CS-TR-3878

Reduction of Materialized View Staleness Using Online Updates[*]

Alexandros Labrinidis[†] labrinid@cs.umd.edu

Nick Roussopoulos[‡] nick@cs.umd.edu

Department of Computer Science, and
Institute for Systems Research,
University of Maryland
College Park, MD 20742

February 1998

Abstract

Updating the materialized views stored in data warehouses usually implies making the warehouse unavailable to users. We propose *MAUVE* , a new algorithm for *online* incremental view updates that uses timestamps and allows consistent read-only access to the warehouse while it being updated. The algorithm propagates the updates to the views more often than the typical once a day in order to reduce *view staleness*.

We have implemented *MAUVE* on top of the Informix Universal Server and used a synthetic workload generator to experiment with various update workloads and different view update frequencies. Our results show that, all kinds of update streams benefit from more frequent view updates, instead of just once a day. However, there is a clear maximum for the view update frequency, for which *view staleness* is minimal.

1 Introduction

Data warehouses contain data replicated from several external sources, collected to answer decision support queries. The replicated data is often copied in replica tables in the warehouse. The degree of replication is further extended by introducing other derived data to facilitate query processing and maintenance. These derived data include all kinds of indices, multidimensional materialized views or partially materialized views, summary tables and aggregate views such as the data cube, and so on. We refer to all these with the most general term "materialized views" ([Rou98]).

[†]Contact author.

[‡]Also with the Institute for Advanced Computer Studies, University of Maryland.

[*]This work was supported by the Terrestrial Wireless Extensions of Direct Broadcast project, a Hughes Network Systems and State of Maryland cooperative Industry-University contract from the Maryland Industrial Partnerships Program (MIPS).

When data on the external sources change, updates are sent to the warehouse, which has to perform a *refresh* operation. Except for updating the base data tables, derived data also need be updated in order for the warehouse to reach a fully consistent state. The two issues at hand are *how* to implement the refresh operation, and *when* to refresh the warehouse ([CD97]).

View Maintenance deals with the issue of how to perform the refresh operation on materialized views, once the underlying data has changed ([GM95]). The simple solution of recomputing the entire materialized view from scratch is not suitable for most cases, and instead *incremental algorithms* ([BALT86, RK86, CW91, Rou91, QW91, GMS93, GL95, RCK$^+$95, ZGMHW95, ZGMW96, CGL$^+$96, AASY97, GLT97, QW97]) have been proposed to compute only the changes to the materialized view given the updates on the base tables. The choice of which algorithm to choose can be left to the query optimizer ([Vis98]).

The *when* to refresh issue, is closely associated with the overhead that the view update algorithm places on the warehouse. In most cases, the update algorithms render the warehouse *off-line* (i.e. no queries are allowed to run concurrently with the update process since they would possibly access inconsistent data), and as such are usually scheduled overnight. However, the new world order of globalization in operations takes away the luxury of refreshing the warehouse during the night, because it is always daytime in some part of the world. One solution is to try to minimize this downtime, and make the effects of the warehouse being off-line as little as possible ([CGL$^+$96]). An even better solution is to eliminate downtime altogether, by using an *online* algorithm, that allows read-only queries to access the warehouse while it is being updated ([QW97]).

Even with the best online view maintenance algorithm, the decision of *when* to update is not straightforward. If we wish to update the materialized views as soon as changes to the base tables arrive, this *immediate maintenance* imposes a significant overhead both to the update process, and to the rest of the warehouse users. As an alternative, we can use *deferred view maintenance*, which will allow the view data to become *stale* (i.e. inconsistent with the view definition) and perform the update at some time in the future.

Deferred view maintenance raises two concerns. First of all, in the case that we want warehouse readers to always see consistent data (the default case for most applications), we must come up with a way to "filter out" the parts that are inconsistent, possibly by supplying readers with an older version of the entire warehouse. Secondly, we must strike balance between grouping many view updates together for better performance, and not letting view data become too stale.

In this paper we present *MAUVE*[1], a new online algorithm for incrementally updating materialized views. *MAUVE* uses versioning to allow read-only warehouse queries to run concurrently with warehouse update jobs and always "see" the warehouse at a completely consistent state. Incoming update streams are split up into "chunks" of updates. For each chunk, *MAUVE* first applies all of the updates to the base tables and then propagates these updates to the views.

By controlling the *chunk size*, we affect the *view staleness* (the time it takes for the base table updates to

[1]*MAUVE* stands for Multi-version Algorithm for Updating materialized Views onlinE.

"reach" the views) for the warehouse. Since the view update phase in a sense takes processing time "away" from base table updates, having a relatively small chunk size can have an overwhelming delay on the update process, whereas a big chunk size will let the view data become too stale. Finding the chunk size that leads to an *optimal* view staleness for the warehouse will guarantee a good tradeoff.

This paper contributes to the work on view maintenance by presenting a new online view update algorithm, and by establishing view staleness as a key metric to use when deciding how often to propagate the updates to the materialized views.

The rest of this paper is organized as follows. Our online view update algorithm is presented in Section 2. View Staleness is defined and calculated in Section 3. Section 4 contains our experiments and Section 5 discusses related work. Finally, Section 6 has our conclusions and plans for future work.

2 View Update Algorithm

MAUVE is an online algorithm for incrementally updating materialized views that uses versioning. The warehouse refresh process, "fed" by the incoming *update stream*, is continuously applying the updates to the base tables. When certain conditions are met (e.g. number of updates reaches a predefined threshold or certain time has passed since the last view update) *MAUVE* will interrupt the regular processing of base table updates in order to propagate these updates to the views (see Figure 1). Versioning allows readers, running concurrently with the update process, to "filter out" pending or not fully propagated updates, thus always seeing the warehouse at a completely consistent state.

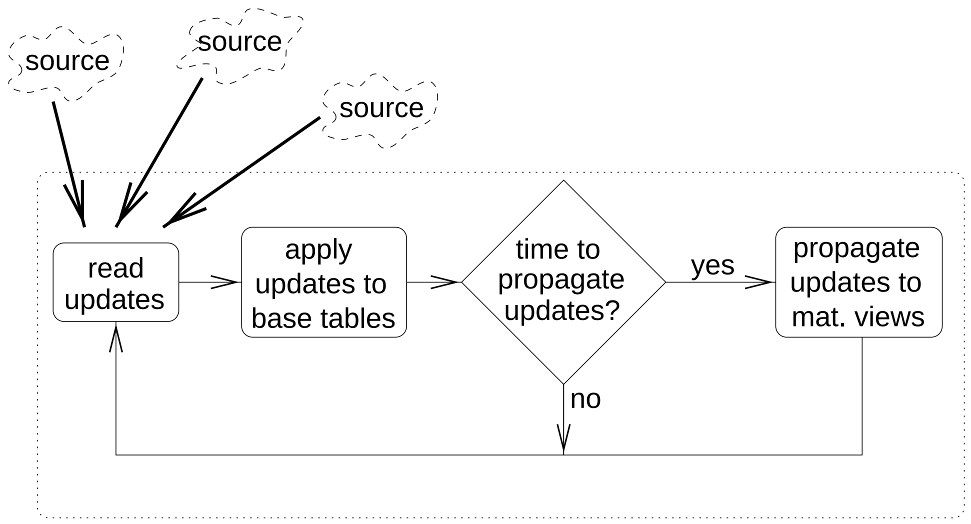

source
source
source
read updates
apply updates to base tables
time to propagate updates?
yes
no
propagate updates to mat. views

Figure 1: MAUVE

2.1 Using Timestamps

MAUVE supports multiple versions by using Time Travel ([Sto87]). Each row has two extra attributes, T_{min}, the *insertion timestamp*, and T_{max} the *deletion timestamp*. Timestamps in our system are rather coarse, and assigned at the warehouse by grouping many updates together into logical *chunks* of work ([RES93]).

To illustrate the use of timestamps, we follow the life-cycle of a tuple. First of all, suppose tuple r gets inserted into the source database. This insertion will propagate through the network and eventually reach the warehouse. At the warehouse, the insertion will receive a timestamp, say T_{ir}, grouping it along with other updates that share the same timestamp number. When the insertion is applied to the base table in the warehouse, the newly created tuple gets $T_{min} = T_{ir}$, and also T_{max} = *null*. At some point in time, tuple r might be deleted from the source database and the deletion will once again arrive at the warehouse receiving a timestamp of T_{dr}. However, this time, there will be no actual deletion of tuple r at the warehouse, as we will only update its T_{max} to T_{dr} instead. At some later point in the future, tuple r can finally be physically deleted from the warehouse.

In our model, updates can be either insertions or deletions (so updating an attribute of a tuple would correspond to a pair of update operations). We also assume that updates for each relation come from only one source and that we have in-order delivery of updates at the warehouse. If there are multiple sources, we assume each relation resides in one source. These assumptions guarantee that $T_{ir} <= T_{dr}$ and consecutively that $T_{min} <= T_{max}$ (or T_{max} = *null*) hold for all tuples.

2.2 Queries

T_{min} and T_{max} are used by our system to provide different versions of the warehouse to queries. Each version corresponds to a state of the warehouse that is completely consistent with the source databases. A special variable, global_timestamp, is used to indicate the current maximum version, which would correspond to the most up-to-date consistent state of the warehouse.

Each query, on startup, records the current global_timestamp into a private variable, local_timestamp. Throughout the execution of the query, only tuples with

$$T_{min} <= \textsf{local\_timestamp} \tag{1}$$

$$T_{max} > \textsf{local\_timestamp} \quad \text{or} \quad T_{max} = \textit{null} \tag{2}$$

are "visible" to the query. In other words, we only allow the query to access tuples that were created sometime in the past, and have not yet been marked as "deleted".

Implementing this versioning scheme for queries is simple, since it can be done by query modification ([Sto75]) where we rewrite the queries to include Eq. 1 and Eq. 2 as part of their predicate (i.e. the `where` clause in SQL).

2.3 Incremental View Updates

Let $V_{R,S} = R \bowtie S$ be our materialized view. R' and S' are the updated base tables R and S. I_R, I_S are the sets of insertions to R and S respectively, and D_R, D_S are the sets of deletions.

MAUVE uses the formulas from [Rou91] to compute incremental updates to the view:

$$\begin{aligned} V'_{R,S} = R' \bowtie S' &= (R \cup I_R - D_R) \bowtie (S \cup I_S - D_S) \\ &= (R \bowtie S) - (R \bowtie D_S) - (D_R \bowtie S) - (D_R \bowtie D_S) \\ &\quad \cup (R \bowtie I_S) - (D_R \bowtie I_S) \\ &\quad \cup (I_R \bowtie S) \,\cup\, (I_R \bowtie I_S) - (I_R \bowtie D_S) \\ &= [R \bowtie S - \{D_R, *\} - \{*, D_S\}] \,\cup\, (R^- \bowtie I_S) \,\cup\, (I_R \bowtie S') \end{aligned} \qquad (3)$$

where $\{D_R, *\}$ are the pairs (tid_R, tid_S) of $V_{R,S}$ for which $tid_R \in D_R$,
$\{*, D_S\}$ are the pairs (tid_R, tid_S) of $V_{R,S}$ for which $tid_S \in D_S$, and
$R^- = R - D_R$.

One interesting observation for Eq. 3 is that there is a “hidden” assumption that $I_R \cap D_R = \emptyset$ and $I_S \cap D_S = \emptyset$. If for example $I_R \cap D_R = r$, then tuple r will appear in $V'_{R,S}$, because when D_R is applied, r has not been inserted yet and thus won’t be deleted. One way around this is to have some sort of preprocessing to eliminate such “trivial” updates ([Sta89]). Another way is to “clean up” the two sets of insertions I_R and I_S on the fly, by checking to see if any of their members also belong to D_R or D_S respectively.

Each update phase in *MAUVE* gets assigned a unique timestamp, update_timestamp. This is used as the value for T_{min} for all tuples inserted to the warehouse by the update process (be it at a base table or at a view), and also as the T_{max} for all deleted tuples (again both at base tables or views). After the update process completes, global_timestamp is incremented to reflect a more up-to-date consistent version of the warehouse. Using timestamp “arithmetic”, we derive the SQL statements for the sets in Eq 3:

```
$D_R$  =  select  * from  $R$
           where  $T_{max}$ = update_timestamp
```

```
$I_R$  =  select  * from  $R$
           where  $T_{min}$ = update_timestamp
             and  ($T_{max}$ > update_timestamp  or  $T_{max}$ = null )
```

```
$R^-$  =  select  * from  $R$
           where  $T_{min}$ < update_timestamp
             and  ($T_{max}$ > update_timestamp  or  $T_{max}$ = null )
```

```
$S'$   =  select  * from  $S$
           where  $T_{min}$ <= update_timestamp
             and  ($T_{max}$ > update_timestamp  or  $T_{max}$ = null )
```

2.4 Discussion

Timestamps are generated at the warehouse, so their use imposes no overhead to the data sources whatsoever. Implementation overhead is also quite low. They create little interaction between readers and updaters (common access to global_timestamp), and also allow for concurrent, consistent access to the warehouse.

Timestamps are used to group update operations together. However, the granularity of this grouping is arbitrary[2] and can be defined on a per workload basis or even dynamically. Therefore, chunk size should be seen as a special *knob* in our system, a unique feature of *MAUVE*. On the one hand, batching a lot of operations together usually improves performance (except in the extreme case of long transactions where it might have adverse side effects on the rest of the system). On the other hand, the smaller each update phase is, the more up-to-date the warehouse will be (except of course for the degenerate case of very small update phases that saturate the system). In the next section we propose a way on how to "tune" chunk size to optimize view staleness.

3 View Staleness

We assume a warehouse that keeps complete replicas of the base tables, or at least a part of the base tables that hold all the *relevant updates* ([BALT86], [HZ96a]). In our environment, updates from the sources arrive at the warehouse and are being applied to the base tables. At some points, this process gets interrupted in order to propagate these updates to the view(s).

Even with the best online view update algorithm, the decision of when to update the views is not straightforward. On the one hand, if the updates are propagated to the views too often, the extra overhead will probably delay the future updates (both on base tables and views) significantly. On the other hand, if the views get updated too infrequently the result would be "stale" view data. The latter is the usual practise nowadays, with the updates being committed to the warehouse once a day. Finding a solution in-between would be desirable in order to keep the data in the warehouse relatively "fresh" without too much overhead.

The same problem applies to views that are *self-maintainable* ([QGMW96], [Huy97]). In that case, although there are no base tables, changes in base tables are mapped to changes in the view and the auxiliary views. This implies that we still need to strike a balance between propagating the updates to the "target" views frequently versus tolerating "stale" view data.

As illustrated in figure 2, frequent view updates cause base table updates to get "pushed back", but these changes get propagated to the views significantly faster. Case (a), updates the view once at the end. We can clearly see however that the first base table updates (e.g. tuple r_1) will wait a long time till they get propagated to the view. In case (b), the view gets updated twice. Although some updates will be propagated sooner to the views (e.g. r_1), some will be delayed (e.g tuples r_2, r_3). Finally, the same tradeoff

[2]Actually, if we want to maintain complete consistency with the data sources, we have to limit "splitting" the update stream at transaction boundaries only. This shouldn't place too much of a constraint.

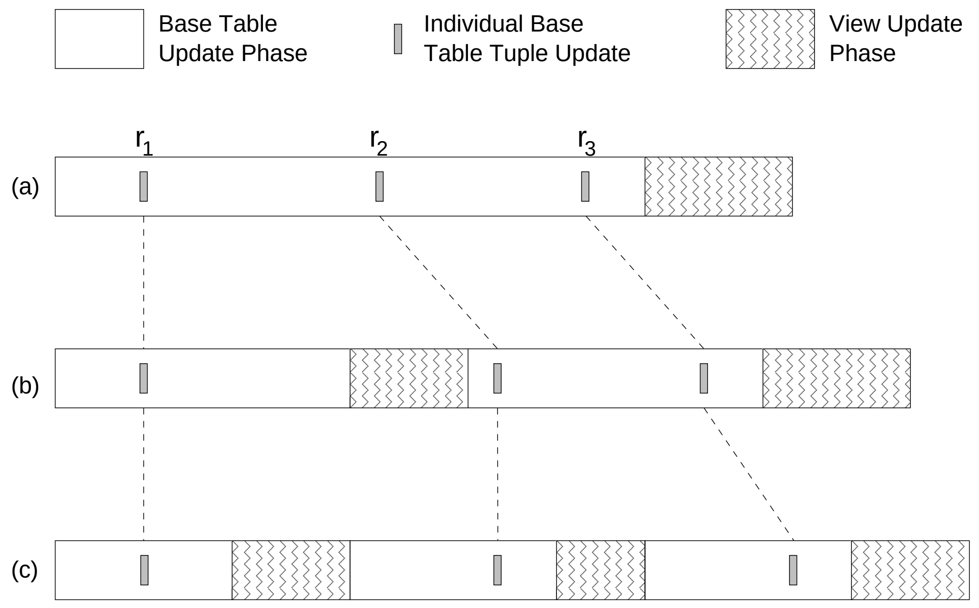


Figure 2: Executing the updates from logs with different view update frequencies

characterizes case (c), were we have three view update phases.

In the following sections, we define *view staleness*, derive a simple mathematical formula for its value under varying view update frequencies, and, finally, give some rough analysis of its behavior.

3.1 Definition

We are focusing on the freshness of views. For each base table update operation we measure the elapsed time between T_b, the moment that the update is applied to the base table, and T_v, the moment it is propagated to the materialized views. We can define view staleness as the average of this elapsed time for all updates. However, this definition does not capture the fact that frequent view updates delay the entire warehouse update process and must be "penalized" somehow. Let T_{eb} be the *earliest time* when the update would have been applied to the base table, if no view updates were interleaved.

We define the actual *view staleness* for each base table update:

$$\mathit{VS} = T_v - T_{eb} \tag{4}$$

By rewriting Eq. 4, as $\mathit{VS} = T_v - T_{eb} + T_b - T_b = \underbrace{T_b - T_{eb}}_{S_1} + \underbrace{T_v - T_b}_{S_2}$ we obtain two terms:

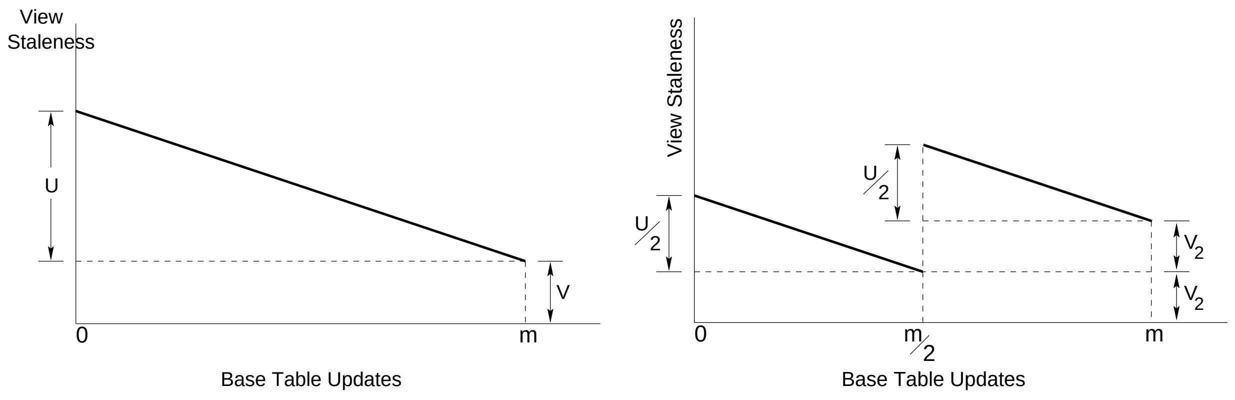
- $S_1 = T_b - T_{eb}$, is the extra "delay" introduced to base table updates from their interleaving with view updates. S_1 is expected to rise as the view update frequency increases.

- $S_2 = T_v - T_b$, is the view staleness as "seen" from the update process. S_2 is expected to decrease as the view update frequency increases, since the window of base table updates that are being propagated to the view gets smaller.

The advantages of this definition of view staleness are twofold. First of all, it provides a fair comparison to schemes with different view update frequencies, as it takes into consideration both the delay caused by each view update phase and the speed by which the updates get propagated to the views. Secondly, by making the time of the application of the updates to the base tables as the starting point of our measurements, view staleness becomes insensitive to the speed by which these base table updates are processed by the system (which would be the case if the arrival times were used). It also makes this definition insensitive to the update rates of the sources.

3.2 Calculation

Having defined view staleness, we proceed to calculate an analytic formula for it, under varying view update frequencies. If n is the number of view update phases, let VS_n be the average view staleness over all base table updates. Also, let U be the total time it takes to process the base table updates. We expect U to be roughly the same even with different view update frequencies. Finally, let V_n be the average view update time.



a) One view update phase b) Two view update phases

Figure 3: View staleness calculation

When a view is updated only once, all base table updates are processed without interruption and hence S_1 is always 0 for all of them (see Figure 3a). S_2 is high for the first update and decreases monotonically for the rest. The minimum for S_2 is achieved for the last base tuple update and equals to the time to complete the view update, V_1. The maximum for S_2 would be reached for the first base tuple update and it would be the time to process all the base table updates, U, plus the time to complete the view update V_1. Summing up, we have $\min(VS_1) = S_1 + \min(S_2) = 0 + V_1$. Also, $\max(VS_1) = S_1 + \max(S_2) = 0 + (U + V_1)$. The average view staleness becomes:

$$VS_1 = \frac{\min(VS_1) + \max(VS_1)}{2} = \frac{V_1 + (U + V_1)}{2} = \frac{U}{2} + V_1 \quad (5)$$

The case with two view update phases is similar, but now we have to do a similar analysis for each of the two "chunks" that the log is split into. During each segment, approximately half of the base table updates will be completed (in roughly $\frac{U}{2}$ time), and each of the view update phases will take on average V_2 time (see figure 3b). Since all the base table updates in the second segment will have to be "delayed" by V_2, the average view staleness for updating the view twice would be:

$$VS_2 = \frac{(\frac{U}{4} + V_2) + (\frac{U}{4} + 2 \times V_2)}{2} = \frac{U + 6 \times V_2}{4} \tag{6}$$

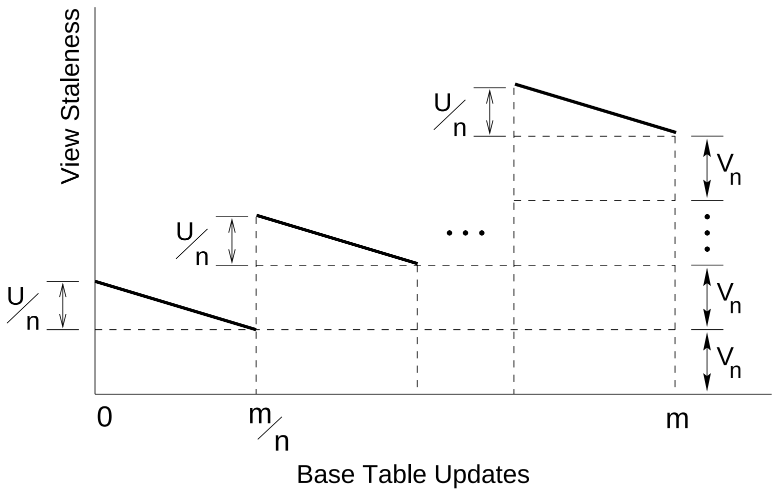


Figure 4: View Staleness with n view update phases

In the general case, with n view update phases (Figure 4), the average view staleness is roughly:

$$VS_n = \frac{1}{n}\left(n \times \frac{U}{2n} + V_n \times \sum_{i=1}^{n} i\right) = \frac{U + n \times (n+1) \times V_n}{2n} \tag{7}$$

where U is the total time needed for base tables updates and V_n is the average view update time per phase, and S_1, S_2 are:

$$S_1 = \frac{(n-1) \times V_n}{2} \text{ and } S_2 = \frac{U}{2n} + V_n \tag{8}$$

Eq. 8 captures the meaning of S_1 and S_2. S_1 increases as n gets bigger, and measures the "delay" caused by frequent view updates. S_2 steadily decreases as n increases, and accounts for the decrement of the time it takes for each update to be propagated to the views.

3.3 Analysis

Figure 5 plots view staleness, *VS*, together with S_1 and S_2, for some typical values of the total update time U and the average view update time V_n. In order to simplify calculations, we assumed the same value for V_n for all n, a rather pessimistic approach for the cases with high view update frequency, for which V_n is expected to be lower.

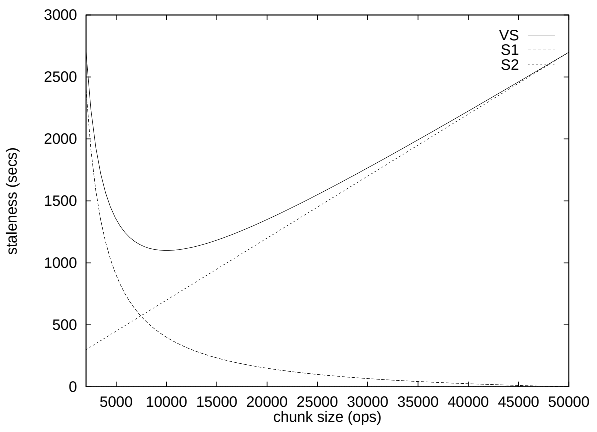


Figure 5: Sample plot of view staleness

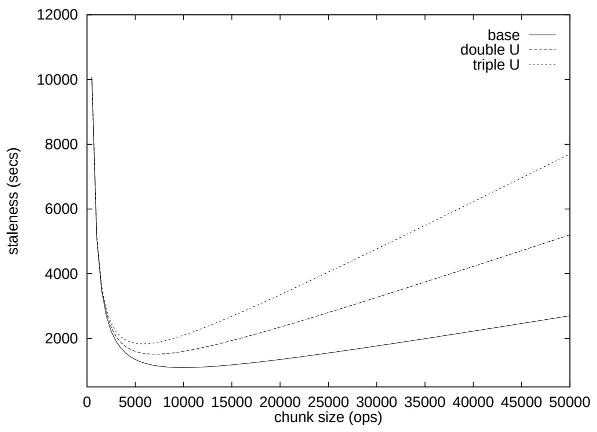
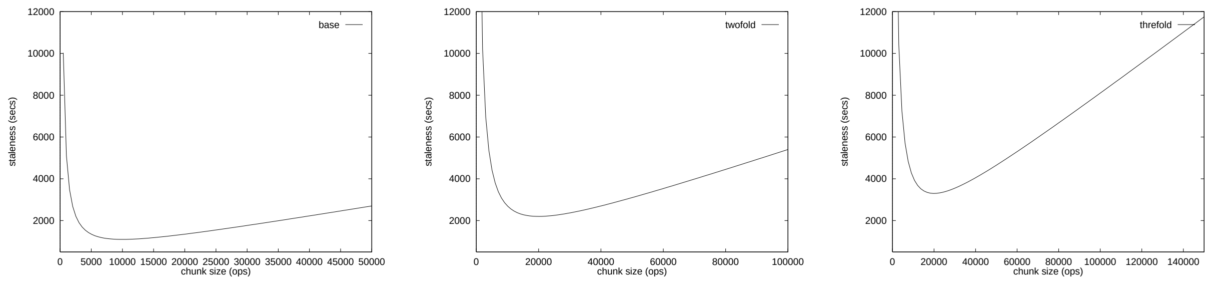


Figure 6: Increasing the update time

The x-axis is the *chunk size*, which is the number of base table updates each view update phase will have to propagate. Note that the *view update frequency* is the inverse of chunk size. For example, in Figure 5, a chunk size of 10,000 corresponds to a view update frequency of 5 ($= \frac{50{,}000}{10{,}000}$).

We can clearly see that S_1 is increasing exponentially with the chunk size getting smaller, and that S_2 is increasing linearly with the chunk size getting bigger. View staleness, the sum of S_1 and S_2, is a concave curve. It starts off with really high values (attributed to the very high values of S_1), but plunges until it reaches a minimum and then starts to increase again (picking up the high values of S_2). The minimum for *VS* is around the point where the two curves S_1 and S_2 meet.

Total update time, U is expected to have a big effect on the value of view staleness. Figure 6 plots the view staleness for three different cases, where the update time is set to be double and triple as that of the "base" case. Clearly, the bigger the update time, the more the reduction of view staleness if we update the views more frequently. Such a case might occur, when the update stream is "slow" (because the sources aren't sending their updates fast, or because of network delay) or has intervals between update arrivals with no activity.



Base Twofold scaling Threefold scaling

Figure 7: Scaling both U and V_n

Figure 7 has a plot of scaling up all the parameters (total update time, number of update operations, average view update time), where the second curve is a twofold scaleup of the "base" curve and the third curve a threefold scaleup. Comparing these two curves with the "base" curve, we see that the bigger the scale-up is, the "deeper" the curve is. A deeper curve would imply that the view staleness improves significantly after updating the views more frequently than just once. This is really good news, as we expect this scaleup to be the case when we have multiple views in our warehouse and want to aggregate the individual per view measurements.

The various view staleness plots indicate that:

- In all cases there exists an optimal (minimum) value for view staleness. In most cases this optimal value is less than what the view staleness is when the view is update only once.
- In all cases there is a clear threshold, after which view staleness will soar. This means that increasing the view update frequency beyond that point will have adverse effects.

4 Experiments

We developed a synthetic workload generator, *Genesis*, in order to create update streams with varying characteristics. The base table that we used in our experiments have the same tuple-size as the Wisconsin benchmark [Dew93] (plus the extra timestamp attributes), but we only provided explicit values for the join attribute. *Genesis* uses techniques from [GSE$^+$94] and [PTVF92] in order to generate values which follow different distribution functions. It was used to create update streams with many different data patterns and also various "behaviors" over time (e.g. "slow" or "high speed" streams).

In our experiments, the join attribute value for the base relations had a uniform distribution. The min, max values were different among experiments and were picked so that the join selectivity would result in a view with roughly the same size as each of the two base relations. The materialized view in our system was stored in the form of a *View Index* ([Rou82], [Val87]), as another table.

MAUVE , our online view maintenance algorithm that uses timestamps, was implemented on top of the Informix Universal Server version 9.12 ([inf97]). It run as a separate client on the same machine where the server was running. The database was stored as a raw partition (to avoid any "outside" buffering from the OS). For all our experiments we used a SUN UltraSparc 1 model 170, with 64 MB of main memory, running Solaris 2.5. Since this was a networked machine, we tolerated some minor fluctuations in our results and thus we had to average our measurements over multiple runs.

For every experiment, a complete database (base tables & materialized views) had already been installed. *MAUVE* was used to apply the base table updates read from a log[3], propagating the updates to the view, every time we had completed "chunk size" number of base table tuple updates. For each tuple update we measure view staleness and average it over all tuples at the end.

[3]The update log also had timing information together with the update data, in order to mimic real update streams.

For all the experiments we plot the measured average view staleness over all the updates. We also plot the theoretic average for *VS* (given by Eq. 7) using some representative U and V_n numbers (usually the averages of all the runs in each experiment).

4.1 Update size

We expect the total number of update operations, the *update size*, to influence view staleness significantly. The bigger the update size, the longer the total update time (the total time it takes to process just the base table insertions and deletions), which as the analysis indicated, is really important for view staleness.

We used two 50 MB tables and as a view their join. We then run 4 sets of experiments with the following update sizes: *30,000* (= 4% of both tables), *40,000* (6%), *70,000* (10%) and *100,000* (15%). Figures 8 through 11 have the results.

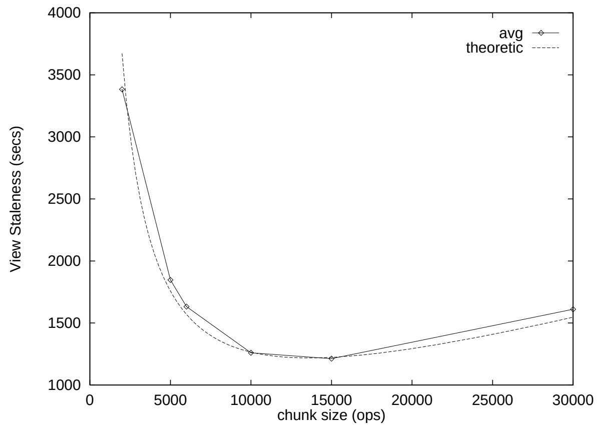


Figure 8: 30K updates

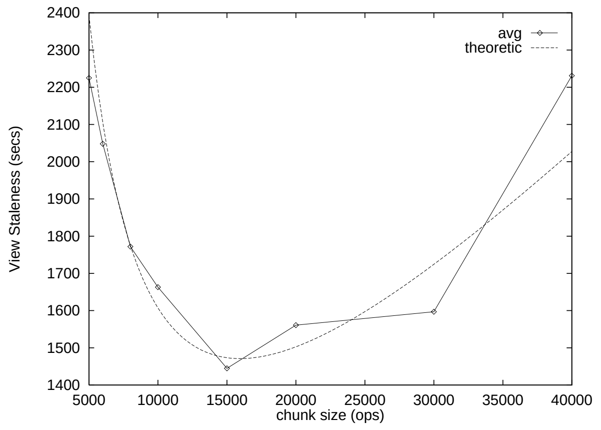


Figure 9: 40K updates

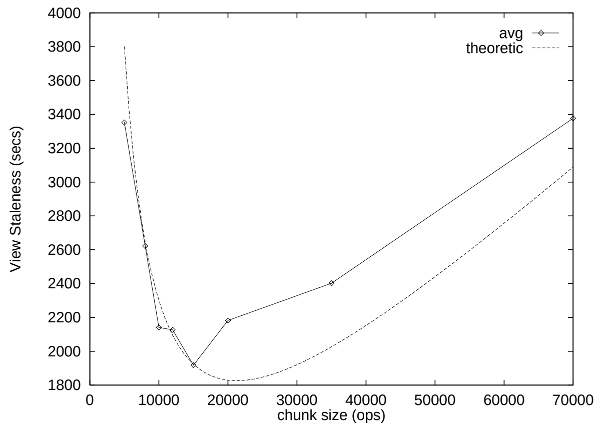


Figure 10: 70K updates

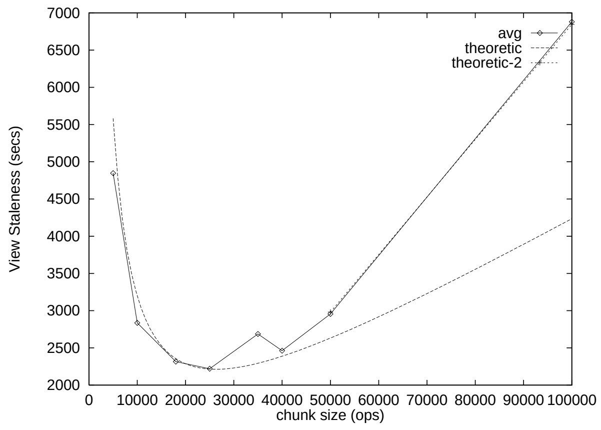


Figure 11: 100K updates

The first observation is that in all cases, propagating the updates to the view more than just once at the end, clearly *improves* the average view staleness. For Figure 8 the minimum view staleness is reached with chunk size = 15K, for Figure 9 with chunk size = 15K, for Figure 10 with chunk size = 15K and, for

Figure 10 with chunk size = 25K. These chunk sizes correspond to applying the incremental updates to the view respectively twice, three times, five times and four times instead of doing it once at the end.

As expected from our analysis, the bigger the update size, the deeper the view staleness curve is. In the first two sets of experiments with relatively small update sizes (Figures 8 and 9), *VS* is almost “flat” around the area of the minimum, and the difference from the case with only one view update phase is not very big. However, as the update size increases (Figures 10 and 11), so does the difference between the optimal view staleness and the view staleness for the case with only one view update.

In conclusion, we found that our experiments verified the intuition that the bigger update sizes are more prone to benefit by frequent view updates, as early updates become much more “stale” at the end, if not propagated in the mean time. In other words, it makes sense to break down “long” update jobs so that some of the updates get propagated to the views.

Comparing the experimental data with the predicted curve, we can see that there are cases where there is significant deviation. Since we used a single V_n in the calculation of the predicted view staleness (Eq. 7) for simplicity, this can be explained by the fact that there was considerable deviation in average view times among the different experiments. For example, in Figure 11, the average view time for the case with only one view update was about 3000, compared to a mere 570 on average for all the other cases. A solution to this problem is to use the actual V_n value for each point when calculating the predicted view staleness, instead of the overall average. We used this idea in Figure 11 for plotting a second curve, “theoretic-2”, which illustrates the fact that the predicted values coincide with the ones from the experiments.

4.2 Stream Continuity and Stream Update Rate

Following our experiments with varying update sizes, we experimented with workloads of different kinds of update streams. We identified two parameters that characterize the temporal behavior of incoming update streams:

- *Update rate* is the combined rate at which the sources supply updates to the warehouse. Streams are classified into those with *high update rate*, *medium update rate*, and *low update rate*, depending on whether the updates are coming at a speed higher than what the warehouse can process, just about, or less, respectively.

- *Continuity* refers to whether the stream exhibits great variations in the update rate or not. We classify update streams into, *steady*, i.e. those that have a nearly constant update rate, and *discontinuous*, i.e. those that exhibit greatly varying update rates (even with no update activity at some times).

The experiments of the previous section were for steady streams. In this section we concentrate on discontinuous streams.

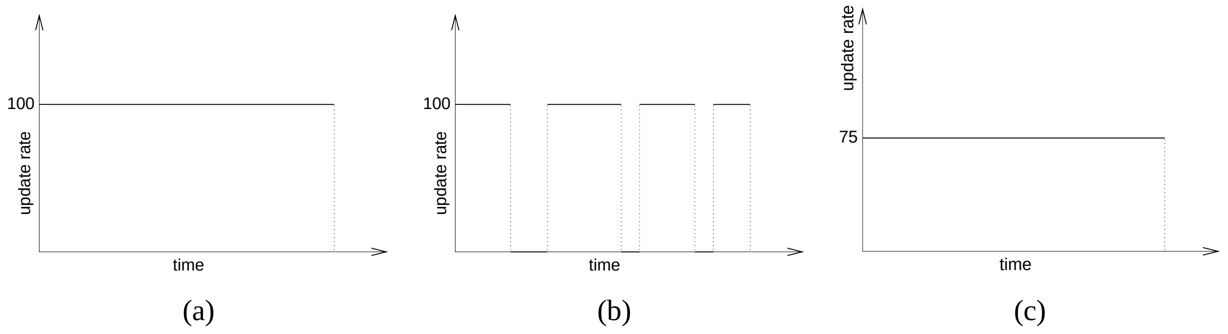


Figure 12: Comparison of three streams

Figure 12 has a plot of the update rate over time for three different streams. Cases (a) and (c) are steady streams, and (b) is an example of a discontinuous stream, with (c) having the same effective update rate as (b).

The motivation behind studying this kind of streams is that even with the sources constantly "willing" to supply the warehouse with updates, there are a lot of reasons for periods of no incoming update activity at the warehouse, because of network congestion or failures or other outside factors.

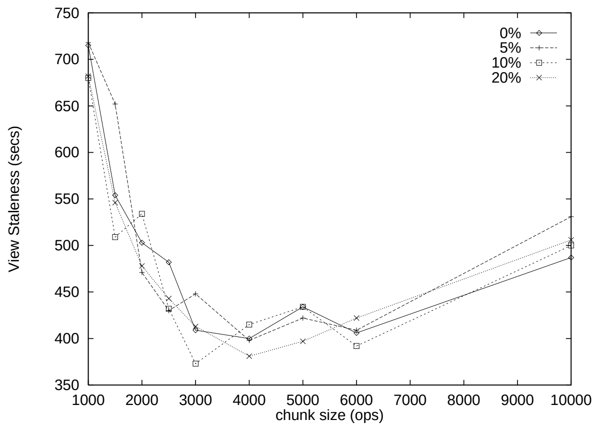


Figure 13: High update rate

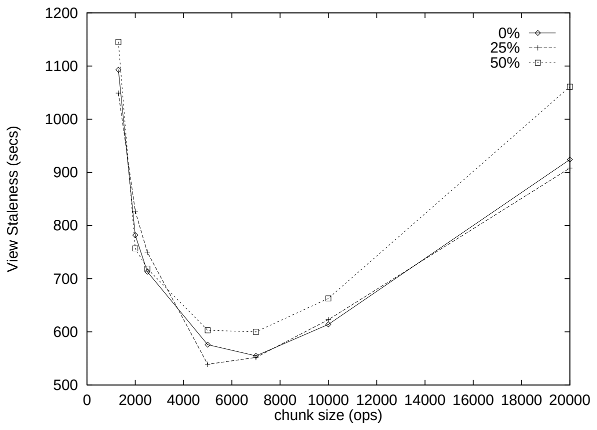


Figure 14: Medium update rate

The case of discontinuous streams with high update rates is plotted in Figure 13. We can see, that the percentage of periods[4] of no update activity versus regular activity on the warehouse has no major influence on view staleness. Although view staleness can be improved by updating the views more than once, all four view staleness curves seem to be roughly the same. The reason is that as the update rate is really high, it makes no big difference even if it has discontinuities. If for example it takes X minutes to process the updates, and these updates all arrive on the first minute, then it would make no difference if there were any periods of no activity in this minute or not. Even if it took two minutes for the updates to arrive (twice the

[4]The length of these periods was made to follow a negative exponential distribution in order to agree with typical inter-arrival rates.

previous time), it still wouldn't make much of a difference.

For the second set of experiments, we studied discontinuous streams with medium update rates. In Figure 14, the curves for view staleness are much more concave than the previous set of experiments, which means that these cases can benefit even more from higher view update frequencies.

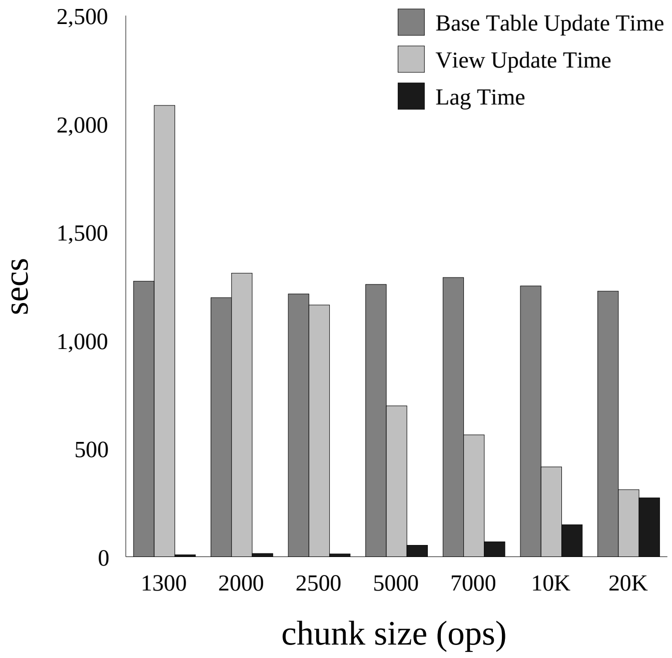


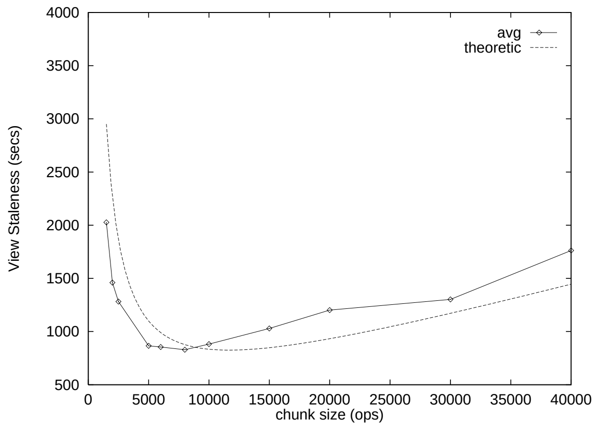
Figure 15: Lag time for a discontinuous stream

When we have discontinuous medium update rate streams, the warehouse begins to experience *lag time*, or in other words, idle time because of no update activity. In this case, it makes sense to interrupt the update stream in order to propagate the updates to the views. Figure 15 has measurements for total *lag time*, compared to the total base table update time and the total view update time, for the medium update rate stream with the most discontinuity. Clearly, as the view chunk size decreases (i.e. the views are updated more frequently), the *lag time* decreases as well, because that time is being used for updating the views.

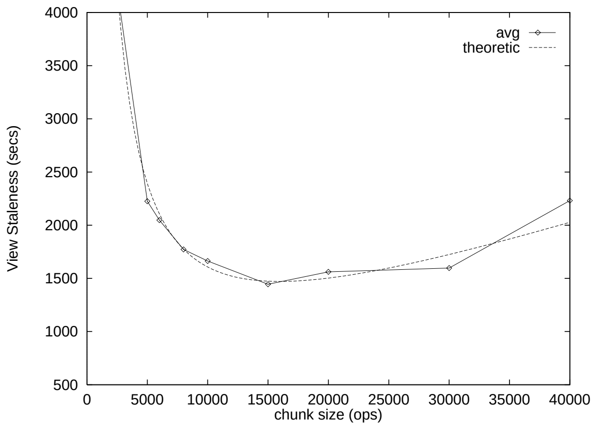
Finally, in other experiments with low update rate streams (not shown here because of space considerations), we were able to verify our intuition that as the update rate gets lower, the benefits of propagating the updates to the views more frequently increase. In other words, it makes sense to "interrupt" a light update job.

4.3 Size of base tables

With size being a key consideration in data warehousing environments, we conducted a scalability experiment to see how table size affects view staleness. We compared plots from the two different table sizes of the previous experiments, 30MB and 100MB, while keeping the update size constant.



30MB tables



100MB tables

Figure 16: Scaling the table size / 40K updates

Figure 16 is the case where the absolute number of updates was kept constant (to 40,000 updates in our experiment). Comparing the two runs for the different base table sizes (30MB vs 100MB) we see that the view staleness curves have roughly the same shape (with the second curve perhaps being slightly less concave than the first one). In other words, scaling the database size didn't seem to have any major impact on view staleness.

5 Related Work

[QW97] proposes the *2VNL* algorithm for online view maintenance where pre-update versions of updateable attributes are kept for readers to access while a maintenance transaction is active. Their approach has little overhead and can be implemented by query rewrite. *MAUVE* provides a natural extension to n versions, thus avoiding the session expiration problem when a reader overlaps with more than one maintenance transaction.

Algorithms for deferred view maintenance that minimize downtime were presented in [CGL$^+$96]. They avoid the "state bug" where direct application of pre-update algorithms in the post-update state results in incorrect view change calculation. The use of timestamps in *MAUVE* allows us to access the pre-update phase of the tables, circumventing the state bug. Also, we improve on the "deferredness" of view updates, by introducing view staleness as a key metric to optimize in order to achieve a good view freshness/performance tradeoff.

Data staleness is introduced in [AGMK95], where they study various scheduling policies among update and query transactions in a soft real-time database system. Nevertheless, all updates in their system are applied directly to the database tables without the need to propagate these updates to any derived data, as is the case with *view staleness* and materialized views in our work.

The notion of view freshness also appears in [HZ96a] and [HZ96b] during the presentation of the

Squirrel integration mediators. However, their study is different from ours, as they study the tradeoffs between different view materialization approaches (fully materialized, partially materialized and fully virtual) under eager update processing, whereas we focus our study on the tradeoffs between different view update frequencies.

The work in [SLSV95] contains algorithms for splitting up long transactions. Although, *MAUVE* in effect splits up a very long transaction (the update stream), the difference in our case is that we still enforce a serial ordering on those “chunks” of work to guarantee in-order application of the updates to the warehouse.

6 Conclusions & Future Work

We have proposed *MAUVE* , a new online view update algorithm. *MAUVE* guarantees full consistency while allowing concurrent read-only access to the warehouse during the refresh operation. It has little implementation overhead as it can be implemented with query rewrite, and also imposes little storage overhead. *MAUVE* allows for the base table updates to be propagated to the views at arbitrary points (the chunk size), which can be used as a knob in the system to optimize view staleness.

We have given a definition for view staleness that provides a fair comparison to schemes with different view update frequencies. We have also derived an analytical formula for it, which was verified by experimental results. Our experiments showed that view staleness can be greatly improved if updates to the views are propagated more frequently than once a day.

Future Work We based this work on the assumption that all views in the warehouse are equally “important”, and thus it always makes sense to try “stealing” some cycles from the base table updates in order to refresh the views. However, we want to experiment with cases where there is a distinction between “hot” and “cold” data in the warehouse and see if allowing different view update frequencies per view can give a better solution for these cases.

Acknowledgments We would like to thank Kostas Stathatos, Yannis Kotidis and Damianos Karakos for many useful discussions, comments and suggestions.

References

[AASY97] Divyakant Agrawal, Amr El Abbadi, Ambuj K. Singh, and Tolga Yurek. “Efficient View Maintenance at Data Warehouses”. In *Proc. of the ACM SIGMOD Conference*, pages 417–427, Tucson, Arizona, May 1997.

[AGMK95] Brad Adelberg, Hector Garcia-Molina, and Ben Kao. “Applying Update Streams in a Soft Real-Time Database System”. In *Proc. of the ACM SIGMOD Conference*, pages 245–256, San Jose, California, May 1995.

[BALT86] José A. Blakeley, Per Ȧke Larson, and Frank Wm. Tompa. “Efficiently Updating Materialized Views”. In Carlo Zaniolo, editor, *Proc. of the ACM SIGMOD Conference*, pages 61–71, Washington, D.C., May 1986.

[CD97] Surajit Chaudhuri and Umeshwar Dayal. “An Overview of Data Warehousing and OLAP Technology”. *SIGMOD Record*, 26(1):65–74, March 1997.

[CGL$^+$96] Latha S. Colby, Timothy Griffin, Leonid Libkin, Inderpal Singh Mumick, and Howard Trickey. “Algorithms for Deferred View Maintenance”. In *Proc. of the ACM SIGMOD Conference*, pages 469–480, Montreal, Canada, June 1996.

[CW91] Stefano Ceri and Jennifer Widom. “Deriving Production Rules for Incremental View Maintenance”. In *Proc. of the Very Large Data Bases (VLDB) Conference*, pages 577–589, Barcelona, Catalonia, Spain, September 1991.

[Dew93] David J. Dewitt. “The Wisconsin Benchmark: Past, Present, and Future”. In Jim Gray, editor, *“The Benchmark Handbook for Database and Transaction Processing Systems”*, chapter 4. Morgan Kaufmann, 2nd edition, 1993.

[GL95] Timothy Griffin and Leonid Libkin. “Incremental Maintenance of Views with Duplicates”. In *Proc. of the ACM SIGMOD Conference*, pages 328–339, San Jose, California, May 1995.

[GLT97] Timothy Griffin, Leonid Libkin, and Howard Trickey. “An Improved Algorithm for the Incremental Recomputation of Active Relational Expressions”. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):508–511, May 1997.

[GM95] Ashish Gupta and Inderpal Singh Mumick. “Maintenance of Materialized Views: Problems, Techniques, and Applications”. *Data Engineering Bulletin*, 18(2):3–18, June 1995.

[GMS93] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. “Maintaining Views Incrementally”. In *Proc. of the ACM SIGMOD Conference*, pages 157–166, Washington, D.C, May 1993.

[GSE$^+$94] Jim Gray, Prakash Sundaresan, Susanne Englert, Kenneth Baclawski, and Peter J. Weinberger. "Quickly Generating Billion-Record Synthetic Databases". In *Proc. of the ACM SIGMOD Conference*, pages 243–252, Minneapolis, Minnesota, May 1994.

[Huy97] Nam Huyn. "Multiple-View Self-Maintenance in Data Warehousing Environments". In *Proc. of the Very Large Data Bases (VLDB) Conference*, pages 26–35, Athens, Greece, August 1997.

[HZ96a] Richard Hull and Gang Zhou. "A Framework for Supporting Data Integration Using the Materialized and Virtual Approaches". In *Proc. of the ACM SIGMOD Conference*, pages 481–492, Montreal, Canada, June 1996.

[HZ96b] Richard Hull and Gang Zhou. "Towards the Study of Performance Trade-offs Between Materialized and Virtual Integrated Views". In *Proc. of Workshop on Materialized Views: Techniques and Applications (VIEW 1996)*, pages 91–102, Montreal, Canada, June 1996.

[inf97] *"INFORMIX Universal Server DataBlade API"*, June 1997.

[PTVF92] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *"Numerical Recipes in C"*. Cambridge University Press, 2nd edition, 1992.

[QGMW96] Dallan Quass, Ashish Gupta, Inderpal Singh Mumick, and Jennifer Widom. "Making Views Self-Maintainable for Data Warehousing". In *Proc. of the Conference on Parallel and Distributed Information Systems (PDIS)*, Miami Beach, Florida, 1996.

[QW91] Xiaolei Qian and Gio Wiederhold. "Incremental Recomputation of Active Relational Expressions". *IEEE Transactions on Knowledge and Data Engineering*, 3(3):337–341, September 1991.

[QW97] Dallan Quass and Jennifer Widom. "On-Line Warehouse View Maintenance". In *Proc. of the ACM SIGMOD Conference*, pages 393–404, Tucson, Arizona, May 1997.

[RCK$^+$95] Nick Roussopoulos, Chungmin Melvin Chen, Stephen Kelley, Alex Delis, and Yannis Papakonstantinou. "The ADMS Project: Views R Us". *Data Engineering Bulletin*, 18(2):19–28, June 1995.

[RES93] Nick Roussopoulos, Nikos Economou, and Antony Stamenas. "ADMS: A Testbed for Incremental Access Methods". *IEEE Transactions on Knowledge and Data Engineering*, 5(5):762–774, October 1993.

[RK86] Nick Roussopoulos and Hyunchul Kang. "Preliminary Design of ADMS $\pm$: A Workstation-Mainframe Integrated Architecture for Database Management Systems". In *Proc. of the Very Large Data Bases (VLDB) Conference*, pages 355–364, Kyoto, Japan, August 1986.

[Rou82] Nick Roussopoulos. “View Indexing in Relational Databases”. *ACM Transactions on Database Systems*, 7(2):258–290, June 1982.

[Rou91] Nick Roussopoulos. “An Incremental Access Method for ViewCache: Concept, Algorithms, and Cost Analysis”. *ACM Transactions on Database Systems*, 16(3):535–563, September 1991.

[Rou98] Nick Roussopoulos. “Materialized Views and Data Warehouses”. *SIGMOD Record*, 27(1), March 1998.

[SLSV95] Dennis Shasha, Franois Llirbat, Eric Simon, and Patrick Valduriez. “Transaction Chopping: Algorithms and Performance Studies”. *ACM Transactions on Database Systems*, 20(3):325–363, September 1995.

[Sta89] Antonios G. Stamenas. “High Performance Incremental Relational Databases”. Technical report, UMIACS-TR-89-49, CS-TR-2245, Department of Computer Science, University of Maryland, College Park, MD 20742, May 1989.

[Sto75] Michael Stonebraker. “Implementation of Integrity Constraints and Views by Query Modification”. In *Proc. of the ACM SIGMOD Conference*, pages 65–78, San Jose, California, May 1975.

[Sto87] Michael Stonebraker. “The Design of the POSTGRES Storage System.”. In *Proc. of the Very Large Data Bases (VLDB) Conference*, pages 289–300, Brighton, England, September 1987.

[Val87] Patrick Valduriez. “Join Indices”. *ACM Transactions on Database Systems*, 12(2):218–246, June 1987.

[Vis98] Dimitra Vista. “Incremental View Maintenance as an Optimization Problem”. In *Proc. of the International Conference on Extending Database Technology (EDBT)*, Valencia, Spain, March 1998. (to appear).

[ZGMHW95] Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, and Jennifer Widom. “View Maintenance in a Warehousing Environment”. In Michael J. Carey and Donovan A. Schneider, editors, *Proc. of the ACM SIGMOD Conference*, pages 316–327, San Jose, California, May 1995.

[ZGMW96] Yue Zhuge, Hector Garcia-Molina, and Janet L. Wiener. “The Strobe Algorithms for Multi-Source Warehouse Consistency”. In *Proc. of the Conference on Parallel and Distributed Information Systems (PDIS)*, Miami Beach, Florida, December 1996.

A Table of Symbols

Symbol	Meaning
R, S	base tables
$V_{R,S}$	materialized view
R', S'	updated versions of R, S
$V'_{R,S}$	updated version of $V_{R,S}$
I_R, I_S	set of insertions on R, S respectively
D_R, D_S	set of deletions on R, S respectively
T_a	base table update arrival time
T_b	actual time when update is applied to base table
T_{eb}	earliest possible T_b, i.e. time when update would have been applied to base table, had there been no view updates
T_v	time when update is propagated to view
VS	view staleness (*VS* $= S_1 + S_2$)
VS$_n$	view staleness when view update frequency = n
S_1	base table updates delay
S_2	presumed view staleness
U	total time to update base tables
V_n	average view update time

Table 1: Table of symbols