, changes the sign of every count in R. The notation $R^1 - R^2$ is used as a shorthand for $R^1 + (-R^2)$.

The transaction history is assumed to be serializable, and the order of transaction commits is assumed to be consistent with the serialization order. This would be the case, for example, in any system that used strict two-phase locking as its concurrency control mechanism. The notation R_a is used to denote the state of table R at time t_a. R_a includes the effects of all transactions that have committed at or before t_a, and does not include any effects from transactions that commit after t_a.

In delta tables, the value of a tuple's `timestamp` attribute is the commit time of the transaction that inserted or most recently updated the tuple. The operation $\sigma_{a,b}$ selects all tuples having timestamps greater than t_a and less than or equal to t_b. The notation $R_{a,b}$ is a shorthand for $\sigma_{a,b}(\Delta^R)$.

The algorithms presented in this paper compute view deltas using *propagation queries*. In general, a computed view delta will be the union of the results of one or more such queries. The propagation queries for a view V have the same form as V's definition, except that one or more of the base tables are replaced by their corresponding delta tables. For example, if V is defined by $\pi(\sigma(R^1 \bowtie R^2 \bowtie \ldots \bowtie R^n))$, a possible propagation query is $\pi(\sigma(R^1 \bowtie R^2_{a,b} \bowtie \ldots \bowtie R^n))$. In this case, R^2 has been replaced by R^2's delta over the time interval from t_a to t_b. The notation $Q^V[i]$ is used to represent the ith relation in a propagation query Q^V for view V. $Q^V[i]$ is either R^i or $R^i_{x,y}$, depending on whether or not R^i has been replaced by its delta table in the query.

Propagation queries produce delta tables. The `timestamp` and `count` attributes for these tables are computed from the `timestamp` and `count` attributes of the tables on which the query is defined. The `count` of a view delta tuple is the product of the `counts` of the tuples from which it is derived. The `timestamp` of a view delta tuple is the minimum of the `timestamps` of the tuples from which it is derived. Section 3.3 describes why the minimum `timestamp` value is chosen.

The notation Q^V_b is used to represent the result of evaluating query Q^V at time t_b. Thus, if $Q^V = R^1R^2_{a,b} \ldots R^n$, then $Q^V_b = R^1_bR^2_{a,b} \ldots R^n_b$. (Similarly, V_b is the state of view V at time t_b.) At times it will be necessary to consider query results such as $R^1_aR^2_{a,b} \ldots R^N_b$, in which different base tables are seen at different times. Such a query will be denoted by $Q^V_{[a,,b]}$, i.e., by explicitly listing the vector of relation times, or by defining $\tau = [a,,b]$ and writing Q^V_τ. Note that $\tau = [a,,b]$ specifies that R^1 is seen at t_a and R^3 is seen at t_b, but it does not specify a time for R^2, which appears as a delta table in Q^V. It is not necessary to specify times for delta tables because they do not evolve over time.

Suppose that V is a three-way join view, Q^V is a propagation query for V, and τ is a vector timestamp. The query result Q^V_τ is said to be *realizable* at time t_x iff the following two conditions hold for all $1 \leq i \leq n$:

- if $Q^V[i] = R^i$, then $\tau[i] = t_x$
- if $Q^V[i] = R^i_{a,b}$, then $t_b \leq t_x$.

For example $R^1_bR^2_{a,b}R^n_b$ $(t_a < t_b)$ is realizable at time t_b, and only at time t_b. The query result $R^1_bR^2_{a,b}R^3_c$ $(t_a < t_b < t_c)$ is not realizable at any time, since R^1 and R^3 are seen at different times. Unless historical snapshots of base relations are maintained or updates to R_1 are prevented between times t_b and t_c (so that $R^1_b = R^1_c$), no serializable transaction can generate this result. Similarly, the query result $R^1_aR^2_{a,b}R^3_a$ $(t_a < t_b)$ is not realizable at any time. In this case, the result uses changes to R^2 up through time t_b, but R^1 and R^3 need to be seen at an earlier time t_a. Note that results of queries that involve only delta tables are realizable at any time after the end of the latest delta time interval in the query.

3 The Rolling Join Propagation Algorithm

This section starts with a presentation of a simple, synchronous algorithm for view delta propagation. The asynchronous, rolling propagation algorithm is

arrived at by successive refinement of the original algorithm. Each refinement addresses a shortcoming of its predecessor.

3.1 Synchronous Propagation

Several techniques for generating view deltas for SPJ views have been described in the literature.[3, 7, 4] For example, if V is $R^1R^2R^3$, then $V_{a,b}$ can be calculated as:

$$\begin{aligned} V_{a,b} &= R^1_{a,b}R^2_bR^3_b + R^1_bR^2_{a,b}R^3_b + R^1_bR^2_bR^3_{a,b} \qquad (1)\\ &\quad -R^1_{a,b}R^2_{a,b}R^3_b - R^1_{a,b}R^2_bR^3_{a,b} - R^1_bR^2_{a,b}R^3_{a,b}\\ &\quad +R^1_{a,b}R^2_{a,b}R^3_{a,b} \end{aligned}$$

The view delta is computed as the union of the results of seven propagation queries. This approach is easily generalized to n-way join views. In the general case, the view delta is computed as the union of $2^n - 1$ query results, one query for each possible combination of base and delta tables that includes at least one delta table.

Except for the all-delta query, all of the queries of Equation 1 are realizable only time t_b, i.e., all of the queries must see the base tables as they exist at t_b. That is, the queries must be executed together as an atomic transaction at time t_b. A propagation query used to compute a view delta $V_{a,b}$ is *synchronous* if it cannot be realized later than t_b. All of the queries of Equation 1 (except the last) are synchronous.

A synchronous propagation technique that requires only n query results to propagate a delta for an n-way join view is described in [7]. For the case $n = 3$, this technique computes $V_{a,b}$ using:

$$V_{a,b} = R^1_{a,b}R^2_bR^3_b + R^1_aR^2_{a,b}R^3_b + R^1_aR^2_aR^3_{a,b} \qquad (2)$$

Note that in each propagation query, base tables to the left of the delta table must be seen at the beginning of the propagation interval (at t_a), and those to the right of the delta table must be seen at the end of the interval, at t_b. Although this technique uses fewer queries to produce the view delta, two of the query results ($R^1_aR^2_{a,b}R^3_b$ and $R^1_aR^2_aR^3_{a,b}$) are not realizable. For this reason, Equation 2 may be less useful in practice that Equation 1. For the purposes of this paper, however, Equation 2 serves as a useful starting point since it involves fewer propagation queries.

3.2 Asynchronous Propagation

The computation described by Equation 1 must be performed atomically at time t_b. Thus, the cost of propagating the view delta from 4:00pm to 5:00pm must incurred in the form of a transaction that runs at 5:00pm. The propagation transaction cannot be delayed. Furthermore, it may be long-lived, particularly if the propagation interval is long or the view is complex. Long-lived propagation transactions can lead to data contention at the base tables from which the view is derived.

The goal of asynchronous propagation is to break the propagation computation into smaller pieces, each of which is performed after the propagation interval. Thus, view delta for the period from 4:00pm to 5:00pm would be computed by a series of smaller transactions that might not begin running until well after 5:00pm.

Asynchronous propagation can be achieved using *compensation*.[12, 13] Consider the query result $Q^V_b = R^1_{a,b}R^2_bR^3_b$ from Equation 1. Suppose that Q^V is evaluated at some later time t_c rather than t_b, resulting in $Q^V_c = R^1_{a,b}R^2_cR^3_c$. Since R^2 and R^3 may have evolved between t_b and t_c, Q^V_b and Q^V_c will, in general, not be the same. However, this problem can be fixed by compensating for any errors caused by changes to R^2 and R^3 made between t_b and time t_c. Compensation involves adding extra changes to the view delta. For example, if Q^V_c includes extra view tuple insertions not found in Q^V_b, they are compensated for by the addition of matching deletions to the view delta.

The difference between Q^V_b and Q^V_c is $Q^V_{b,c}$. That is, the query Q^V can be treated as a view definition and its incremental change from t_b to t_c can be calculated. The compensation required to correct for this difference is exactly $-Q^V_{b,c}$ since every insertion in $Q^V_{b,c}$ becomes a matching deletion in $-Q^V_{b,c}$, and deletions become matching insertions. Since Q^V has the same form as the view V, $Q^V_{b,c}$ (and hence $-Q^V_{b,c}$) can be calculated using the same method used to calculate the view delta, e.g., Equation 2 or Equation 1.

Consider the following example for $V = R^1R^2$. Using the method illustrated in Equation 2, the view delta $V_{a,b}$ can be calculated using

$$V_{a,b} = R^1_{a,b}R^2_b + R^1_aR^2_{a,b}$$

The first query can be moved to time t_c and the second to time t_d, and compensation can be added for each query to correct any errors introduced by the moves. This leads to

$$V_{a,b} = R^1_{a,b}R^2_c - (\mathbf{R^1_{a,b}R^2})_{\mathbf{b,c}} + R^1_dR^2_{a,b} - (\mathbf{R^1R^2_{a,b}})_{\mathbf{a,d}}$$

The compensation queries are shown in bold face, and the asynchronous *forward* queries are not.[1] The first compensation can be calculated by another application of the method of Equation 2:

$$(R^1_{a,b}R^2)_{b,c} = (R^1_{a,b})_{b,c}R^2_c + (R^1_{a,b})_bR^2_{b,c}$$

[1] The term *forward* query will be used to describe propagation queries that involve only a single delta table. The term *compensation* query is used for queries that involve more than one delta table.

```
ComputeDelta(Q, τ_old, t_new) {
  for each i from 1 to n do
    // generate one query for each base relation in Q
    if (Q[i] = R^i) ∧ (τ_old[i] < t_new)
      // Q' is the query to be executed
      let Q' ← Q[1]...Q[i−1]R^i_{old[i],new}Q[i+1]...Q[n]
      // t_exec holds the execution time of the query, which is returned by Execute
      t_exec ← Execute(Q')
      if Q' has any base tables then
        // using the method of Equation 2, tables left of i should be seen at τ_old,
        // tables right of i should be seen at t_new
        let τ_intended ← [old[1], ..., old[i−1],, new, ..., new]
        // tables were actually seen at t_x, so recursively compensate back to the intended time
        ComputeDelta(−Q', τ_intended, t_exec)
      fi
    fi
  od
}
```

Figure 4: Asynchronous Propagation Using Recursive Compensation

Since $R^1_{a,b}$ does not evolve, $(R^1_{a,b})_{b,c}$ is empty, and the compensation expression can be simplified to:

$$(R^1_{a,b}R^2)_{b,c} = R^1_{a,b}R^2_{b,c}$$

Using the same approach, the compensation term $(R^1R^2_{a,b})_{a,d}$ can be found to be $R^1_{a,d}R^2_{a,b}$, which gives the following asynchronous calculation of $V_{a,b}$:

$$V_{a,b} = R^1_{a,b}R^2_c - R^1_{a,b}R^2_{b,c} + R^1_dR^2_{a,b} - R^1_{a,d}R^2_{a,b} \quad (3)$$

Figure 4 shows an asynchronous propagation algorithm called `ComputeDelta` that generalizes this approach to views defined over n base relations. The algorithm takes an n-way propagation query Q an initial vector timestamp τ_{old} and a new time t_{new}. It computes $Q_{old,new}$, i.e., the delta for Q from time τ_{old} to time t_{new}. It can be used to compute a view delta $V_{a,b}$ by setting $Q = V$, $\tau_{old} = [a, a, \ldots, a]$, and $t_{new} = t_b$. For example, execution of `ComputeDelta`$(R^1R^2, [a, a, \ldots, a], t_b)$ will produce the asynchronous calculation shown in Equation 3.

`ComputeDelta` uses a function `Execute` to execute queries. Each `Execute` call is assumed to insert its results into a view delta table in which the view delta is being accumulated. The view delta table itself is not indicated explicitly in Figure 4. Each call to `Execute` performs its query as a separate transaction and returns the commit time of that transaction. Section 5 describes how this is accomplished in our implementation.

3.3 Timestamps

The algorithm of Figure 4 performs asynchronous propagation of the view delta for a propagation interval whose endpoints are defined by the parameters τ_{old} and t_{new}. Over which propagation intervals should a view delta be propagated? There are two conflicting answers to this question.

The length of the propagation interval determines the cost of the propagation queries. Choosing small intervals leads to many small propagation queries. Choosing larger intervals leads to fewer, larger queries. Thus, the interval acts as a parameter that can be tuned to balance query execution overhead against data contention. In addition, the propagation interval determines the time points to which the materialized view can be rolled. If the view is materialized at time t_a and the view delta $V_{a,b}$ is propagated, then it is possible to roll the view forward from t_a to t_b. However, it is not possible to roll the view to any time $t_{b'}$, where $t_a < t_{b'} < t_b$.

Ideally, the propagation and apply processes should be as independent as possible. The choice of a propagation interval can be made independent of the apply process by generating a *timestamp* for each tuple in the propagated view delta. The timestamp indicates commit time of transaction that generated the change. To roll a view from time t_a to time $t_{b'}$, the apply process selects view delta tuples with timestamps in that interval and applies those tuples (only) to the materialized view. With this change, the propagation process can proceed independently of apply. The propagation interval can be used solely as a tuning parameter for the propagation process, allowing size of the propagation queries to be controlled. The propagation process can be implemented as a loop that continuously generates timestamped view delta tuples

```
Propagate(V, t_initial) {
  t_cur ← t_initial
  do forever
    choose a propagation interval length δ
    ComputeDelta(V, [cur, ..., cur], t_cur + δ)
    t_cur ← t_cur + δ
  od
}
```

Figure 5: A Continuous, Asynchronous Propagation Process

for successively later time intervals. Such a process, called `Propagate` is shown in Figure 5. `Propagate` generates a view delta starting at a specified time $t_{initial}$. The variable t_{cur} tracks the view delta high-water mark, which was illustrated in Figure 3. At the conclusion of every iteration of `Propagate`, the view delta is accurate for the interval from $t_{initial}$ to t_{cur}.

Each base table's delta includes a timestamp attribute indicating the commit times of the changes it records. As was noted in Section 2, a view delta tuple's timestamp is the *minimum* of the timestamps of the tuples that joined to produce it.[2] That the minimum timestamp is the correct one to choose may be counter-intuitive. In Section 4 this choice is shown to be correct. However, the following examples may provide some intuituion.

Suppose that $V = R^1R^2$ and that V_0 contains a tuple r^1r^2, where r^1 is a tuple found in R_0^1 and r^2 is a tuple found in R_0^2. At time t_a, where $t_0 < t_a < t_1$, tuple r^1 is deleted from R^1. At time t_b, where $t_a < t_b < t_1$, tuple r^2 is deleted from R^2. The effect of these deletions should be the deletion of r^1r^2 from the view at time t_a.

The execution of `ComputeDelta(`V`,[`$0,\ldots,0$`],`t_1`)` (see Figure 4) calculates $V_{0,1}$ as

$$V_{0,1} = R^1_{0,1}R^2_c - R^1_{0,1}R^2_{1,c} + R^1_dR^2_{0,1} - R^1_{0,d}R^2_{0,1}$$

where t_c and t_d are query execution times later than t_1. Query $R^1_{0,1}R^2_c$ will produce an empty result, since r^2 has already been deleted from R^2 at time t_c. Query $R^1_dR^2_{0,1}$ will also produce an empty result for a similar reason. Query $-R^1_{0,1}R^2_{1,c}$ will be empty because R^2 does not change between t_1 and t_c in this example. Query $-R^1_{0,d}R^2_{0,1}$ will find the tuple $(t_a,-1,r^1)$ in $R^1_{0,d}$ and the tuple $(t_b,-1,r^2)$ in $R^2_{0,1}$. (The -1's are the values of the `count` attribute.). The compensation query will join those tuples and add $(t_a,-1,r^1r^2)$ to the view delta, since the minimum timestamp (t_a) and the negated product of the counts will be used. This is the desired result.

Now consider an insertion scenario. Suppose that tuple x^1 is inserted into R^1 at time t_a and tuple x^2 is inserted into R^2 at time t_b. If x^1 and x^2 join, the effect should be the insertion of x^1x^2 into V at time t_b. `ComputeDelta(`V`,[`$0,\ldots,0$`],`t_1`)` calculates the view delta as follows. Query $R^1_{0,1}R^2_c$ will find the tuple $(t_a,+1,x^1)$ in $R^1_{0,1}$, and it will find x^2 in R^2_c. Thus, it will add $(t_a,+1,x^1x^2)$ into the view delta. For a similar reason, $R^1_dR^2_{0,1}$ will add $(t_b,+1,x^1x^2)$ to the view delta. Query $-R^1_{0,1}R^2_{1,c}$ is empty, but $-R^1_{0,d}R^2_{0,1}$ will find insertion tuples in $R^2_{0,1}$ and $R^1_{0,d}$. Since it chooses the minimum timestamp, it will add $(t_a,-1,x^1x^2)$ to the view delta. In net effect, the insertion and deletion at t_a cancel each other out, leaving only the insertion at time t_b. Again, this is the desired result.

3.4 Rolling Propagation

The `Propagate` algorithm from Figure 5, together with timestamps, addresses the incremental view maintenance issues discussed in Section 1. The rolling propagation algorithm is a refinement of `Propagate`. The principal difference between the two is that rolling propagation provides more control over the sizes of the propagation queries.

The `ComputeDelta` procedure, on which `Propagate` is based, provides one tunable parameter: the length of the propagation interval. All forward queries use the same interval. However, in many cases the base tables from which a view is derived will be updated at different rates. For example, consider a star schema in which the central fact table is frequently updated and the surrounding dimension tables are rarely updated. If the propagation interval is the same for all forward queries, the forward queries for the fact table will be much larger than the forward queries for the dimension tables. The rolling propagation algorithm allows a different interval to be used for each base table. Thus, rolling propagation provides n independent tunable parameters, rather than one.

Rolling propagation also tends to generate fewer, larger propagation queries than `Propagate` does. Although both algorithms are based on `ComputeDelta`, rolling propagation defers the compensations for some forward queries and combines them with compensations for later queries. As it result, it makes fewer calls to `ComputeDelta` than `Propagate` does.

Figures 6,7,8, and 9 constitute a graphical explanation of the rolling propagation algorithm and its relationship to `ComputeDelta` and `Propagate`. Suppose that $V = R^1R^2$. Figure 6 shows a coordinate space with one time axis for R^1 and one for R^2. The point t_a on the R^1 axis represents R^1_a, i.e., R_1 as it exists at t_a. Time t_0 on each axis represents the relation cre-

[2] Recall that base tables are considered have implicit null timestamps, so only timestamps from the delta tables are considered when choosing the minimum. Every maintenance query involves at least one delta table.

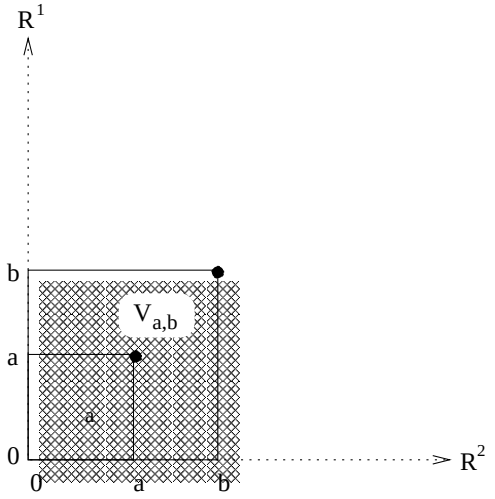

Figure 6: Graphical Representation the Evolution of R^1R^2

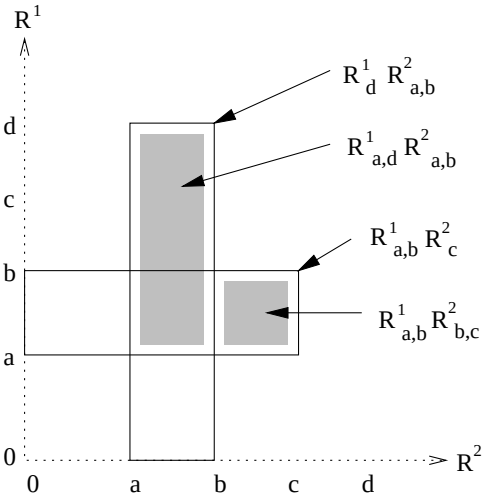

Figure 7: `ComputeDelta`$(R^1R^2,[a,\ldots,a],t_b)$

ation times. The region below and to the left of the point (a,a) represents the join $R^1_aR^2_a = V_a$. Similarly, the region below and to the left of (b,b) represents V_b, and the view delta $V_{a,b}$ is the L-shaped cross-hatched region.[3]

`ComputeDelta`$(V,[a,\ldots,a],t_b)$ generates $V_{a,b}$ using the four propagation queries shown in Equation 3. In Figure 7, each of these queries is represented diagrammatically by a rectangular region. For example, $R^1_{a,b}R^2_{b,c}$ is represented by the region bounded by t_a and t_b on the R^1 axis and by t_b and t_c on the R^2 axis. Similarly $R^1_dR^2_{a,b}$ is bounded by t_0 and t_d on the R^1 axis and t_a and t_b on the R^2 axis. The forward queries are represented by unshaded rectangles, and the shaded regions are the compensation queries. Note that the net effect of these four queries is exactly the L-shaped region representing $V_{a,b}$ that was shown in Figure 6. The portion of $R^1_dR^2_{a,b}$ that protrudes above the line R^1_b, which is not part of the L-shaped region, is compensated for. So is the portion of $R^1_{a,b}R^2_c$ that protrudes beyond the line R^2_b. The compensation of the region $R^1_{a,b}R^2_{a,b}$ corrects for the double counting of that region by the two forward queries, which overlap there.

The `Propagate` process from Figure 5 generates consecutive view deltas by repeated application of `ComputeDelta`. This is illustrated graphically in Figure 8, which shows the sequential calculation of $V_{a,b}$, $V_{b,c}$, and $V_{c,d}$ by three iterations of `Propagate`. Each of these three calculations is identical in form to the calculation shown in Figure 7.

Finally, the behavior of the rolling propagation

[3]It may be helpful to think of a relation like R^1 as consisting of a log of changes since t_0. In net effect, the portion of this log from t_0 to t_a is the same as R^1_a.

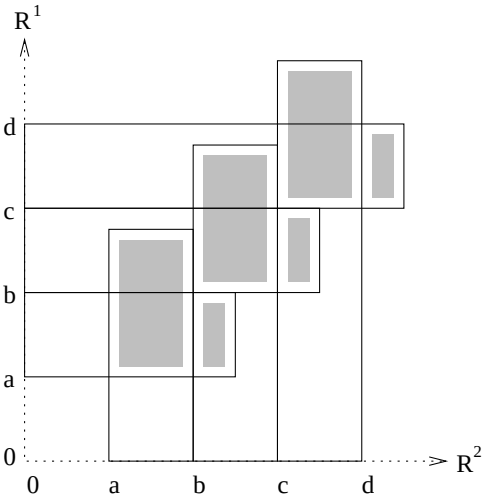

Figure 8: Three Iterations of `Propagate`(R^1R^2,t_a)

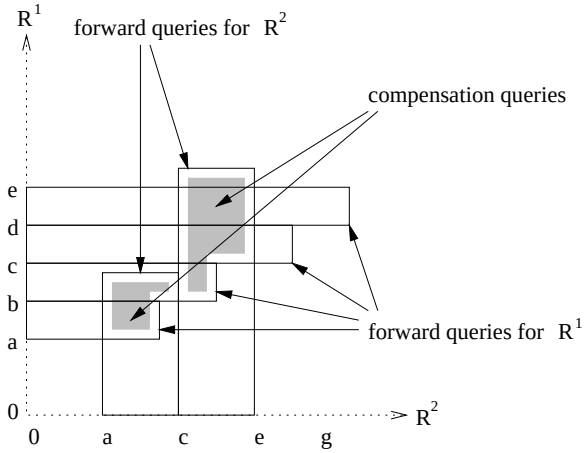

Figure 9: Rolling Propagation

process is shown in Figure 9. Compare this to `Propagate` in Figure 8. Note that in Figure 9, the forward queries for R^2 are wider than the forward queries for R^1. This reflects the fact that the rolling propagation algorithm allows a different propagation interval to be used for each relation's forward queries.

The use of different propagation intervals for each base relation allows for a great deal of control over the propagation process. However, it also complicates the structure of the view delta computation. In `Propagate` (Figure 8), the propagation queries used to calculate $V_{a,b}$ are independent of the queries that calculate $V_{b,c}$. That is, the calculation of the view delta from $V_{b,c}$ does not start until the calculation of $V_{a,b}$ has finished. Rolling propagation, however, does not partition the view delta calculations so cleanly. As a result, it is less obvious when the calculation of any portion of the view delta has been completed. It is also less obvious how to generate compensation for the forward queries. In `Propagate`, the compensation queries depend only on which portion of the view delta (e.g., $V_{a,b}$ vs. $V_{b,c}$) is currently being calculated.

The rolling propagation algorithm's solution to these problems is based on the observation that compensation queries must cover exactly those portions of the R^1R^2 space that are (or will be) double-counted by forward queries. From Figure 8, it can be seen that `Propagate`'s compensations accomplish this. Figure 9 shows that rolling propagation's compensation queries also accomplish it, but with a different set of compensations.

Figure 10 shows the `RollingPropagate` process. Each iteration of `RollingPropagate` performs a single forward query plus some compensations. The variable $t_{fwd}[i]$ keeps track of the progress of forward queries for R^i. The variable $querylist[i]$ keeps track of the progress of compensation for R^i's queries. Specifically, $querylist[i]$ lists the forward queries for R^i that have not yet been completely compensated for.[4] The variable $t_{comp}[i]$ tracks the oldest uncompensated query in $querylist[i]$ (or $t_{fwd}[i]$ if $querylist[i]$ is empty). For example, after the queries shown in Figure 9, both $t_{fwd}[1]$ and $t_{fwd}[2]$ would be t_e. $querylist[2]$ would be empty, and $t_{comp}[2] = t_{fwd}[2] = t_e$, since both of R^2's forward queries have been compensated for. However, $querylist[1]$ would include the two most recent forward queries for R^1 (those with propagation intervals $R^1_{c,d}$ and $R^1_{d,e}$), since portions of these queries that require compensation have not yet been compensated for. $t_{comp}[1] = t_c$, the beginning of the oldest query in $querylist[1]$.

When `RollingPropagate` performs a forward query $Q = R^1 \ldots R^{i-1} R^i_{x,y} R^{i+1} \ldots R^n$ for R^i, it compensates for forward queries of lower-numbered relations with which Q overlaps. Thus, when $V = R^1R^2$, no compensation is performed for R^1's forward queries, and R^2's forward queries perform compensation that accounts for overlap between R^2's queries and R^1's. Ideally, the necessary compensation can be accomplished with a single call to `ComputeDelta`. In general, however, the region for which compensation is required is not rectangular. This is true of both of the compensation queries shown in Figure 9. In these cases, the region to be compensated is divided into rectangular sub-regions, and one call to `ComputeDelta` is used to perform the compensation for each sub-region. This is accomplished by the `repeat/until` loop in `RollingPropagate`.

The rolling propagation computation is more flexible that that of `Propagate`. An obvious question, however, is how to determine the view delta high-water mark. (Recall from Figure 3 that the view delta high-water mark indicates which portion of the view delta has been completely calculated.) For example, given the state of the computation shown in Figure 9, how far forward from t_a can the apply process roll the materialized view?

`RollingPropagate(`V`,`t_a`)` will have completely calculated the view delta from t_a to t_b once all of the forward queries that overlap with that interval have been completely compensated. The `RollingPropagate` algorithm tracks the compensation of forward queries through the $t_{comp}[i]$ variables. After any iteration of the algorithm, the view delta high water mark is determined by the minimum value of $t_{comp}[i]$ over all of the relations R^i. Thus, after the computation in Figure 9, $t_{comp}[1] = t_c$, $t_{comp}[2] = t_e$, and the view delta high water mark is at t_c.

4 Correctness

There are many equivalent ways to represent a view delta. For example, a insertion can be represented using a single tuple with a count of $+1$, as two tuples, one with a count of $+2$ and another with a count of -1, and so on. The *net effect* operator, ϕ, maps equivalent tables into a canonical form.

Definition 4.1
The net effect *of a table R, written $\phi(R)$, is the table obtained from R by the following steps. First, R is grouped on all attributes except* `count` *and* `timestamp`. *Within each group,* `count` *values are aggregated using addition, and the group's* `timestamp` *becomes null. Finally, tuples for which count is equal to zero are eliminated.*

Note that, when applied to a base table, the net effect operation turns a multiset into a set, with tuple multiplicity in the original multiset represented by `count` values in the net effect. The net effect operation

[4] A forward query has been completely compensated for if every region of the query that will overlap another forward query has been compensated for.

```
RollingPropagate(V, t_initial) {
  // t_fwd[i] tracks the progress of forward queries for R^i
  // querylist[i] lists forward queries for R^i that have not been fully compensated
  // t_comp[i] tracks oldest query in querylist[i]
  for each 1 ≤ i ≤ n : t_fwd[i] ← t_initial, t_comp[i] ← t_initial, querylist[i] ← ∅
  do forever
    choose a base relation R^i with the smallest t_fwd[i]
    PruneQueryLists(t_fwd[i])
    choose a propagation interval length δ for R^i
    // perform forward query for R^i.  t_e holds the execution time, which is returned by Execute
    t_e ← Execute(R^1 ... R^{i-1} R^i_{fwd[i],fwd[i]+δ} R^{i+1} ... R^n)
    if i < n then insert R^1_e ... R^{i-1}_e R^i_{fwd[i],fwd[i]+δ} R^{i+1}_e ... R^n_e into querylist[i]
    // generate call(s) to ComputeDelta to compensate for the forward query
    if i > 1 // no need to compensate for R^1's forward queries
      repeat
        δ' ← min(δ,CompInterval(R^i, t_fwd[i]))
        // compensate for overlap with forward queries for R^1 through R^{i-1}.
        τ_d ← [CompTime(R^1, t_fwd[i]),...,CompTime(R^{i-1}, t_fwd[i]),,t_e,...,t_e]
        ComputeDelta(R^1 ... R^{i-1} R^i_{fwd[i],fwd[i]+δ'} R^{i+1} ... R^n, τ_d, t_e)
        t_fwd[i] ← t_fwd[i] + δ'
        δ ← δ − δ'
      until δ = 0
    fi
  od
}

// determine how wide a compensation for R^i can be, starting at t,
// without becoming non-rectangular
CompInterval(R^i, t) {
  let t_e be the smallest execution time greater than t of all queries appearing in
    querylist[1] ∪ ... ∪ querylist[i − 1]
  return(t_e − t)
}

// determine how far back a compensation query at t should compensate R^i
CompTime(R^i, t) {
  let R^1_e ... R^{i-1}_e R^i_{x,y} R^{i+1}_e ... R^n_e be the query from any querylist[i]
    with the smallest execution time t_e that is greater than t
  return(t_x)
}

// remove fully-compensated queries from the query lists, and update t_comp[i]
PruneQueryLists(t) {
  for each R^i do
    remove from querylist[i] all queries with execution times less than or equal to t
    if querylist[i] is empty
      t_comp[i] ← t_fwd[i]
    else
      let R^1_e ... R^{i-1}_e R^i_{x,y} R^{i+1}_e ... R^n_e have the smallest execution time in querylist[i]
      t_comp[i] ← t_x
  od
}
```

Figure 10: The Rolling Propagation Process

has the following properties:

$$\begin{aligned}\phi(\phi(R)) &= \phi(R)\\ \phi(R+S) &= \phi(\phi(R)+S) = \phi(R+\phi(S))\\ &= \phi(\phi(R)+\phi(S))\\ \phi(RS) &= \phi(R)\phi(S)\end{aligned}$$

Provided that the selection condition does not involve `count` or `timestamp` and that a projection does not eliminate `count` or `timestamp`, the net effect operation also has the following properties:

$$\begin{aligned}\phi(\sigma(R)) &= \sigma(\phi(R))\\ \phi(\pi(R)) &= \phi(\pi(\phi(R)))\end{aligned}$$

Intuitively, a table Δ is a delta table for R if it can be used to "roll" the state of R from one time to another. That is, if $\phi(R_i + \Delta) = \phi(R_j)$, then Δ is a delta table for R from time t_i to time t_j. However, the delta tables used by the rolling compensation algorithm are more flexible than this because they include timestamps. With such a delta table, it is possible to select tuples having timestamps within a particular time window and then use the selected tuples as a view delta over that window. This is captured by the following definition:

Definition 4.2
The table Δ *is a* timed delta table *for* R *from* t_i *to* t_j *(*$t_i < t_j$*) iff for all times* t_a *and* t_b *such that* $t_i \leq t_a < t_b \leq t_j$ *it satisfies the following condition:*

$$\phi(\sigma_{a,b}(\Delta) + R_a) = \phi(R_b)$$

Timed delta tables can be split to produce timed delta tables over smaller time intervals. Consecutive timed delta tables can also be combined to produce a single timed delta table over a larger interval.

Lemma 4.1 *If* Δ *is a timed delta table for* R *from* t_i *to* t_j*, and* t_x *is a time between* t_i *and* t_j*, then* $\sigma_{i,x}(\Delta)$ *is a timed delta table for* R *from* t_i *to* t_x*, and* $\sigma_{x,j}(\Delta)$ *is a timed delta table for* R *from* t_x *to* t_j*.*

Lemma 4.2 *If* Δ *is a delta table for* R *from* t_i *to* t_x*, and* Δ' *is a delta table for* R *from* t_x *to* t_j*, then* $\Delta + \Delta'$ *is a timed delta table for* R *from* t_i *to* t_j*.*

The following theorems state that the `ComputeDelta`, `Propagate`, and `RollingPropagate` procedures are correct. Because of space limitations, they are presented here without proof. Proofs can be found in the extended version of this paper.[11]

Theorem 4.1 *Let* Δ *be the result of executing* `ComputeDelta`*(*V*,*τ_a*,*t_b*) for a view* V*.* Δ *is a timed delta table for* V *from time* τ_a *to time* t_b*.*

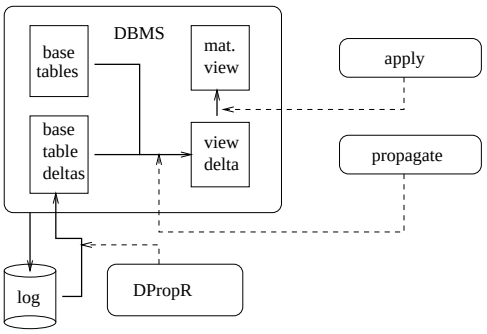

Figure 11: View Maintenance Architecture

Theorem 4.2 *Let* Δ *be the delta table that has been produced by* `Propagate`*(*V*,*t_a*) after some number of complete iterations, and let* t_b *be the value of* t_{cur} *in* `Propagate`*.* Δ *is a timed delta table for* V *from time* t_a *to time* t_b*.*

Theorem 4.3 *Let* Δ *be the table produced by* `RollingPropagate`*(*V*,*t_a*) and let* t_b *be the minimum value (over all* i*) of* $t_{comp}[i]$*. At all times,* $\sigma_{a,b}(\Delta)$ *is a timed view delta for* V *from* t_a *to* t_b*.*

5 Implementation Issues

A prototype of the rolling propagation algorithm has been implemented using a set of external drivers around the DB2 relational database engine. Figure 11 gives a high-level view of the prototype's architecture. Solid lines represent data flow. The dashed lines are used to indicate which driver controls each data flow.

The external drivers use a set of control tables maintained in the database engine. The control tables identify the tables associated with each materialized view, including the view delta table, the underlying base tables, and their delta tables. The control tables also record the current view materialization time and the view delta high-water mark. The `propagate` driver implements the rolling propagation algorithm to populate view delta tables. The `apply` driver implements incremental point-in-time refresh by applying changes recorded in the view delta table.

One issue that arose during the design of the prototype was the method of populating the base table deltas. There are two options. One is to define triggers on each base table R, so that updates, insertions, and deletions will trigger the insertion of change records into Δ^R. The other option is to populate Δ^R by extracting changes from the database engine's transaction log.

The trigger method is simpler to implement and it does not require knowledge of the database engine's log format. However, it has several disadvantages. One is that it expands the update footprint of any transaction that modifies R to include Δ^R. Thus, the transaction

can conflict with propagation queries (initiated by the `propagate` driver) that read the delta table. Note that if a materialized view depends on R, every propagation transaction will read either R or Δ^R. A more serious problem for the trigger-based approach is the generation of timestamps for the tuples in Δ^R. As was noted in Section 2, the timestamp of a delta tuple is supposed to identify the serialization order of the transaction that performed the change. In many systems (e.g., those that use two-phase locking for concurrency control), the serialization order of a transaction is not known until it commits. This means that a trigger that fires at the time of an insertion or deletion into R will not be able to attach an appropriate timestamp.

The prototype view maintenance implementation uses a tool called DB2 DataPropagator (DPropR) to populate the base delta tables directly from the transaction log. DPropR tags each delta tuple with a unique transaction identifier. In addition, DPropR maintains a separate global table, called the *unit-of-work* table, which maps the identifier of each relevant transaction to its commit sequence number and commit timestamp. Both the sequence number and the timestamp are consistent with the transaction serialization order, but the sequence numbers are unique, while commit timestamps may not be. DPropR populates the unit-of-work table as it encounters commit records of relevant transactions in the log. A transaction is relevant if it has made a change to one of the view's underlying tables.

The `propagate` process obtains commit sequence numbers and timestamps for the tuples in Δ^R by joining Δ^R with the unit-of-work table on the unique transaction identifier. Internally, `propagate` uses commit sequence numbers as "times". However, it records both a sequence number and a timestamp in each view delta tuple so that real times can be used to specify propagation intervals and materialized view states.

A similar approach might be possible in a trigger-based system if the database engine provides commit triggers. A commit trigger could potentially be used to record each transaction's serialization point in a unit-of-work table, provided that the trigger has some means of determining the serialization order. However, unless relevant transactions could somehow be identified, the unit-of-work table would have to record serialization times for all update transactions.

A related issue that must be addressed by the `propagate` process is how to determine the evaluation time of a propagation query. The function `Evaluate` used by `ComputeDelta` and `RollingPropagate` is expected to return the serialization time of the transaction in which the propagation query is evaluated. In the prototype, `propagate` determines the commit sequence number of such a transaction by forcing it to write a unique value into a special global table. The special table has an associated delta table, which is populated by DPropR. Once the transaction has completed, `propagate` waits for DPropR to capture its special table update. From the captured update, `propagate` can determine the maintenance query's transaction identifier. The unit-of-work table can then be used to determine its serialization time.

6 Related Work

Many view maintenance algorithms have been described. An overview of this work can be found in [6], which also discusses applications. The algorithms described in [3, 7, 4] are probably most closely related to the rolling propagation algorithm. The algorithms in [3] handle select/project/join views only, and require pre-update snapshots of the base table. Those of [7] handle a broader class of view definitions, including views with union, recursion, and negation, but they require both pre-update and post-update snapshots of the base tables. [4] presents algorithms for deferred maintenance of views that may involve union and difference operations in addition to select, project and join. All of these approaches are synchronous: incremental maintenance queries must see base tables either at the beginning of the propagation interval or at the end. [4] also proposes the decomposition of the view maintenance problem into separate propagate and apply phases. Another technique, based on multi-versioning, for reducing contention between materialized view updates and concurrent reads is presented in [10]. A description of the implementation of incremental view maintenance in a commercial relational database system can be found in [2].

The summary-delta table method was proposed in [8] for incremental maintenance of relational views that involve aggregation. A summary-delta table records the net change to an aggregate over a particular time window. The rolling propagation technique can be extended to support views with aggregation by using summary-delta tables.

Compensation as a view maintenance technique was proposed in [12]. The Eager Compensation Algorithm described in [12] operates in a warehousing environment in which the base tables and the materialized views are located in different systems. The warehouse (where the view is located) is notified explicitly of each update to the base tables. The warehouse responds to an update notification by issuing a maintenance query to compute incremental change necessary to reflect the update in the materialized view. If further updates occur while the maintenance query is pending, the warehouse modifies the maintenance queries for those updates so that the updates' effects on the pending queries will be compensated for. Compensation is also

used in [13] and [1] in a more general environment in which the base tables may be distributed across several systems.

Self-maintainable views are views that can be incrementally maintained using only the base deltas, and not the base tables themselves. Self-maintainable views were introduced in [5], and [9] presents techniques for expanding materialized views so that they become self-maintainable.

7 Conclusion

Most incremental view maintenance techniques are synchronous. They tie the incremental refresh effort to the refresh time, since the base tables must be seen in a particular state. Compensation can be used to decouple the refresh effort from the refresh time, while the database continues to evolve.

Rolling propagation is a compensation-based technique for asynchronous incremental view maintenance. Unlike other compensation-based techniques, it provides explicit control over the granularity of the view maintenance transactions. Rolling propagation also completely decouples the propagation of a view changes from the application of those changes to the materialized view. Applications can control point-in-time refresh of the materialized view, while view delta propagation is tuned independently to suit the requirements of the underlying database system.

8 Acknowledgements

The authors would like to thank Beth Hamel, Hamid Pirahesh, Jay Shanmugasundaram, and Richard Sidle for their help and comments.

References

[1] D Agrawal, A. El Abbadi, A. Singh, and T. Yurek. Efficient view maintenance at data warehouses. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 417–427, 1997.

[2] Randall Bello, Karl Dias, Alan Downing, James Feenan, Jim Finnerty, William Norcott, Harry Sun, Andrew Witkowski, and Mohamed Ziauddin. Materialized views in Oracle. In *Proceedings of the International Conference on Very Large Data Bases*, pages 659–664, 1998.

[3] José A. Blakeley, Per-Åke Larson, and Frank Wm. Tompa. Efficiently updating materialized views. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 61–71, 1986.

[4] Latha S. Colby, Timothy Griffin, Leonid Libkin, Inderpal Singh Mumick, and Howard Trickey. Algorithms for deferred view maintenance. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 469–480, 1996.

[5] Ashish Gupta, H. V. Jagadish, and Inderpal Singh Mumick. Data integration using self-maintainable views. In *International Conference on Extending Database Technology*, pages 140–144, 1996.

[6] Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques, and applications. *Bulletin of the IEEE Technical Committee on Data Engineering*, 18(2):3–19, 1995.

[7] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 157–167, 1993.

[8] Inderpal Singh Mumick, Dallan Quass, and Barinderpal Singh Mumick. Maintenance of data cubes and summary tables in a warehouse. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 100–111, 1997.

[9] Dallan Quass, Ashish Gupta, Inderpal Singh Mumick, and Jennifer Widom. Making views self-maintainable for data warehousing. In *Conference on Parallel and Distributed Information Systems*, pages 158–169, 1996.

[10] Dallan Quass and Jennifer Widom. On-line warehouse view maintenance. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 393–404, 1997.

[11] K. Salem, K. Beyer, B. Lindsay, and R. Cochrane. How to roll a join: Asynchronous incremental view maintenance. Technical Report CS-2000-6, Dept. of Computer Science, University of Waterloo, February 2000.

[12] Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, and Jennifer Widom. View maintenance in a warehousing environment. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 316–327, 1995.

[13] Yue Zhuge, Hector Garcia-Molina, and Janet Wiener. The strobe algorithms for multi-source warehouse consistency. In *Conference on Parallel and Distributed Information Systems*, 1996.