WPI-CS-TR-03-29 Mar 2003

A Framework for Optimizing View Maintenance Plans over Distributed Data Sources

by

Bin Liu
Elke A. Rundensteiner and David Finkel

Computer Science Technical Report Series

WORCESTER POLYTECHNIC INSTITUTE

Computer Science Department
100 Institute Road, Worcester, Massachusetts 01609-2280

A Framework for Optimizing View Maintenance Plans over Distributed Data Sources *

Bin Liu, Elke A. Rundensteiner, and David Finkel

Department of Computer Science
Worcester Polytechnic Institute
Worcester, MA 01609–2280
{binliu|rundenst|dfinkel}@cs.wpi.edu

Abstract

Materialized views defined over distributed data sources are a well recognized technology for data integration, e-business, and data warehousing. Many algorithms have been proposed to date for incrementally maintaining materialized views, typically processing one update at a time. In situations when a real-time refresh of the view extent is not critical, changes to the sources are combined and maintained periodically such as once a day to improve the maintenance performance and to reduce the conflicts with user's read sessions upon the view extent.

In this work, we explore the key factors that affect the performance of view maintenance, in particular the number of maintenance queries and their complexity. We present four alternative strategies. First, we describe an algorithm for batching all updates from the same data source. This reduces the total number of maintenance queries to O(n^2) where n is the number of data sources that the view is defined upon, regardless how many source updates are being maintained. Second we enhance this batching strategy by sharing common subexpressions in the different maintenance processes. This further reduces the number of maintenance queries. Third, we propose two grouping strategies, namely, maximal grouping and conditional grouping, which both reduce the number of maintenance queries to O(n). The reduction in the number of maintenance queries comes as a trade-off in terms of an increase in the complexity of these queries. A cost model to analyze and compare these four strategies is provided. These maintenance strategies have been implemented in our TxnWrap materialized view maintenance system. Experimental studies illustrate the trade-offs between the different design choices for realizing maintenance strategies. Our experiments reveal an additional dimension of this design space, namely the impact of the cooperation of the remote sources in the maintenance process on the performance of such maintenance strategies.

*This work was supported in part by several grants from NSF, namely, the NSF NYI grant #IRI 97–96264, the NSF CISE Instrumentation grant #IRIS 97–29878, and the NSF grant #IIS 9988776.

1 Introduction

1.1 Materialized Views and Their Maintenance

Materialized views [6, 1] built by gathering data from possibly distributed data sources and integrating it into one repository customized to users' needs are a well recognized technology for data integration, e-business, and data warehousing. One important task of a view manager is to maintain materialized views upon source changes, since frequent updates are common for many applications. Unless the underlying data sources are small, it is usually desirable to maintain view extents incrementally [2, 5]. That is, instead of recomputing the view extent from scratch, the delta of the view extent is computed and committed to refresh the view extent. In this process, the view manager needs to send *maintenance queries* [19] to underlying data sources to determine the changes of the view extent that related to the updates.

Many algorithms have been proposed in the literature [19, 1, 20, 3, 18, 16] that incrementally maintain the materialized view extent, typically processing one update at a time. In situations when a real-time refresh of the view content is not critical, changes to the source can be buffered and propagated periodically to update the view extent. Two benefits can be reaped from such batching. One, we may gain a better overall maintenance performance. Two, fewer conflicts with users' read sessions upon the view extent may arise. However, most existing algorithms [19, 1, 3, 18] were designed to handle the maintenance for a single source update one at a time. They were not optimized for maintaining multiple source updates together in one process. In this work, we instead focus on how to incrementally maintain materialized views more efficiently for a given set of source updates. First, we review a batch algorithm [10, 8] using *delta changes* (one *delta change* describes changes made to a data source in a certain time period). Then, exploring the trade-off between the number of maintenance queries and their complexity, we propose a series of algorithms to optimize the overall performance of maintaining a given set of source updates.

1.2 State-of-the-Art in Materialized View Maintenance

The example below is used to illustrate state-of-the-art incremental view maintenance algorithms.

Example 1 *Assume we have three data sources with one relation each, as shown in Figure 1. A view Asia-Customer is defined by the SQL depicted in Query 1.*

Sequential Maintenance. The SWEEP algorithm introduced in [1] maintains a view extent incrementally for *one* source update at a time in a distributed environment. We illustrate this algorithm on Example 1. Assume one data

R_1: Customer(Name, Age, Address, Phone)
R_2: Tour(TourID, TourName, Type, NoDays)
R_3: FlightRes(Name, FlightNo, Source, Dest)

Figure 1: Description of Data Sources.

$$\begin{array}{ll} \text{CREATE VIEW} & Asia-Customer \text{ AS} \\ \text{SELECT} & C.Name,\ C.Age,\ F.Dest \\ & F.FlightNo,\ T.TourID \\ \text{FROM} & Customer\ C,\ FlightRes\ F,\ Tour\ T \\ \text{WHERE} & C.Name\ =\ F.Name \\ & AND\ F.Dest\ =\ `Asia' \\ & AND\ F.Name\ =\ T.TourName \end{array} \quad (1)$$

update "$DU^1_{R_1}$ = *insert into Customer values ('Ken', 27, 'MA', 5857)*" happened at R_1. The subscript "R_1" in $DU^1_{R_1}$ denotes the data source where this update happened, while superscript "1" represents the ith ($i \geq 1$) update in this source (the sequential number in order of occurance). In order to determine the delta effect on the view extent, this requires us to send maintenance queries [19] to sources R_2 and R_3. Queries 2 and 3 represent these two maintenance queries due to update $DU^1_{R_1}$, assuming the result of Query 2 is ('Ken', 28, 69).

$$\begin{array}{ll} \text{SELECT} & 'Ken'\ as\ Name,\ 28\ as\ Age,\ T.TourID \\ \text{FROM} & Tour\ T \\ \text{WHERE} & T.TourName\ =\ 'Ken' \end{array} \quad (2)$$

$$\begin{array}{ll} \text{SELECT} & 'Ken'\ as\ Name,\ 28\ as\ Age,\ F.Dest \\ & F.FlightNo,\ 69\ as\ TourID \\ \text{FROM} & FlightRes\ F \\ \text{WHERE} & F.Name\ =\ 'Ken' \\ & AND\ F.Dest\ =\ 'Asia' \end{array} \quad (3)$$

Thus, to maintain one source update using SWEEP, we may have to traverse all the underlying data sources besides the one where the source update originated from to compute the incremental effect on the view extent. If multiple source updates need to be maintained at the same time, we would repeat this process for each update until all updates have been processed [1].

Batch View Maintenance. In our earlier work [10], we have proposed a batch maintenance strategy which maintains the view extent using source-specific *delta changes*. For example, assume within a certain period, five updates happened on the data sources defined in Example 1 and are reported for maintenance in the order listed in Figure 2. Instead of maintaining these five source updates individually as described above, we first construct *delta changes* specific for each source. Thus, ΔR_1 = { +('Ben', 28, 'MA', 6136), -('Ken', 27, 'MA', 5857) } [2], ΔR_2 = { +(63, 'Ben', 'L', 10) }, and ΔR_3 = { +('Ben', 168, 'MA', 'Aisa'), +('Tom', 169, 'CA', 'Aisa') }. Thereafter, we compute the incremental view extent for all these updates in three steps. Within each step, one source-specific delta traverses the underlying data sources once to compute the maintenance result.

Batch view maintenance is usually more efficient than sequential maintenance in terms of the total processing time for a given set of source updates [10, 8]. Sequential maintenance involves many maintenance queries (depending on both the number of source updates and data sources) to be sent to the data sources with each maintenance query reflecting a single source update. In batch maintenance, however, we have a small number of maintenance

[1] Concurrent source updates could happen during the maintenance process. Thus an additional concurrency control strategy is necessary to keep the view extent consistent [20], as further discussed in Section 4.3.

[2] For simplicity, we use '+' to represent an insert operation and '-' to denote a delete operation.

$DU^1_{R_1}$:	Insert('Ben', 28, 'MA', 6136) into Customer
$DU^2_{R_3}$:	Insert('Ben', 168, 'MA', 'Aisa') into FlightRes
$DU^3_{R_1}$:	Delete('Ken', 27, 'MA', 5857) from Customer
$DU^4_{R_2}$:	Insert(63, 'Ben', 'L', 10) into Tour
$DU^5_{R_3}$:	Insert('Tom', 169, 'CA', 'Asia') into FlightRes

Figure 2: Sequence of Source Updates.

queries (depending only on the number of data sources) with each maintenance query corresponding to a set of source updates. This now opens the opportunity to combine several updates and construct a combined maintenance query that may outperform handling each individual update one by one. The study of this trade-off between the number for maintenance queries and their complexity in view maintenance performance is the main issue that this work addresses.

1.3 Contributions of this Work

Our key contributions in this work are:

1. We study state-of-the-art materialized view maintenance algorithms and illustrate the trade-offs between the number of maintenance queries and their complexity.

2. We propose one strategy to exploit the common subexpressions typically found in a maintenance process. This improves maintenance performance by reducing the number of maintenance queries.

3. We propose two grouping strategies, namely, *maximal grouping* and *conditional grouping*, which both reduce the total number of maintenance queries to O(n) where n is the number of data sources that the materialized view is defined upon, regardless how many source updates need to be maintained.

4. We provide a cost model to analyze and identify the trade-offs that affect the maintenance performance for a given set of source updates.

5. We have implemented these maintenance strategies in our TxnWrap view maintenance system. Our experimental study shows the trade-offs between the different design choices for realizing maintenance strategies. Our experiments reveal an additional dimension of this design space, namely the impact of the cooperation of remote sources on the maintenance performance.

The rest of the paper is organized as follows. Section 2 describes the trade-offs in the view maintenance processes. Sections 3 and 4 describe the proposed strategies. A cost model is provided in Section 5. Section 6 discusses the experimental results, while related work and conclusions are given in Sections 7 and 8 respectively.

2 View Maintenance Trade-offs

For simplicity and without loss of generality, we simplify the representation of view definitions and maintenance queries using $\bowtie$ [3]. Thus, the view defined in Example 1 can be represented by $R_1 \bowtie R_2 \bowtie R_3$ where R_i $(1 \leq i \leq 3)$ represents the extent of the corresponding data source. The incremental view effect (ΔV) due to a source update $DU^1_{R_1}$ described in Section 1.2 can be represented by the formula "$\Delta V = DU^1_{R_1} \bowtie R_2 \bowtie R_3$". While for the batch maintenance strategy described, we represent the process as follows: $\Delta V = (\Delta R_1 \bowtie R_2 \bowtie R_3) + (R'_1 \bowtie \Delta R_2 \bowtie R_3) + (R'_1 \bowtie R'_2 \bowtie \Delta R_3)$. Here ΔR_i represents the *delta change* of source R_i and $R'_i = R_i + \Delta R_i$ $(1 \leq i \leq 3)$. Note that concurrency control strategies (either compensation based [1, 19] or multiversion based [3]) are needed in case of additional source updates happening concurrently to the process of evaluating above formulae.

To see the difference between the sequential and batch maintenance approaches, we abstract these two processes as follows. Assume a materialized view V is defined upon n distributed data sources denoted by $R_1 \bowtie R_2 \ldots \bowtie R_n$. Given a certain period, k source updates have happened and are reported for maintenance, $DU^1_{R_1}$, $DU^2_{R_1}$, $DU^3_{R_2}$, $\ldots$, $DU^k_{R_n}$. The subscript R_i $(1 \leq i \leq n)$ represents in which source the update has happened. If we maintain these updates sequentially, we have k steps with each one submitting n-1 maintenance queries (join operations) to the corresponding data source (depicted by Query 4). For batch maintenance, we first prepare $\Delta R_i = \sum_{j \geq 1} DU^j_{R_i}$ $(1 \leq i \leq n)$. Then we use n steps with each step possibly having n-1 join operations to the underlying data sources (depicted by Query 5) [4].

$$\begin{aligned} &\Delta V = DU^1_{R_1} \bowtie R_2 \bowtie R_3 \ldots \bowtie R_n \\ &+DU^2_{R_1} \bowtie R_2 \bowtie R_3 \ldots \bowtie R_n \\ &+\ldots \\ &+DU^k_{R_n} \bowtie R_1 \bowtie R_2 \ldots \bowtie R_{n-1} \end{aligned} \tag{4}$$

$$\begin{aligned} &\Delta V = \Delta R_1 \bowtie R_2 \bowtie R_3 \ldots \bowtie R_n \\ &+R'_1 \bowtie \Delta R_2 \bowtie R_3 \ldots \bowtie R_n \\ &+\ldots \\ &+R'_1 \bowtie R'_2 \bowtie R'_3 \ldots \bowtie \Delta R_n \end{aligned} \tag{5}$$

Typically, the number of source updates k is much larger than the number of data sources n. Thus, this opens the opportunity to combine several updates and construct a combined maintenance query that may outperform handling each individual update one by one. A **maintenance plan**, generated by one of the maintenance strategies, specifies how to maintain a given set of source updates at an abstract level. We call each "line" in the maintenance plan a **maintenance step**. Such a step traverses all the data sources (except the one where the update(s) has originally happened) to compute the maintenance result. The term **maintenance step granularity** refers to the number of source updates being combined. Thus, we have k maintenance steps with each maintenance step granularity being 1 in a sequential maintenance plan. While in a batch maintenance plan, we have n maintenance steps with each

[3] Like most algorithms proposed in the literature [1, 20, 18, 16, 10, 8], we assume that the materialized views are SPJ views that integrate data by joins across multiple data sources.

[4] As we mentioned early, more compensation queries are needed in both approaches if we choose a compensation-based concurrency control strategy. Without loss of generality, we only focus on the maintenance queries here.

maintenance step granularity typically being larger than 1. We end up computing $k(n\text{-}1)$ maintenance queries ($\bowtie$) for sequential maintenance and $n(n\text{-}1)$ for batch maintenance. Batch maintenance reduces the number of maintenance queries to distributed data sources by combining multiple source updates. However, two questions remain. First, is it possible to further reduce the number of maintenance queries, say to less than $O(n^2)$. Second, does a lower number of maintenance queries imply a reduction in total maintenance time. Or, put differently, what are the key factors that affect the maintenance performance. The remaining sections of this paper now explore these questions.

3 Shared Common Subexpressions

One way to reduce the number of maintenance queries is to identify and then share any common subexpressions between different maintenance steps of an overall maintenance plan. Studying the batch maintenance plan (See Query 5), we observe that a large number of common subexpressions do exist between different maintenance steps due to the regularity of the view definition structure. For example, the first two maintenance steps of the formula have the common subexpressions $R_3 \bowtie \ldots \bowtie R_n$, while the second and the third steps have R'_1 and $R_4 \bowtie \ldots \bowtie R_n$ in common. If we share such common expressions, the total number of maintenance queries (join operations) would be reduced.

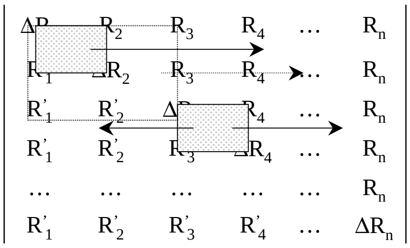

Figure 3: Group Adjacent Steps to Share Common Expressions

Our depiction of the batch maintenance plan highlights the common expressions between adjacent maintenance steps. In Figure 3, we illustrate one possible heuristic for sharing such common expressions. Namely, we can divide the maintenance steps equally and group the adjacent steps along the main diagonal. Then we share the rest of the expressions in grouped steps. For example, if we group steps by two, then the first two steps become $(\Delta R_1 \bowtie R_2 + R'_1 \bowtie \Delta R_2) \bowtie R_3 \bowtie \ldots \bowtie R_n$. The total number of maintenance queries for these two steps is reduced from $2(n\text{-}1)$ to n. While for the third and the fourth steps, we rewrite it to $R'_1 \bowtie R'_2 \bowtie (\Delta R_3 \bowtie R_4 + R'_3 \bowtie \Delta R_4) \bowtie \ldots \bowtie R_n$, and so on. If we divide steps equally, i.e., we group m adjacent steps together along the main diagonal, then the total number of join operations can be described by Equation 6.

$$\lfloor\frac{n}{m}\rfloor(m(m-1)+(n-m))+\Re,\ \Re=(n-\lfloor\frac{n}{m}\rfloor m)(n-1). \tag{6}$$

$\Re$ includes the leftover factors for the n that can't be divided by m. By simple calculation, we know that when m is around $\sqrt{n}$, then the total number of join operations will reach its minimum. Other grouping criteria are also possible, i.e., grouping maintenance steps unevenly based on the estimated delta size. By sharing common expressions, we can reduce the total number of maintenance queries to $O(n^{3/2})$.

4 Grouping Strategies

In the above strategy, we combine different deltas together if and only if they have the same schema. We now relax this constraint and introduce two new approaches that reduce the total number of maintenance queries to O(n) by combining more *delta changes* together. To simplify the discussion, we assume that the view is defined upon n distributed data sources denoted by $R_1 \bowtie R_2 \ldots \bowtie R_n$ and has the corresponding *delta changes* ΔR_1, ΔR_2, ..., ΔR_n need to be maintained [5].

4.1 Maximal Grouping

We now introduce the *maximal grouping* strategy maintains a view extent incrementally in n combined maintenance queries. Assume we have one global vector T at the view manager to hold the temporary results. We use T_k^i $(i \geq 0, 1 \leq k \leq n)$ to represent the value of the kth element in T corresponding to the data source R_k after the ith query. Initially, we have $T_k^0 = \Delta R_k$ $(1 \leq k \leq n)$. We define $R'_k = R_k + \Delta R_k$ $(1 \leq k \leq n)$. That is, R'_k represents the current state of the data source R_k [6].

Query 1. As the first query, we send T_k^0 $(2 \leq k \leq n)$ to data source R_1 and evaluate the result of $R'_1 \bowtie T_k^0$ $(2 \leq k \leq n)$. The result is sent back to T for the next process (See Figure 4). Note that several issues need to be considered for this process, i.e, how to send these *delta changes* together as a combined query. These issues will be discussed in Sections 4.3 and 4.4. Here we simply investigate the feasibility of doing this at a logical level. Thus, after the first query, $T_k^1 = T_k^0 \bowtie R'_1 = \Delta R_k \bowtie R'_1$ $(2 \leq k \leq n)$, $T_1^1 = T_1^0 = \Delta R_1$.

[5] It also can be extended to handle more general SPJ views. We discuss this extension with more details in [11].

[6] The concurrency issue will be discussed in Section 4.3.

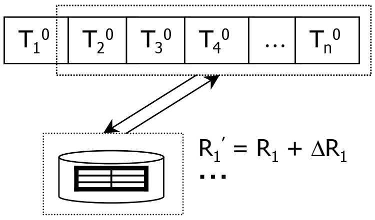

Figure 4: Sending Deltas to Source 1

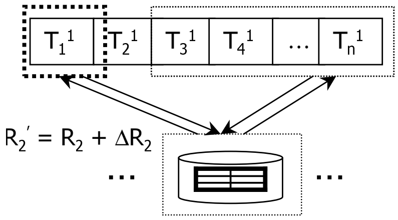

Figure 5: Sending Deltas to Source 2

Query 2. In the second query, we send T_k^1 $(1 \leq k \leq n, k \neq 2)$ to data source R_2 and evaluate the result of $T_k^1 \bowtie R_2'$ $(1 \leq k \leq n, k \neq 2)$. However, we should evaluate the result for data sources that have been visited in previous steps based on R_2 instead of R_2'. Thus, $T_1^2 = T_1^1 \bowtie R_2'$ (T_1^2) - $T_1^1 \bowtie \Delta R_2$. We refer to this as the *compensation operation*. This overall process is depicted in Figure 5. The bold-dashed box in T_1^1 indicates that necessary compensation work may have to be done before the next query. After the second query, we have $T_1^2 = \Delta R_1 \bowtie R_2$, $T_2^2 = \Delta R_2 \bowtie R_1'$, $T_k^2 = \Delta R_k \bowtie R_1' \bowtie R_2'$ $(3 \leq k \leq n)$.

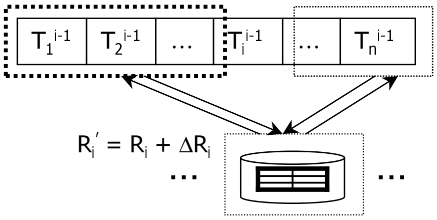

Figure 6: Sending Deltas to Source i

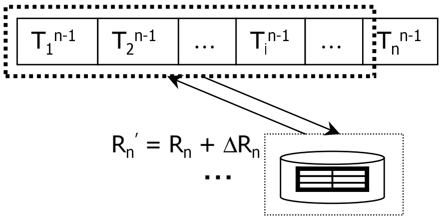

Figure 7: Sending Deltas to Source n

Query i. Let's generalize the process for any ith query $(2 < i \leq n)$. In query i, we send T_k^{i-1} $(1 \leq k \leq n, k \neq i)$ to R_i and get the result to be $T_k^i = T_k^{i-1} \bowtie R_i'$. After that, we compensate the results T_k^i using ΔR_i for $1 \leq k \leq i-1$. Thus, $T_k^i = T_k^{i-1} \bowtie R_i'$ $(i < k \leq n)$, $T_k^i = T_k^{i-1} \bowtie R_i$ $(1 \leq k \leq i-1)$. This process is depicted in Figure 6. The bold-dashed box again indicates the necessary compensation work. After this query, $T_i^i = \Delta R_i \bowtie R_1' \bowtie R_2' \bowtie \ldots \bowtie R_{i-1}'$, $T_k^i = \Delta R_k \bowtie R_1' \bowtie R_2' \bowtie \ldots \bowtie R_i'$ $(i < k \leq n)$, $T_k^i = R_1' \bowtie R_2' \bowtie \ldots \bowtie \Delta R_k \bowtie R_{k+1} \ldots \bowtie R_i$ $(1 \leq k < i)$.

Query n. As described in Query i, we send T_k^{n-1} $(1 \leq k \leq n-1)$ to R_n and compensate the result using ΔR_n. This is depicted in Figure 7. Thus, we have $T_1^n = \Delta R_1 \bowtie R_2 \bowtie R_3 \ldots \bowtie R_n$, $T_2^n = R_1' \bowtie \Delta R_2 \bowtie R_3 \ldots \bowtie R_n$, ..., $T_n^n = R_1' \bowtie R_2' \bowtie R_3' \ldots \bowtie \Delta R_n$. Clearly, the union of T_k^n $(1 \leq k \leq n)$ is equal to the formula we have developed for the batch maintenance plan (Query 5).

Thus, we end up with n maintenance queries using *delta changes* to calculate the incremental view extent which traverse each data source once. However, one weakness of maximal grouping is the possible **large intermediate**

result size caused by the lack of a join condition between some of the intermediate results and the data source. For example, we will send $\Delta R_2, \Delta R_3, ..., \Delta R_n$ to data source R_1 in the first query. Only R_2 has the join condition with R_1 given the view is defined by $R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$. Thus, to evaluate the result $\Delta R_k \bowtie R'_1$ $(3 \leq k \leq n)$, we may have to compute the cartesian product instead. Given that the size of each data source may be huge, this approach is thus likely not to be feasible for many practical settings.

4.2 Conditional Grouping

To address the above shortcoming, we propose a *conditional grouping* algorithm, which makes use of join conditions in the view definition to group deltas. We divide the conditional grouping algorithm into two phases, called **Scroll Up** and **Scroll Down**. As in Section 4.1, vector T holds the temporary results. T_k^i represents the value of the kth element in the vector T corresponding to data source R_k after the ith query. The initial value of vector T is $T_1^0 = \Delta R_1$, $T_k^0 = \emptyset$ $(2 \leq k \leq n)$.

Scroll Up Phase. There are n - 1 queries in this phase. First, we send T_1^0 (ΔR_1) to R_2 and get the result $T_1^1 = \Delta R_1 \bowtie R'_2$. We then compensate the result using ΔR_2. Thus, $T_1^1 = T_1^0 \bowtie R'_2 - T_1^0 \bowtie \Delta R_2 = T_1^0 \bowtie R_2$. After that, we set $T_2^1 = \Delta R_2$. We have $T_1^1 = \Delta R_1 \bowtie R_2$, and $T_2^1 = \Delta R_2$ after the first query.

In the second query, we send T_1^1 and T_2^1 to R_3 and get the result T_1^2= $T_1^1 \bowtie R'_3$, $T_2^2 = T_2^1 \bowtie R'_3$. We then compensate the results using ΔR_3. Thus, $T_1^2 = T_1^1 \bowtie R_3$, $T_2^2 = T_2^1 \bowtie R_3$. After that, we set $T_3^2 = \Delta R_3$. We have $T_1^2 = \Delta R_1 \bowtie R_2 \bowtie R_3$, $T_2^2 = \Delta R_2 \bowtie R_3$ and $T_3^2 = \Delta R_3$ after the second query.

To generalize the process, in any ith query, we do the following four operations. First, we send all T_k^{i-1} $(1 \leq k \leq i)$ to R_{i+1}. Second, we evaluate the result $T_k^i = T_k^{i-1} \bowtie R'_{i+1}$ $(1 \leq k \leq i)$. Third, we compensate the result using ΔR_i. Thus, $T_k^i = T_k^i$ - $T_k^{i-1} \bowtie \Delta R_{i+1}$ $(1 \leq k \leq i)$. Finally, we set $T_{i+1}^i = \Delta R_{i+1}$. We have $T_k^i = \Delta R_k \bowtie R_{k+1} \bowtie \ldots \bowtie R_{i+1}$ $(1 \leq k \leq i)$, $T_{i+1}^i = \Delta R_{i+1}$ after ith query.

After the (n-1)th query, we get the scroll up phase result $T_k^{n-1} = \Delta R_k \bowtie R_{k+1} \ldots \bowtie R_n$ $(1 \leq k \leq n)$.

Scroll Down Phase. There are also n-1 queries in this phase. For simplicity, we use another vector W to hold the temporary result. Initially, we have $W_k = T_k^{n-1}$ $(1 \leq k \leq n)$. In the first query, we send W_n to source R_{n-1}, and get the result $W_n \bowtie R'_{n-1}$. That is $W_n = \Delta R_n \bowtie R'_{n-1}$, $W_k = V_k^{n-1}(1 \leq k \leq n-1)$. Note that no compensation is necessary in this phase.

In second query, we send W_{n-1} and W_n to R_{n-2} and evaluate the result. Thus, $W_{n-1} = W_{n-1} \bowtie R'_{n-2}$, $W_n = W_n \bowtie R'_{n-2}$, $W_k = T_k^{n-1}(1 \leq k \leq n-2)$.

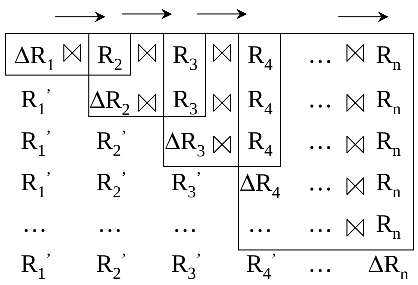

Figure 8: Scroll Up Phase

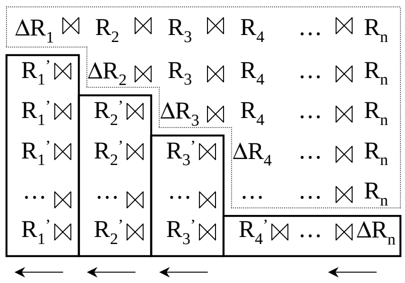

Figure 9: Scroll Down Phase

In general, in any ith query $(2 < i \leq n-1)$, we send W_k $(n-i < k \leq n)$ to R_{n-i} and get the result $W_k = W_k \bowtie R'_{n-i}$ $(n-i < k \leq n)$. After the (n-1)th query, $W_1 = \Delta R_1 \bowtie R_2 \bowtie R_3 \ldots \bowtie R_n$, $W_2 = R'_1 \bowtie \Delta R_2 \bowtie R_3 \ldots \bowtie R_n$, …, $W_n = R'_1 \bowtie R'_2 \bowtie R'_3 \ldots \bowtie \Delta R_n$. Thus, the union of these $W_k (1 \leq k \leq n)$ equals the batch maintenance plan result.

The overall process of these two phases is depicted in Figures 8 and 9. That is, the scroll up phase calculates the upper part along the main diagonal in n-1 queries, while the scroll down phase computes the remaining part of the computation matrix in n-1 queries.

4.3 Handling Concurrent Updates

In the algorithms proposed thus far, we assume that there is no concurrency interfering with the current view maintenance plan. This can be easily achieved by a multi-version based system [3] because we can always retrieve the right data source states from versioned data. However, if a compensation based approach is being used such as [4], concurrent updates have to be considered. To address this, we propose the following method to maintain the view extent under concurrent environments.

We use two vectors to hold source updates: the **current vector** (**CV**) holds the *delta change* per source that currently is being maintained, while the **concurrent vector** (**CONV**) holds all updates that occur concurrently to the current maintenance plan. Initially, CONV is empty because all source updates will be put into CV. After we begin to maintain the deltas in CV, newly incoming updates will be put into CONV. As usual, we use R_i $(1 \leq i \leq n)$ to represent its original data source state. While R'_i $(R'_i = R_i + \Delta R_i, 1 \leq i \leq n)$ represents the state that incorporates the effect of source updates in vector CV (ΔR_i). We use R^c_i $(R^c_i = R'_i + \Delta R^c_i, 1 \leq i \leq n)$ to represent the current state, where ΔR^c_i denotes the corresponding delta changes accumulated in CONV.

Figure 10 illustrates the process of putting source updates into the two vectors. As done in the literature [1, 3, 18], we assume that all messages transfer between sources and the view manager using a FIFO scheme. That

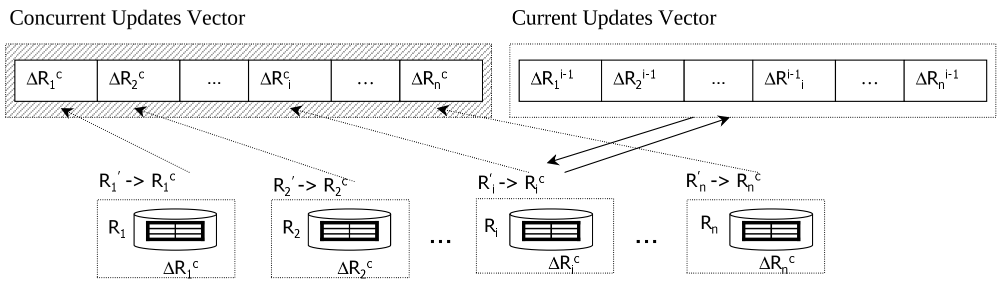

Figure 10: Concurrent Communications between Sources and the View Manager

is, all updates happen on a data source R_i ($1 \leq i \leq$ n) after we have evaluated the maintenance query upon this data source (R_i) will also arrive at the vector CONV after the view manager gets the result of this maintenance query. That is, we can use *delta changes* in both vectors ($\Delta R_i, \Delta R_i^c$) to restore the right data source states, either R_i' or R_i, when the view manager gets the result of a certain maintenance query. As an example, the conditional grouping algorithm in a concurrent environment is revised as follows:

Scroll Up Phase. In ith query, we send all T_k^{i-1} ($1 \leq k \leq i$) to R_{i+1} and get the result $T_k^i = T_k^{i-1} \bowtie R_{i+1}^c$ ($1 \leq k \leq i$). We then compensate the result using $\Delta R_i + \Delta R_i^c$ locally in the view manager, that is, $T_k^i = T_k^i$ - $T_k^{i-1} \bowtie (\Delta R_i + \Delta_i^c)$ ($1 \leq k \leq i$). After the compensation, we set $T_{i+1}^i = \Delta R_{i+1}$.

Scroll Down Phase. There is no need to compensate in the scroll down phase if there are no concurrent updates. However, given any concurrent updates, then we use ΔR_k^c ($1 \leq k \leq n$) to compensate the result. Thus, in any ith query ($1 \leq i \leq n-1$), we send W_k ($n-i < k \leq n$) to R_{n-i} and set the result $W_k = W_k \bowtie R_{n-i}^c$ - $W_k \bowtie \Delta R_{n-i}^c$ ($n-i < k \leq n$) for the next query.

After these two maintenance phases, the vector W contains the effects on the view extent which exactly only incorporates the source updates in the CV. After we refresh the view extent, we simply dump the delta changes in the CONV to the CV and set $R_k = R_k'$ ($1 \leq k \leq n$). Thereafter, we can repeat the maintenance process for the next set of collected updates.

4.4 Grouping Deltas Together

Up to now, we haven't indicated how we combine different deltas and use such heterogeneous deltas to evaluate the maintenance result. For example, consider building a combined maintenance query containing *delta changes* $\Delta R_1 \bowtie R_2$ and ΔR_2. If the query engine at the data source were advanced, it could exploit the similarity among the queries to scan the source relation once when processing all these deltas. However, data sources may not be that advanced. Thus, we instead propose a non-intrusive method to address this issue of unifying different deltas that are from different data sources. The basic idea is to construct one large table that contains the schema of different deltas and fill the respective unrelated fields with default values. We ship this table to the data source as one large delta and evaluate the result together. After we get the result, we split the large query result back into different deltas per source at the view manager. We may even append certain identification related information to the delta so we can split the query result back into deltas more easily. As shown in Figure 11, instead of sending delta tables $\Delta R_1 \bowtie R_2$ and ΔR_2 to the data source R_3 separately, we build a union table which contains the information of both deltas and send them together to R_3 to evaluate the maintenance result in one pass.

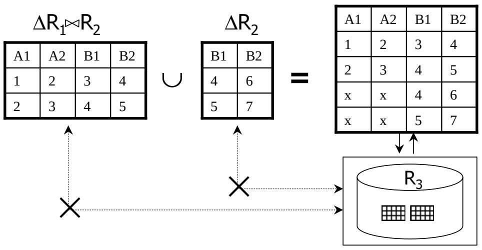

Figure 11: Example of Unifying Different Deltas

5 Cost Model Analysis

We provide a cost model to analyze the maintenance performance in terms of total processing time. We mainly focus on two key variables in this analysis: the network cost of transferring data between the view manager and the distributed data sources, denoted by C_{net}, and the cost of evaluating a maintenance query (join operation), represented by C_{join}. To simplify the description of the cost model, we don't consider the compensation cost in our cost model. In fact, no compensation cost would exist if we were to apply a multiversion based concurrency control strategy. This happens indeed to be the view environment we have at our disposal for our experimental study (See Section 6). We use the following assumptions to simplify the formula we develop:

- We assume all sources are identical in terms of the cost of answering similar maintenance queries. That is, the cost of $\Delta \bowtie R_i$ equals the cost of $\Delta \bowtie R_j$ given a delta change Δ $(1 \leq i, j \leq n)$. Thus, we use R to represent a data source R_i $(1 \leq i \leq$ n).
- We assume k updates evenly distributed among n data sources. Thus, each source experiences k/n updates. We assume that the cost of $\Delta R_i \bowtie R$ equals the cost of $\Delta R_j \bowtie R$ $(1 \leq i, j \leq n)$.

For simplicity, we use Δs to represent a single source update, and ΔS to denote the *delta changes* of each source. To represent the maintenance query result, we define $\Delta S_{i+1} = \Delta S_i \bowtie R$ with $\Delta S_1 = \Delta S$ $(1 \leq i \leq n-1)$. For a single source update, we also define $\Delta s_{i+1} = \Delta s_i \bowtie R$ with $\Delta s_1 = \Delta s$.

For a sequential maintenance plan, the view manager takes one source update at a time, sends maintenance queries to the sources one by one and collects the result. This process is repeated until all k source updates have been maintained. Thus the total cost for maintaining k source updates can be represented by Equation 7. Similarly, the cost of the basic batch maintenance (Section 1.2) is given by Equation 8. $C_{net}(\Delta s_i)$ or $C_{net}(\Delta S_i)$ represents the cost of sending the maintenance query while $C_{net}(\Delta s_{i+1})$ or $C_{net}(\Delta S_{i+1})$ denotes the cost of transferring maintenance result. C_{join} denotes the cost of answering the corresponding maintenance query.

$$T_s = k \sum_{i=1}^{n-1} [C_{net}(\Delta s_i) + C_{join}(\Delta s_i) + C_{net}(\Delta s_{i+1})] \quad (7)$$

$$T_b = n \sum_{i=1}^{n-1} [C_{net}(\Delta S_i) + C_{join}(\Delta S_i) + C_{net}(\Delta S_{i+1})] \quad (8)$$

For the shared common expression approach, we assume that we divide the maintenance steps evenly into a group of size m. Then the cost can be measured by Equation 9. $m\Delta S_i$ denotes the union of m ΔS_is.

$$T_c = \frac{n}{m} \{ m \sum_{i=1}^{m-1} [C_{net}(\Delta S_i) + C_{join}(\Delta S_i) + C_{net}(\Delta S_{i+1})] + \sum_{i=m}^{n-1} [C_{net}(m\Delta S_i) + C_{join}(m\Delta S_i) + C_{net}(m\Delta S_{i+1})] \} \quad (9)$$

Based on the same assumptions, the cost of conditional grouping is given in Equation 10.

$$T_g = \sum_{i=1}^{n-1} [C_{net}(\sum_{j=1}^{i} \Delta S_j) + C_{join}(\sum_{j=1}^{i} \Delta S_j) + C_{net}(\sum_{j=2}^{i+1} \Delta S_j)] + \sum_{i=1}^{n-1} [C_{net}(i\Delta S_i) + C_{join}(i\Delta S_i) + C_{net}(i\Delta S_{i+1})] \quad (10)$$

The trade-off between these maintenance strategies is that as we reduce the number of maintenance queries to distributed sources, we do increase the message size in each C_{net} and C_{join}. To accentuate this difference, we further simplify the cost model by assuming that $\Delta S_{i+1} = \Delta S_i$. That is, the maintenance query result equals in size with that of the maintenance query. In Figure 12, we depict that trade-off based on the simplified model. If the cost of C_{join} and C_{net} on a *delta change* with size k (combining k updates) is much smaller than k times the cost of C_{join} and C_{net} with size 1 (one update), performance improvements are expected by reducing the number of maintenance queries.

k:	# of Source Updates
n:	# of Data Sources
C_{net}:	Network Cost
C_{join}:	Cost of Maintenance Query
Δ:	Source Specific Delta Change
$v\Delta$:	Union of v Deltas

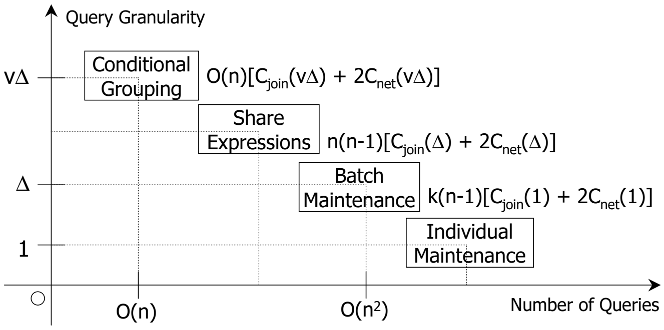

Figure 12: Simplified View Maintenance Cost Model

6 Experimental Evaluations

6.1 Experimental Testbed

We have implemented these proposed maintenance strategies within the TxnWrap system developed at WPI [3]. TxnWrap is a multiversion-based view maintenance system which removes concurrency control concerns from its maintenance logic. Thus, it is not necessary to apply compensation for handling concurrent source updates in our setting. The basic TxnWrap system maintains one single source update at a time using the known SWEEP algorithm [1]. The batch TxnWrap [10] combines the updates from the same data source and maintains the view extent using the delta change for each source.

In our experimental study, we have conducted our experiments on four Pentium III PCs with 256M memory, running Windows NT and Oracle 8*i*. We employ six data sources with one relation each that are evenly distributed over three PC machines. Each relation has two attributes and 100,000 tuples. One materialized join view is defined upon these six source relations residing on a separate (the fourth) machine.

6.2 Impact of Source Dependent Properties

We first investigate the impact of source dependent properties (**cooperative** versus **non-cooperative** in particular) on the maintenance performance. A non-cooperative source only answers maintenance queries (SQL queries), but offers no other services or controls to the view manager. A cooperative data source would cooperate with the view manager by allowing to synchronize processes or lock its data. This means to evaluate the operation $\Delta R_i \bowtie R_j$ against a non-cooperative data source, we have to use a composite SQL query which unions maintenance queries for a single source update to evaluate the result. While a cooperative source would allow the view manager to build a temporary table directly at the data source, ship the delta data, evaluate it locally and send the result back. These two methods of evaluating maintenance queries are different. In Figures 13, 14 and 15, we vary the number of data updates from 10 to 150 (and then from 500 to 2000) with all updates from the same source (on x-axis). The y-axis represents the maintenance time.

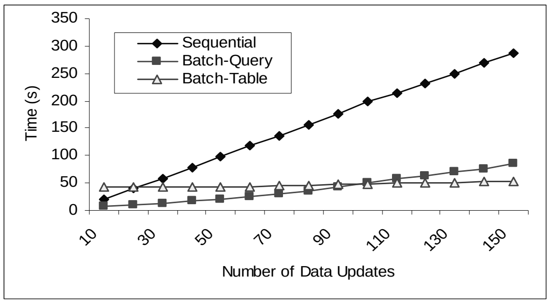

Figure 13: Batch a Small Number of Updates

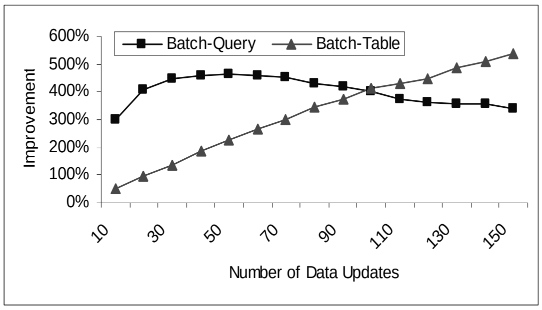

Figure 14: Performance Ratio of Bat. over Seq.

From Figure 13, the processing time of batch maintenance using a composite query increases slowly. For the temporary table approach, the increase of the total cost is even slower than that of using a composite query. This is due to the fact that the setup cost (create temporary table and populate its extent) dominates the actual maintenance expenses for small cases. This also explains that with a small number of updates, batch processing using a temporary table is even more expensive than sequential maintenance.

Figure 14 displays the ratio of the sequential processing time divided by batch processing using the data gotten from Figure 13. The higher the ratio, the larger a performance improvement is achieved. We observe that when the number of updates is around 50 in our current setting, the composite query batch processing approach is the most efficient. While for batch maintenance using temporary tables, the ratio increases steadily.

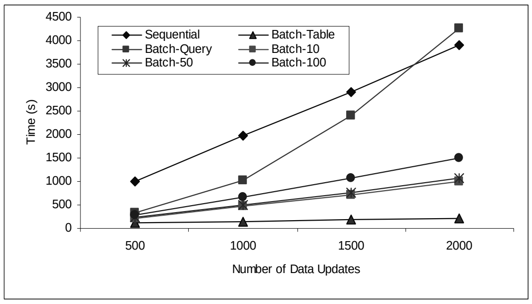

Figure 15: Batch Large Number of Updates

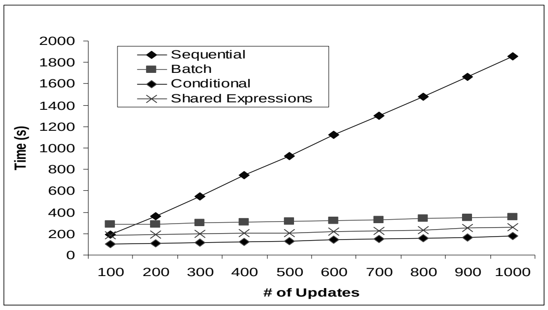

Figure 16: Grouping Small Number of Updates

We see that the larger the number of source updates being grouped, the lower the improvement of the maintenance cost when using the composite query approach. In Figure 15, if we increase the number of source updates, then the cost of batch maintenance using composite query approach becomes even worse than sequential processing.

A composite query composed of the union of a large number of queries will result in a cost increase compared to the cost of sequential processing. Thus, for this composite query approach, we instead suggest to divide such a large number of updates into smaller subbatch queries of size k based on the ratio measured in Figure 14. The cost of the sum of these subqueries will be smaller than that of the one large composite query. As seen in Figure 15, when we choose k equal 50, the total maintenance cost using a composite query approach will reach its optimum in our setting. However, if we use the temporary table approach, the total cost is even lower than that of the optimized composite query approach. This is because the ratio of the increase of each such batch maintenance query to the increase in the number of source updates is very low. Without loss of generality, we utilize this more efficient temporary table approach as batching representive when comparing our new proposed strategies.

6.3 Change the Number of Source Updates

Figure 16 shows the average maintenance cost (on the y-axis) by varying the number of source updates from 100 to 1000 (on the x-axis). These updates are evenly distributed among six data sources. From Figure 16, batch processing takes less processing time than sequential processing due to the reduction in the number of maintenance queries by grouping source updates together. If we further group deltas, then the total maintenance time can be even more reduced. We see such maintenance cost relationship: '*conditional grouping < shared common expressions < batch processing*'. Given that the shared common expressions approach is a medium perfomer between the batch and conditional grouping, we will focus on comparing in more depth the batch and conditional grouping strategies below.

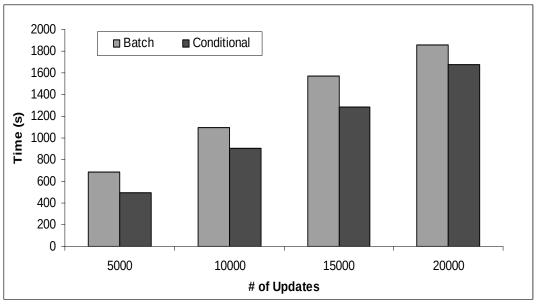

Figure 17: Grouping Large Number of Updates

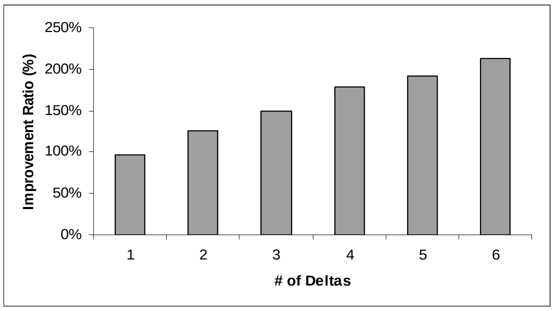

Figure 18: Change the Distributions of Updates

Figure 17 measures the performance changes for both strategies for an increasing number of source updates. The maintenance cost of both approaches increases steadily as the size of each delta increases. The conditional grouping outperforms batch maintenance due to the size of delta not being a major factor on the Oracle query cost if we use the temporary table approach and the conditional grouping has a smaller number of maintenance queries.

6.4 Change the Distribution of Source Updates

We examine the impact of the distribution of 1,000 updates among the data sources on the maintenance performance (Figure 18). On the x-axis, a distribution of 1 denotes that all 1000 updates come from one source, while 6 indicates that these updates are evenly distributed over six data sources. Figure 18 presents the cost ratio (batch maintenance cost divided by grouping maintenance cost). Clearly, the more data sources are involved, the higher the performance improvement. This is because the total number of maintenance queries in batch maintenance changes from 5 to 30 queries if we increase the distribution from 1 to 6 sources, while the conditional grouping only changes from 5 to 10 correspondingly. Thus more improvement is achieved by further reducing the number of maintenance queries.

6.5 Impact of the Join Ratio

We use 200 updates on six sources and vary the join ratio from 0.5 to 3.0 (on x-axis). We define the join ratio as the average number of tuples affected by a source change. For example, a join ratio equals to 2 means that a single update which changes a tuple in the source may cause 2^5 tuples to be updated in the view extent. From Figure 19, we see that the higher the join ratio, the higher both maintenance costs. A high join ratio increases the size of each temporary maintenance result, which in turn increases the time to answer the maintenance query. Both maintenance costs increase rapidly when the join ratio increases. This is because the result size is exponential based on the join ratio in our setting. Also, the higher the join ratio, the closer these two maintenance costs become (Figure 19). Clearly, any change in the temporary result size will be amplified by a exponential factor based on the join ratio. Thus, the benefit of having a smaller number of maintenance queries is overtaken by the increase of each query cost.

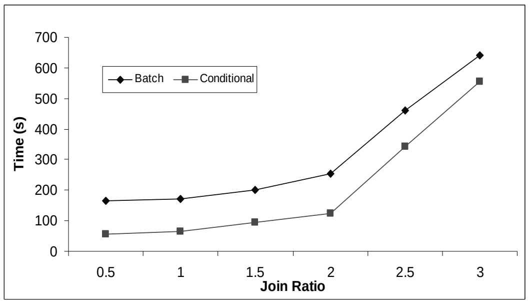

Figure 19: Change the Join Ratio in the View

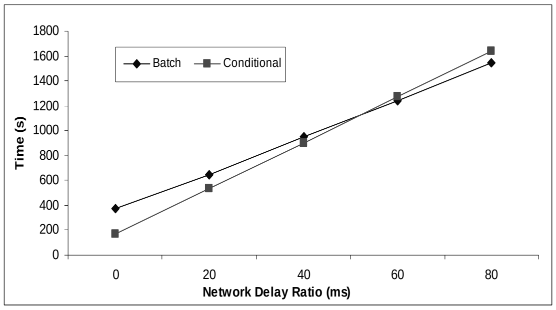

Figure 20: The Impact of Network Delay

6.6 Impact of the Network Delay

To evaluate the impact of different data transfer rates of the network, we insert delay factors before evaluating each query. The delay is generated based on the average time to transfer one tuple. For example, if we assume that the average time to transfer a tuple with 2 attributes is ℓ, then it takes $100*2*\ell$ to transfer one delta with 100 tuples with 4 attributes per tuple. We use 1000 data updates on six sources and vary ℓ from 0 ms to 80 ms. On Figure 20, both maintenance costs grow steadily as the network cost of each maintenance query is increasing. In a typical network environment where the transfer time of one tuple with 2 attributes (<20 characters) is less than 30 ms, conditional grouping is more efficient than the batch method because we have a smaller number of maintenance queries. However, in a slow network, i.e, when the average transfer time for one tuple is larger than 60 ms, then the gain gotten by reducing the number of maintenance queries is overtaken by the increase in the network cost of each query. In conditional grouping, we may have some extra data (null values) transferred on the network. This extra data becomes a burden in a slow network.

7 Related Work

Maintaining materialized views under source updates is one of the important issues in information integration [19]. Initially, some research has studied incremental view maintenance assuming no concurrency [5, 12]. In approaches that need to send maintenance queries to the data sources, especially in a distributed environment with autonomous data sources, concurrency problems can arise. [19] introduces a compensation-based algorithm, called ECA, for incremental view maintenance under concurrent source data updates restricted to a single source, while Strobe [20] handles the concurrency which comes from multiple data sources. SWEEP [1] applies local compensation of maintenance over distributed sources.

In situations where the real time refresh of the view extent is not critical, changes to the sources can be buffered and propagated periodically to maintain the view extent to gain better maintenance performance. [13, 15, 5, 16, 8] propose algorithms to maintain materialized views incrementally using grouped source updates. In our previous work [10], we have proposed a batch view maintenance strategy that works even when both data and schema changes may happen on data sources. However, these existing approaches are only concerned with batching updates from the same source. [9] introduces a delta propagation strategy that reduces the number of maintenance queries to data sources. However, it only focuses on sharing common subexpressions in maintenance steps.

Distributed query processing [7] is well studied which focuses on query optimization in a distributed environment.

It provides techniques such as in what order to compute a sequence of joins which orthogonal to what we explore here. Making use of shared common expressions is well studied in multiple query optimization area [17]. In the context of optimizing maintenacne plans, common subexpressions could possibly be manually identified because the maintenance queries are relatively fixed given a view definition. [14] discusses how to send maintenance queries to data sources. It is also orthogonal to our proposed strategies because it only focuses on how to optimize the execution of one single maintenance query.

8 Conclusion

In this paper, we have taken a fresh new look at how to realize a given view maintenance plan by exploring the trade-off between the number of maintenance queries and their complexity. These maintenance strategies have been implemented in our TxnWrap view maintenance system. Our experimental studies illustrate that maintenance performance can be greatly improved by reducing the number of maintenance queries. Our experiments also reveal an additional dimension of this design space, namely the cooperation of the data sources in the maintenance process. This is shown to have a significant impact on the performance of such maintenance plans.

References

[1] D. Agrawal, A. E. Abbadi, A. Singh, and T. Yurek. Efficient View Maintenance at Data Warehouses. In *Proceedings of SIGMOD*, pages 417–427, 1997.

[2] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently Updating Materialized Views. In C. Zaniolo, editor, *Proceedings of SIGMOD*, pages 61–71, Washington, DC, May 1986.

[3] J. Chen, S. Chen, and E. A. Rundensteiner. A Transactional Model for Data Warehouse Maintenance. In *ER'02*, pages 247–262, Sep 2002.

[4] J. Chen, X. Zhang, S. Chen, K. Andreas, and E. A. Rundensteiner. DyDa: Data Warehouse Maintenance under Fully Concurrent Environments. In *Proceedings of SIGMOD Demo Session*, page 619, 2001.

[5] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for Deferred View Maintenance. In *Proceedings of SIGMOD*, pages 469–480, 1996.

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

[7] D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys (CSUR)*, 32(4):422–469, 2000.

[8] W. J. Labio, R. Yerneni, and H. García-Molina. Shrinking the Warehouse Updated Window. In *Proceedings of SIGMOD*, pages 383–395, June 1999.

[9] K. Y. Lee, J. H. Son, and M. H. Kim. Efficient Incremental View Maintenance in Data Warehouses. In *CIKM'01*, pages 349–356, November 2001.

[10] B. Liu, S. Chen, and E. A. Rundensteiner. Batch Data Warehouse Maintenance in Dynamic Environments. In *CIKM'02*, pages 68–75, Nov 2002.

[11] B. Liu and E. A. Rundensteiner. Cost-Based View Maintenances in Distributed Environments. Technical Report In Progress, Worcester Polytechnic Institute, Dept. of Computer Science, 2003.

[12] J. J. Lu, G. Moerkotte, J. Schue, and V. S. Subrahmanian. Efficient Maintenance of Materialized Mediated Views. In *SIGMOD*, pages 340–351, 1995.

[13] I. Mumick, D. Quass, and B. Mumick. Maintenance of Data Cubes and Summary Tables in a Warehouse. In *Proceedings of SIGMOD*, May 1997.

[14] K. O'Gorman, D. Agrawal, and A. E. Abbadi. Posse: A framework for optimizing incremental view maintenance at data warehouse. In *Data Warehousing and Knowledge Discovery*, pages 106–115, 1999.

[15] D. Quass and J. Widom. On-Line Warehouse View Maintenance. In *Proceedings of SIGMOD*, pages 393–400, 1997.

[16] K. Salem, K. S. Beyer, R. Cochrane, and B. G. Lindsay. How To Roll a Join: Asynchronous Incremental View Maintenance. In *SIGMOD*, pages 129–140, 2000.

[17] T. K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems (TODS)*, 13(1):23–52, 1988.

[18] X. Zhang, E. A. Rundensteiner, and L. Ding. Parallel Multi-Source View Maintenance. *VLDB Journal*, 2003. to appear.

[19] Y. Zhuge, H. García-Molina, J. Hammer, and J. Widom. View Maintenance in a Warehousing Environment. In *Proceedings of SIGMOD*, pages 316–327, May 1995.

[20] Y. Zhuge, H. García-Molina, and J. L. Wiener. The Strobe Algorithms for Multi-Source Warehouse Consistency. In *Parallel and Distributed Information Systems*, pages 146–157, 1996.