WPI-CS-TR-03-30 June 2003

Cost-Driven View Maintenance over Distributed Data Sources

by

Bin Liu and Elke A. Rundensteiner

Computer Science
Technical Report
Series

WORCESTER POLYTECHNIC INSTITUTE

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

Cost-Driven View Maintenance over Distributed Data Sources *

Bin Liu and Elke A. Rundensteiner

Department of Computer Science
Worcester Polytechnic Institute
Worcester, MA 01609–2280
{binliu|rundenst}@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. One improtant task of view maintenance is to reduce the time taken for updating the view extent due to the constantly increasing size of the view and the rapid rates of source changes.

In this work, we investigate two of the key issues that affect the view maintenance performance in terms of total processing time. First, the selection of maintenance strategy. Different maintenance strategies will exhibit different performances based on the particulars of their methods being used. For example, a *batch maintenance* strategy is usually more efficient compared with a traditional (sequential) algorithm given a lot of updates need to be maintained due to less number of remote maintenance queries are required. However, not all maintenance strategies are obvious in performance. Second, we study the data source related properties. In a distributed environment, various data sources may show different response time for a maintenance query due to the network connection, data source processing capability etc. Given such variations in the view maintenance environment, we propose a two-layer cost model to analyze the view maintenance performance over distributed data sources. We introduce a framework which is based on our cost model to generate maintenance plans for maintaining a given set of source updates. The generated maintenance plans are tuned to the current environment settings to maintain updates efficiently. This maintenance framework has been implemented in our TxnWrap view maintenance system. Experimental studies illustrate that such cost-driven view maintenance optimization improves view maintenance performance especially in a non-homogenous environment.

Keywords: View Maintenance, Cost Estimation, Distributed Query Processing.

*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, data warehousing and semantic web. 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 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 the 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, 17, 14] for this task of incrementally maintaining materialized view extents. 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. Due to the constantly increasing size of data sources and thus view extents as well as the rapid rates of source changes, there is an increasing pressure to reduce the time taken for view maintenance in practical systems. However, most existing algorithms [19, 1, 3, 17] were not designed with such optimization in mind. This urgency to improve the view maintenance performance given highly dynamic data sources that experience high rate data changes is thus the main focus of our work [1].

In our view maintenance context, we assume that the view is defined over several distributed data sources, data updates happen independently among these data sources, and all updates will be reported to the view manager for keeping the view extent up to date. Complex optimization of view maintenance processes in a distributed environment is a challenging and time consuming process. In this work, we thus propose a separation of concerns into two different levels. First, at the logical level of view maintenance, we introduce the notion of strategy selection, i.e., of selecting one maintenance strategy among possibly several to employ for the maintenance task at hand. For example, a batch maintenance strategy is usually more efficient than sequentially maintaining one update at a time. This is so as batching reduces the number of accesses to the underlying data sources. Thus we may choose a batch maintenance whenever it is applicable, that is, when the real-time refresh of view extent is not needed by the application. However,

[1] The mechanisms to decide when to maintain the materialized views are orthogonal to our work. As described in [14], a time-driven or event-driven strategy can be applied to determine how many updates will be maintained together

not all maintenance strategies have such an obvious performance difference, rather it depends on a variety of factors in the environment. Plus, there are several alternate batching algorithms that could be selected, each exploiting possibly different query-optimization techniques in the generation of their query maintenance plans.

Second, at a physical level of view maintenance, we apply source dependent properties to further tune the performance of a given maintenance strategy and then to generate an optimized maintenance plan. For example, we select the *optimal* sequence of accesses to the data sources to improve the maintenance performance. Note that such separation of concerns is not strict, as the underlying data source properties may also affect the first decision about the selection of the maintenance strategy to begin with.

One could argue that standard distributed query processing techniques [8, 7, 15] may be able to be employed to generate efficient maintenance plans. However, the following two key points distinguish the view maintenance optimization from standard distributed query optimization: 1) Each step in a view maintenance process may change the data source state, which in turn may affect the remaining steps. 2) All the maintenance queries in one maintenance process have a similar query shape. Thus, this opens more opportunities for optimization tuned specifically for maintenance. To our knowledge, a distributed query optimizer does typically not handle optimization at this level.

Our key contributions in this work are:

- We investigate key issues that affect the view maintenance performance and then propose a two-layer optimization approach so to separate the concerns of maintenance into two separate tasks, namely, logical and physical view maintenance optimization.

- We introduce and then enhance two alternative view maintenance strategies optimized to handle not just singleton source updates but also large sets of data updates,

- We design a cost model for view maintenance modeling the two view maintenance strategies. In particular, we then can employ the model to cost out different maintenance plans that each view maintenance strategy would generate. We analyze and identify the trade-offs inherent in these view maintenance alternatives.

- Based on the above strategies and cost model, we then propose a view maintenance optimization framework which is able to generate efficient maintenance plans tuned to the current environmental settings. Thus, it provides us with the needed adaptivity between different maintenance processes by allowing us to dynamically select the most effective maintenance plan.

- We have implemented this maintenance framework in our TxnWrap view maintenance system. Our experimental study shows that our two-layered approach towards solving maintenance optimization, i.e., the generation of optimized maintenance plans, improves the maintenance performance significantly, compared to existing techniques in the literature.

The rest of the paper is organized as follows. Section 2 describes the needed background and terminology, along with the introduction of two different batch maintenance strategies. Section 3 introduces the proposed cost model and cost-based optimization techniques. Section 4 discusses the experimental studies. Related work and conclusions are given in Sections 5 and 6 respectively.

2 View Maintenance Strategies

A *view maintenance strategy* describes the way that a view manager maintains the materialized view at a logical level. Various maintenance strategies have different maintenance performances due to the particulars in their methods being used. For example, a batch maintenance strategy [10, 9]is usually much more efficient compared with a traditional (sequential) algorithm when a large number of updates need to be maintained. This is so because the reduction in the number of remote maintenance queries required. However, a maintenance strategy has less maintenance queries may not always win. In this section, we describe two view maintenance strategies that incrementally maintain the view extent using *delta changes* (one *delta change* describes changes made to a data source in a certain time period). First, we introduce the concept of a *view graph* (VG) to represent a view definition, then we describe these two maintenance strategies based on this view graph.

View Definition Model. We model the view definition using a *view graph*. Each node in the graph represents a data source that appears in the view definition. An edge indicates a join condition in the view definition between two data sources. For example, the view *Tour-Customer* is defined by a SQL query in Query 1 based on the data source descriptions in Figure 1. Its view definition graph is depicted in Figure 2. Note that we put edges into the graph if and only if there are join conditions between the data sources. Other operations in the view definition such as projection and selection can be easily incorporated because they can be applied locally at the data source. Like most maintenance strategies proposed in the literature [1, 20, 17, 14, 10, 9], we assume that the materialized views are SPJ views that integrate data using joins across multiple data sources because SPJ views as a general form can cover the core (and most expensive component) of view definitions. Moreover, a SPJ view can be easily extended, i.e., to handle views with aggregrations [13].

R_1: Customer(Name, Address, Phone)
R_2: FlightRes(Name, Age, FlightNo, Dest)
R_3: Participant(Name, TourID, StartDate, Loc)
R_4: Tour(TourID, Name, Type, Dest)

Figure 1: Description of Data Sources

$$\begin{array}{ll}
\text{CREATE VIEW} & Tour-Customer \text{ AS} \\
\text{SELECT} & C.Name,\ F.Dest,\ F.FlightNo, \\
 & T.TourID,\ P.StartDate \\
\text{FROM} & Customer\ C,\ FlightRes\ F,\ Tour\ T \\
 & Participant\ P \\
\text{WHERE} & C.Name\ =\ F.Name\ and\ F.Name\ =\ T.Name \\
 & and\ T.Name\ =\ P.Name\ and\ P.Loc\ =\ F.Dest \\
 & and\ F.Age\ <=\ '65'
\end{array} \tag{1}$$

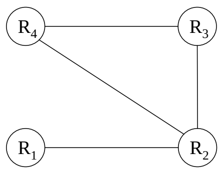

Figure 2: View Definition Graph

Batch View Maintenance. The basic idea of batch maintenance, as described in [9, 10, 14], is to process all the updates from the same data source together. However, given a view definition that may contain multiple join conditions among data sources, more issues need to be considered. For example, R_2 has join conditions both with R_4 and R_3 in Figure 2. Thus, once the join conditions between R_3 and R_4 have been evaluated, we can combine the join conditions R_4-R_2 and R_3- R_2 together and evaluate them at R_2 once. We prefer such combination in a distributed environment because it reduces the number of accesses to remote data sources.

For simplicity, we use $\bowtie_{ij}$ to represent the edge in the view graph which denotes the join condition between data source R_i and R_j. We define D_c as the collection of all data sources that have been visited up to now. For example, $D_c = \emptyset$ initially, and if we have evaluated join condition R_4-R_3, then $D_c = \{R_4,\ R_3\}$. We define $\bowtie_{>j}$ as the collection of the all join conditions (edges) that have the following two properties: 1) Each $\bowtie_{ij}$ in $\bowtie_{>j}$ is a valid edge in the VG. 2) Each R_i in $\bowtie_{ij}$ has $R_i \in D_c$. Here each $\bowtie_{>j}$ is only a abstract operator on source R_j, the content of this operator depends on the data sources that have been visited. Query 2 describes the batch view maintenance strategy assuming the view is defined across n data sources $(R_1, R_2, \ldots, R_n)$. Here R_i represents the underlying data source, ΔR_i $(1 \leq i \leq \text{n})$ represents the *delta change* of a data source R_i, and $R_i' = R_i + \Delta R_i$ $(1 \leq i \leq \text{n})$.

$$
\begin{aligned}
\Delta V \quad &= \Delta R_1 \bowtie_{>2} R_2 \bowtie_{>3} R_3 \ldots \bowtie_{>n} R_n \\
&+ R_1' \bowtie_{>1} \Delta R_2 \bowtie_{>3} R_3 \ldots \bowtie_{>n} R_n \\
&+ \ldots \\
&+ R_1' \bowtie_{>1} R_2' \bowtie_{>2} R_3' \ldots \bowtie_{>n-1} \Delta R_n
\end{aligned}
\tag{2}
$$

We refer to each line in Query 2 as a *maintenance step*. Each maintenance step goes through all the underlying data sources once except the one has the *delta change* to be maintained. We also refer to each join operation within such a step as a *maintenance query*. Each maintenance query goes to one data source. Thus, to maintain the view extent, we may have to send n*(n-1) maintenance queries (join operations) that involve two states of each underlying data sources. Note that concurrent source updates could happen during this maintenance process. Thus an additional concurrency control strategy [2] is nessary to keep the view extent consistent [20].

However, given a maintenance step having $\bowtie_{>j}$ $(1 \leq j \leq n)$, there are multiple valid instances of execution because the order of visiting data sources will affect the content of the following $\bowtie_{>j}$ operations. For example, the following two queries (Queries 3 and 4) are both valid instances of a maintenance step $R_1' \bowtie_{>1} \Delta R_2 \bowtie_{>3} R_3 \bowtie_{>4} R_4$ based on the view definition graph in Figure 2.

$$
\Delta R_2 \bowtie_{23} R_3 \begin{pmatrix} \bowtie_{34} \\ \bowtie_{24} \end{pmatrix} R_4 \bowtie_{14} R_1' \tag{3}
$$

$$
\Delta R_2 \bowtie_{12} R_1' \bowtie_{24} R_4 \begin{pmatrix} \bowtie_{43} \\ \bowtie_{23} \end{pmatrix} R_3 \tag{4}
$$

We refer to the collections of all valid instances that one for each maintenance step as a *batch maintenance plan*. Thus, a lot of batch maintenance plans exist for one given view definition due to each maintenance step may have multiple valid instances. These maintenance plans may exhibit different maintenance performances due to differences among the join conditions and the data sources. Thus, the key to optimize batch maintenance performance is to find an efficient maintenance plan with least estimated cost. In section 3, we will enhance our view definition model with corresponding cost information along with search algorithms to generate such efficient batch maintenance plans.

[2]Either a compensation based or multiple version based approach can be applied, please refer to [4, 3] for more details.

Grouping View Maintenance. In our previous work [11], we proposed a grouping maintenance strategy that maintains materialized views defined as joins across multiple data sources using 2*(n-1) maintenance queries. The basic idea is to make use of the regularity in the maintenance steps. All the maintenance steps have a similar pattern which always goes though all the underlying data sources based on the join conditions between them. For example, assume the view is defined as follows: V = $R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$. Then the grouping maintenance strategy computes the view extent incrementally using two phases (Figures 3 and 4). Each phase has n-1 maintenance queries given n delta changes $\Delta R_1, \Delta R_2, \ldots, \Delta R_n$.

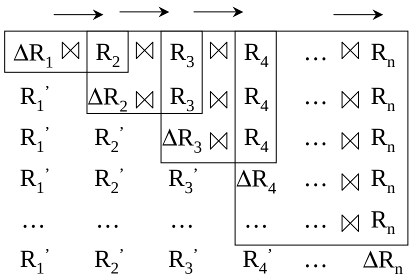

Figure 3: Scroll Up Phase

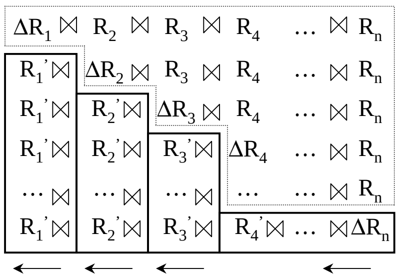

Figure 4: Scroll Down Phase

The *scroll up* phases calculates the upper part of the matrix along the main diagonal in n-1 queries (Figure 3). For example, the first query in the scroll up will send ΔR_1 to data source R_2 and evaluate the result $\Delta R_1 \bowtie R_2$. After we get the result, we then union the result of the first query with ΔR_2 and send them together to data source R_3. We have $\Delta R_1 \bowtie R_2 \bowtie R_3$ and $\Delta R_2 \bowtie R_3$ as the second query result. Then we will union ΔR_3 into the second query result and go on the processing until we reach the data source R_n [3]. While the *scroll down* phase computes the remaining part of the computation matrix also in n-1 queries (Figure 4).

Thus, for a more general view definition defined by a view graph such as the one in Figure 2, the problem can be reduced to find a path that goes through all the data sources once while each node only has a degree of two

[3] Note that there are a lot of issues need to be addressed such as how to union the deltas together, how to process such maintenance combined maintenace queries etc. We ask readers refer to [11] for more detailed information.

except the start and the end node. We refer this as a *grouping path.* Thus, the grouping strategy is divided into two steps: 1) Find a grouping path in the view graph and apply the *scroll up* and *scroll down* phases to calculate the maintenance results based on the join conditions in the grouping path. 2) Apply the remaining join conditions (if any) to the result calculated in the first step. Thus, we will have 2*n-1 maintenance queries to calculate the ΔV (If we evaluate the remaining join conditions on the result of the first step at the view site, the number of maintenance queries will still be 2*(n-1) because these join conditions can be evaluated locally). However, such a grouping path may not exist in all (general) view graphs. For example, a star-shape view graph doesn't have such a path that goes through all nodes with a degree less or equal to two. For these view definitions, the grouping strategy is not available.

As an example, both the following two queries (Queries 5 and 6) are valid instances of the grouping strategy for the view defined in Figure 2. Query 5 chooses the grouping path $R_4 \rightarrow R_3 \rightarrow R_2 \rightarrow R_1$, while Query 6 uses the path $R_1 \rightarrow R_2 \rightarrow R_4 \rightarrow R_3$. The super script represents the remaining join condition(s) that need to be evaluated after we get the maintenance result after the *scroll up* and *scroll down* phases.

$$\begin{pmatrix} \Delta R_4 \bowtie_{43} R_3 \bowtie_{32} R_2 \bowtie_{21} R_1 \\ R_4' \bowtie_{43} \Delta R_3 \bowtie_{32} R_2 \bowtie_{21} R_1 \\ R_4' \bowtie_{43} R_3' \bowtie_{32} \Delta R_2 \bowtie_{21} R_1 \\ R_4' \bowtie_{43} R_3' \bowtie_{32} R_2' \bowtie_{21} \Delta R_1 \end{pmatrix}^{\bowtie_{24}} \tag{5}$$

$$\begin{pmatrix} \Delta R_1 \bowtie_{12} R_2 \bowtie_{24} R_4 \bowtie_{43} R_3 \\ R_1' \bowtie_{12} \Delta R_2 \bowtie_{24} R_4 \bowtie_{43} R_3 \\ R_1' \bowtie_{12} R_2' \bowtie_{24} \Delta R_4 \bowtie_{43} R_3 \\ R_1' \bowtie_{12} R_2' \bowtie_{24} R_4' \bowtie_{43} \Delta R_3 \end{pmatrix}^{\bowtie_{23}} \tag{6}$$

Similarly, we refer to each valid instance of the grouping maintenance strategy as a *grouping maintenance plan.* As seen from above discussions, the basic difference between these two maintenance strategies is that the the number of maintenance queries vs. the complexity of each query. Typically, in a distributed environment, the smaller the number of maintenance queries, the better the maintenance performance. However, given the diversities of maintenance plans even for a given maintenance strategy, we expect cost changes between maintenance strategies.

In the following sections, we discuss our proposed view maintenance optimization framework which both explore the differences between maintenance strategies and the lower (physical) level properties in a view maintenance environment. We first enhance the view definition model to incorporate the related cost information. Then we develop search algorithms to generate efficient maintenance plans for a given set of source updates.

3 VM Optimization Framework

3.1 View Maintenance Cost Model

We extend the view definition graph to incorporate the related cost information. We add the following two functions to each node in the view graph to describe some basic information about the corresponding data sources.

- C_i: The cardinality of the source relation R_i.
- A_i: The number of attributes in relation R_i.

We note that a lot more cost information (factors) related to a data source can be reported to get a more precise or more specific cost model. The examples of additional factors may include the average tuple length, the number of used disk blocks etc. However, in our maintenance framework, we only use the above two factors. As we will illustrate in Section 4, we have found these two factors to be rather effective already in estimating the view maintenance cost. In general, any arbitrary cost information can be incorporated into the framework.

We attach a cost function to each edge in the view graph to estimate the cost of a join operation. In our view maintenance framework, the view manager is responsible of collecting delta changes and intermediate results, and sending maintenance queries to data sources. While the data source is responsible for reporting source updates and answering maintenance queries. Compared with data source relations, the delta changes and the corresponding intermediate results are relatively small. That is, the view manager will always starts from the delta change to evaluate the corresponding maintenance step [4]. Thus, for each edge ($\bowtie_{ij}$) defined in VG, the cost of having the left operand ready in the view manager and evaluating at R_j may different with having the right operand ready in the view manager and evaluating at R_i. Correspondingly, we have two cost estimation functions for each edge. Moreover, a selectivity estimation function S_{ij} is also necessary for each edge $\bowtie_{ij}$ in the view graph.

In summary, the functions to be added to the view graph can be described as follows.

- $J_{ij}(\ldots)$ estimates the processing time that the view manager sends the left operand to data source R_j, evaluates the join condition $\bowtie_{ij}$ and returns the result to the view manager.

[4]There are other processing models, reader may refer to [12] for more detailed discussions.

- $J_{ji}(\ldots)$ estimates the processing time that the view manager sends the left operand to data source R_i, evaluates the join condition $\bowtie_{ij}$ and returns the result to the view manager.

- S_{ij} estimates the selectivity of the join operation $\bowtie_{ij}$ bewtween data sources R_i and R_j.

The input parameters to an edge cost function (J_{ij}) depend on the methods being used to develop the function for a specific environment. In our framework presented here, we use the cardinality and the number of attributes of operand tables. A linear regression technique will be applied [18] to build the cost functions for each join operation defined in the view graph. More details will be discussed in Section 3.4. Thus, the VG described in Figure 2 extended with appropriate cost functions is depicted in Figure 5.

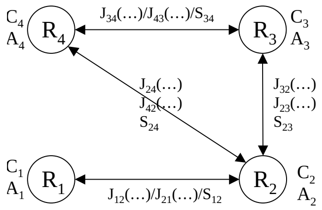

Figure 5: Extended View Definition Graph

3.2 Maintenance Strategy Cost

We propose the following computational model to calculate the query processing time of a maintenance strategy based on a view definition graph and its corresponding cost functions. As described in previous sections, we assume the view is defined upon n data sources $R_1, R_2, \ldots, R_n$ and has updates $\Delta R_1, \Delta R_2, \ldots, \Delta R_n$ need to be maintained. For simplicity, we use $|\Delta R_i|$ to represent the cardinality of the *delta change* ΔR_i, while we use $||\Delta R_i||$ to denote the number of attributes of ΔR_i ($1 \leq i \leq$ n). For a maintenance step i ($1 \leq i \leq$ n) in batch maintenance stategy which calculates the changes on the view extent due to a *delta change* ΔR_i, we assume $k_1, k_2, \ldots, k_n$ is a sequence that the view manager will use to maintain the ΔR_i with k_1 always equals i. Here, $1 \leq k_m \leq$ n, and $\forall k_m, k_s$ $m \neq s \iff k_m \neq k_s$ ($1 \leq m, s \leq$ n). That is, a sequence for ΔR_i is one of the permutations of $1..n$ with k_1 equals i. A k_m ($1 \leq m \leq$ n) in sequence means the view manager will visit the data source R_{k_m} at (m-1)th step. Given that, the batch maintenance strategy cost C_b can be calculated as listed below.

$$\begin{aligned} Attr_m^i &= ||\Delta R_i|| + \sum_{2 \leq s \leq m} C_{k_s} \\ Card_m^i &= |\Delta R_i| * \prod_{1 \leq s \leq m-1} S_{k_s, k_{s-1}} \end{aligned}$$

$$
\begin{aligned}
C_b^i &= \sum_{1 \le m \le n-1} J_{k_m, k_{m+1}}(Card_m^i, Attr_m^i) \\
C_b &= \sum_{1 \le i \le n} C_b^i
\end{aligned}
$$

$Attr_m^i$ records the total number of attributes at the mth maintenance query for maintenance step i. $Card_m^i$ denotes the cardinality of the mth query. C_b^i represents the cost of the maintenance step i. The batch maintenance strategy cost C_b is the sum of the cost of each maintenance steps.

For the grouping maintenance strategy, we assume $k_1, k_2, \ldots, k_n$ is the grouping path found in the view graph. Thus, the cost can be expressed by the following formulae.

$$
\begin{aligned}
T_{up} &= \sum_{i=1}^{n-1} J_{k_i, k_{i+1}}\left(\sum_{j=1}^{i} |\Delta R_{k_j}| * \prod_{j=1}^{i} S_{k_{s-1}, k_s}, \sum_{j=1}^{i} C_{k_j}\right) \\
T_{down} &= \sum_{i=n}^{2} J_{k_i, k_{i-1}}\left(\sum_{j=i}^{n} |\Delta R_{k_j}| * \prod_{j=i}^{n} S_{k_{i-1}, k_i}, \sum_{j=i}^{n} C_{k_j}\right) \\
T_{rm} &= Cost_r\left(\sum_{i=1}^{n} |\Delta R_{k_i}| * \prod_{i=1}^{n-1} S_{k_i, k_{i+1}}, \sum_{i=1}^{n} C_{k_i}\right) \\
T &= T_{up} + T_{down} + T_{rm}
\end{aligned}
$$

T_{up} calculates the cost of the *scroll up* phase, while T_{down} computes the *scroll down* phase cost. The function $Cost_r$ computes the cost of evaluating the remaining join conditions on the result of the grouping maintenance strategy. While the cost of grouping strategy T is the sum of these three parts.

3.3 Selecting View Maintenance Plans

As we described in the previous sections, each maintenance strategy may have multiple maintenance plans with a different processing time due to the differences in the properties such as the join conditions, network connection cost etc. How to find an efficient maintenance plan given a set of source updates is the key to optimizing the maintenance performance. For views to have both batch and grouping maintenance strategies available, we could choose the better one from the most efficient maintenance plans for each individual strategy. While for views that only have batch maintenance strategy, we select the most efficient one from all available batch maintenance plans.

Selecting Optimal Batch Maintenance Plan. One straightforward way to select the optimal batch maintenance plan is enumeration. For each maintenance step, we enumerate all the valid execution instances and estimate their correponding cost. Given each smallest maintenance step cost, the cost of the maintenance plan also reaches minimal.

Figure 7 illustrates a recursive algorithm that enumerates all the possible ways of evalulating a maintenance step which has a delta ΔR_i.

The algorithm is easy to understand by itself. The only thing that needs to be mentioned is in step 5. As we described in the batch view maintenance strategy in Sectoin 2, we prefer to combine all the join edges that have the same ending node while the start nodes of these edges have been visited (It is already in the Node_List). This is because such join conditions can be combined and evaluated at data source together instead of sending them individually. In a distributed environment, reducing the number of accesses to remote sources usually can result in a performance improvement. That is, this algorithm doesn't generate the valid execution instance that evaulates join conditions one by one. As an example, Figure 6 shows all the three possible execution orders for the maintenance step has delta change ΔR_4 for the view defined in Figure 2. The number on each edge indicates the execution order of join conditions. As mentioned earlier, we only enumerate the instances having one access to each source node. For simplicity, we use the sequence of the data source index to indicate the valid maintenance step instance. Thus, the sequences 4-3-2-1, 4-2-3-1 and 4-2-1-3 will be used to represent the three valid instances in Figure 6 respectively.

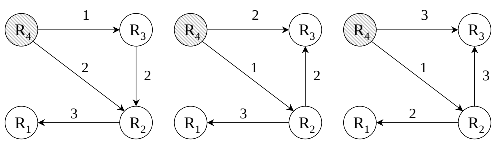

Figure 6: Maintenance Step 4 Enumerations

The cost of each maintenance step can be computed whenever a valid execution instance is found by the enumeration algorithm based on the computation model given in Section 3.2. For these edges that can be combined, we simply use the average cost of each individual estimated cost. For the combined join selectivity, the worst case will be the samllest one in all these individual estimations, while the best case will be the production of all these individual estimations (in case that all these join conditions are independent). Other cost estimation strategies can be applied for these join edges being combined. However, these low level changes on the cost estimation model will not change the search strategy itself, we can plug in any cost estimation function in general. As can be seen, the complexity of such enumeration algorithm can be exponential to the number of nodes in the view graph. It will be expensive for a large number of data sources.

However, the complexity of finding an optimal maintenance step execution instance is the same with the typical join sequence optimization problem, which is proven to be NP-hard [16] in general. Thus, no efficient algorithms have been found yet to solve such problem. A *Greedy Search Algorithm* can be built (in Figure 8) to leverage the optimization cost. Thus, we always select the most profitable edge from all the candidate edges as the next step to

```
Procedure EnumerateBatch()
 /*Enumerate all possible ways of evaluating a maintenance step has ΔRi  */
Input: Node_List //The nodes that have been visited
         Candidate_Edge //The edges that can be selected for next step
         Edge_Left //Edges that have not been processed
         Output_Sequence //The output sequence of evaluating edges
/* Initialization: Node_List only contains Ri, Candidate_Edge has the edges
       that starts from Ri, Edge_Left has all the edges except those in
       Candidate_Edge Output_Sequence is empty */
WHILE (Candidate_Edge != Empty) {
   1. Get an edge from Candidate_Edge and remove it;
   2. Put new nodes into Node_List or continue;
   3. Delete the edge from Edge_Left that has been removed in 1.;
   4. Check edges in Edge_Left that can be put into Candidate_Edge;
   5. Combine edges in Candidate_Edge if applicable;
   6. Call EnumerateBatch() recursively;
}
IF (sizeof (Node_List) equals # of graph nodes) {
    Calculate and output estimated cost;
}
```

Figure 7: Enumeration Algorithm

proceed. However, the greedy search can't guarantee the global optimal due to the cost of each edge will change dynamically based on previous steps.

```
Procedure GreedyAlgo()
 /*selecting most profitable edges at each step for ΔRi*/
Input: Node_List //The nodes that have been visited
         Candidate_Edge //The edges that can be selected for next step
         Edge_Left //Edges that have not been processed
         Output_Sequence //The output sequence of evaluating edges
/* Initialization: Node_List only contains Ri, Candidate_Edge has the edges
       that starts from Ri, Edge_Left has all the edges except those in
       Candidate_Edge Output_Sequence is empty */
WHILE (Candidate_Edge != Empty) {
   1. Get most profitable edge from Candidate_Edge and remove it;
   2. Put new nodes into Node_List or continue;
   3. Delete the edge from Edge_Left that has been removed in 1;
   4. Check edges in Edge_Left that can be put into Candidate_Edge;
   5. Combine edges in Candidate_Edge if applicable;
}
Output sequence and cost;
```

Figure 8: Greedy Search Algorithm

Selecting Optimal Grouping Plan. For views that have grouping maintenance strategy avaialable, the first step is to find the grouping path that goes through each node exactly once. However, it is a **Hamiltonian Path** problem, and known to be NP-Complete. Similar with that of selecting maintenance plans for the batch strategy, we can build a recursive algorithm to enumerate all the possible grouping paths in a view graph (if it is existed). The algorithm will be very simiar with that of in Figure 7. As an example, Figure 9 shows all possible grouping maintenance plans generated by the enumeration algorithm given the view defined in Figure 2. For simplicity, we also use source node index sequence to represent a grouping plan. For example, the sequence 1-2-3-4 denotes the first plan in Figure 9. The dashed line denotes the remaining edge need to be processed after the scroll up and scroll down phases.

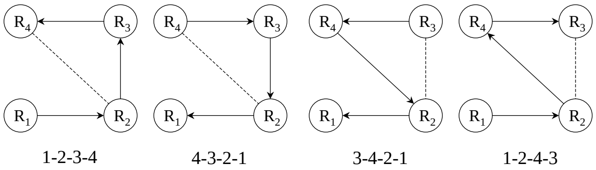

Figure 9: Grouping Plan Enumerations

Once we find a grouping path in the view graph, the total maintenance cost of its grouping plan can be calculated as we described in Section 3.2. The outout grouping maintenance plan will be the plan with the smallest estimated cost. Given that the number of nodes in a view graph is usually not large, such algorithms are still acceptable for most practical cases.

Verification vs. Optimization However, in a relatively stable environment, we may not have to search the complete space to get an optimal sequence whenever we get a new set of source updates that needs to be maintained. For example, if the selectivity S_{ij} are almost the same in the previous five runs, the cost function J_{ij} is close to linear and hasn't been changed since last run, then the optimal sequence found from previous runs will have a high chance to be the optimal sequence for the current set of updates. To reduce the optimization cost, we apply the following two strategies:

- Specify a threshold for the changes on cost function J_{ij} and S_{ij}, and always use the same sequence whenever the changes are below the threshold. For example, we will re-optimize the maintenance plan whenever there are more than 25% difference on selectivities between previous two runs or 20% of the cost functions have been changed compared with the last run.

- For the batch maintenance plans, we can compare the cost estimation on previous sequence using the current set of source updates with the estimated value from the greedy search algorithm. We optimize the sequence whenever the cost estimation on the previous sequence is worse than that of the greedy search.

However, such simplification may result in sub-optimal and some extra space and processing time are necessary for recording and comparing information on previous runs.

3.4 Cost Function Regression

The cost computation model we have developed in Section 3.2 is heavily dependent on the cost estimation function for each join condition (J_{ij}) in the view graph. In general, we can plug any cost estimation function into the cost computation model. In this work, we select the linear regression model [18] to develop the cost estimation function for each join edge defined in the view graph. The basic idea of the regression cost analysis is to derive cost model based on measured costs of several sample queries. A major benefit of using regression model is its local autonomy. That is, we may not have to know any details regarding the remote data sources to estimate the cost, which is convenient in a distributed environment. And in fact, it may simply not be possible to get internal information on a data source, thus externally observing its cost characteristics with probing queries is a practical solution.

As described in Section 3.1, a cost function J_{ij} will be used to estimate the cost that the view manager sends the current intermediate result to the data source R_j. There are two input parameters to a cost function J_{ij}, namely, $Card_i$ to represent the cardinality of the operand table and $Attr_i$ to denote its number of attributes. We propose the following basic formula to model the processing time of a maintenance query [5].

$$J_{ij} = B_0 + (B_1 + B_2 * Card_i) * Attr_i \tag{7}$$

This model is based on some existing cost models for a DBMS. The parameters B_0 can be intepreted as the initialization cost. $B_1 + B_2 * Card_i$ estimates the cost of processing all tuples in the delta table on the source relation, while $(B_1 + B_2 * Card_i) * Attr_i$ incorporates some effect of the number of attributes in each tuple on the total cost.

We have run several sample queries for each J_{ij} defined in view graph in our environment to measure the actual cost of queries on different inputs (a combination of different number of tuples and number of attributes). Based on the observed values and the basic cost model, we apply the linear regression least squares fit method to find the parameters B_0, B_1 and B_2 for each J_{ij} join condition.

4 Experiments

To verify the feasibility and effectness of our cost-driven view maintenance optimization framework, we have implemented the proposed strategies and the search algorithms based on our TxnWrap [3] system. We employ four data sources with one relation each as described in Figure 10 through a local network. Each relation has two attributes

[5] We can build other cost models for different join edges in the view graph, for simplicity but without loss of generality, we only describe one model to illustrate how it works.

and 100,000 tuples. As seen from Figure 10, different data sources have different processing capabilities due to the machine configurations. The materialized view defined over these four data sources can be expressed by the view graph in the upper conner in Figure 10. An index is built on DS_3 along the join condition between DS_2 and DS_3.

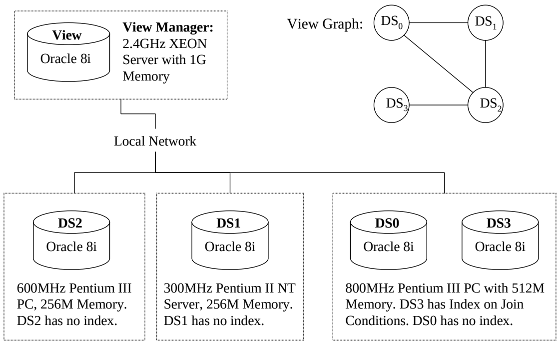

Figure 10: Experimental Environment

Cost Function Regression. Before we estimate the query processing time for a given maintenance strategy, we have to establish the cost function for each edge (join condition) defined in the view graph. Figure 11 shows the regression function we built for data source DS_0 along the join condition between DS_0 and DS_1 using least squares fit (its parameters are $B_0 = 755.37$, $B_1 = 16.04$ and $B_2 = 0.108$). The three solid lines from lower to higher record the measured query processing time (on the y-axis) for maintenance queries with 2, 4, 6 attributes respectively with the number of tuples in the maintenance query changing from 100 to 1000 (on the x-axis). Three dashed lines illustrate the estimated query processing cost given the input parameters (the number of attributes and the number of tuples). Seen from Figure 11, the cost function captures the basic trends of actual query processing costs. However, the lower the total processing cost, the more inaccurate the estimation in our model, this is because the smaller processing time can be more easily affected by random events.

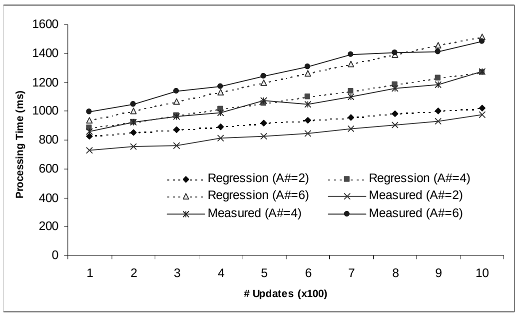

Figure 11: Cost Function Regression

Maintenance Plan Cost. Figures 12 and 13 show the cost differences between various maintenance plans. In Figure 12, the line 'Worst Measured' records the total maintenance query processing time measured using the batch maintenance plan generated (with the highest estimated query processing cost) by the search algorithm when maintaining source updates from 400 to 4000 (on the x-axis) [6]. While the line 'Best Measured' records the corresponding measured times for the best batch maintenance plan (with the lowest estimated cost). The two dashed lines in Figure 12 show the corresponding estimated total query processsing times for these two plans. The estimated cost reflects the measured cost trends fair accurately. However, it only accounts around 80% of the real cost in general for the following reasons: 1) The accumulated errors caused by each individual join cost function. 2) The measured cost includes extra processing time in the view manager such as converting the query results and composing maintenance queries which have not been incorporated into our cost model simplicity reasons.

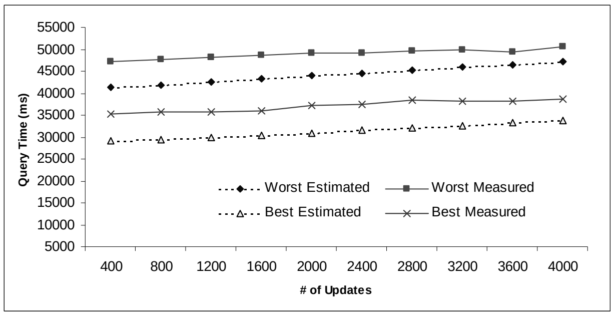

Figure 12: Cost of Batch Maintenance Plans

Figure 13 enumerates all four grouping maintenance plans available for the view definition (in Figure 10). Four solid lines record the measured costs and the dashed lines show the corresponding estimated costs. For the same reasons as those with the batch maintenance cost estimation, the estimated grouping maintenance plan cost is also below that of the real measured cost. However, it again keeps the trends well and thus support the usage of our proposed cost estimation approach.

Batching vs. Grouping. In the above two cases (Figures 12 and 13), we see that even the cost of the best batch maintenance plan is still worse than the worst grouping maintenance plan. This is because the grouping maintenance plans have a smaller number of accesses to the remote data sources. However, this is not true in general cases. To illustrate this, let assume that we have 2000 updates both in DS_0 and DS_1 respectively, and 10 updates in DS_2 and DS_3. We set up that each large delta will have an approximate 4.0 join factor which will return 4 times the number of input tuples as the result due to the join conditions between DS_0-DS_2 and DS_1-DS_2 respectively. While the join

[6]For simplicity, we assume all the updates are evenly distributed among these four data sourcs.

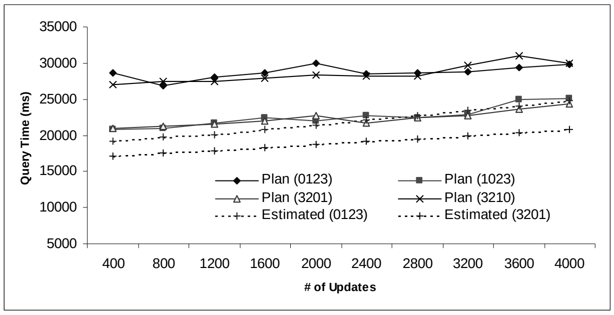

Figure 13: Cost of Group Maintenance Plans

factors of other edges is around 1.0.

Figure 14 illustrates the costs of five different batch maintenance plans given the above settings [7]. Figure 15 shows the costs of four grouping maintenance plans. Seen from these two figures, at least two batch maintenance plans are more efficient than even the best grouping maintenance plan. This is because in each grouping maintenance plan, we have to have one of the high join factor edges (either DS_0-DS_1 or DS_1-DS_2) in the grouping path. Thus, the intermediate result will be amplified by the high join factor of such join edge. For example, if we use the grouping path 0-1-2-3, then the second query of the scroll up phase may return 4000*4 tuples. However, in the batch maintenance plan, we always apply as many join conditions as possible in each maintenance query. Thus, the intermediate result of each maintenance query will be much smaller. In this case, a batch maintenance plan would have a large number of maintenance queries (in this example, it has 12 queries) yet will be still more efficient than a grouping maintenance plan (with 7 queries) because some large maintenance queries in the grouping plan overtake the benefits gained by the smaller number of maintenance queries.

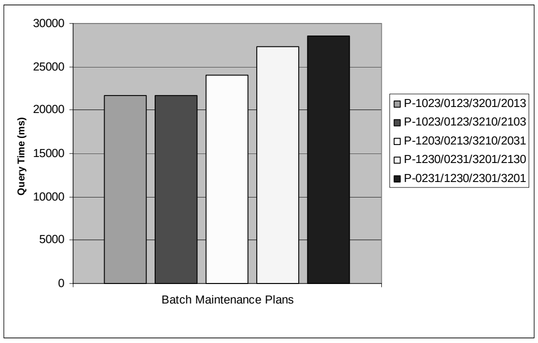

Figure 14: Batch Maintenance Plans

In the above experiments, the difference between maintenance plans is around 40% of the total cost. However, in

[7] A lot more other batch maintenance plans are available, here we only select a small part of them as examples.

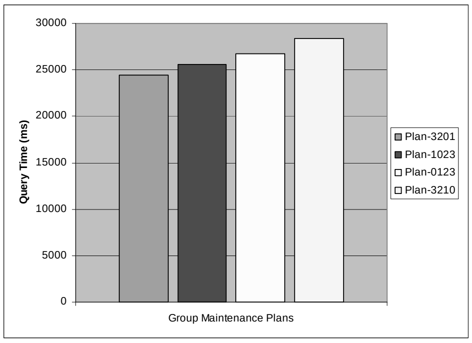

Figure 15: Group Maintenance Plans

a more diverse environment, the cost difference between maintenance plans may change dramatically. The following experiment illustrates the impact of the netwrok delay on the total maintenance cost.

Impact of the Network Cost. To evaluate the impact of different data transfer rates of the network, we insert the delay factors before evaluating each maintenance 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 τ, then it takes 100*2*τ to transfer one delta with 100 tuples with 4 attributes per tuple. Figure 16 shows the batch maintenance plan query time given τ equal to 10 (ms) while all the other settings are the same as in the experiment in Figure 14. Seen from Figure 16, the difference between the maintenance plans can be more than 100% of the total processing time. This is because the slower the network, the more the effect of processing extra intermediate query results due to a bad maintenance plans will become apparent.

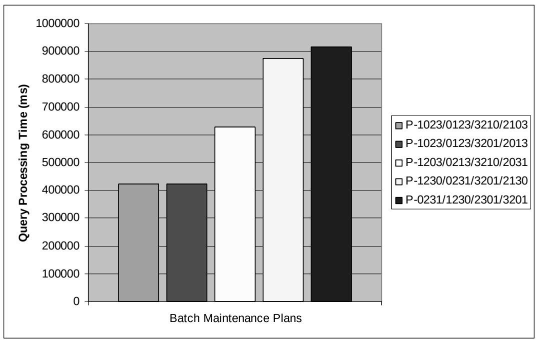

Figure 16: Batch Plans with Network Delay

Optimization Overhead. Due to the inherent complexity of the query optimization problem, there is no efficient algorithm available yet to find the optimal plan. The cost of the enumeration algorithm is small for a small number of data sources, a common situation for many views that integrate 10 or less data sources. For example, the batch enumeration algorithm for the four node view graph defined in Figure 10 takes less than 30 ms. While the grouping path search algorithm takes less than 15 ms to enumerate the paths. As expected, when the number of nodes in the view graph increases, the enumeration time also increases dramatically. A view graph with 9 nodes and 12 join edges needs 12,228 ms to enumerate using the batch enumeration algorithm in our environment. Given that the number of nodes in a view graph is usually not large, such enumeration algorithms are still acceptable for a large number of cases. In addition, a greedy search algorithm can be used to reduce the optimization time when needed. The algorithm described in Figure 8 for selecting batch maintenance plans takes 63 ms for the above 9 nodes 12 edges view graph. However, such reduction in optimization time is no longer guaranteed to find the optimal plan. Figure 17 shows result plan using the greedy search on the view defined in Figure 10. Thus, the greedy search maintenance plan has high estimated cost than the optimal one.

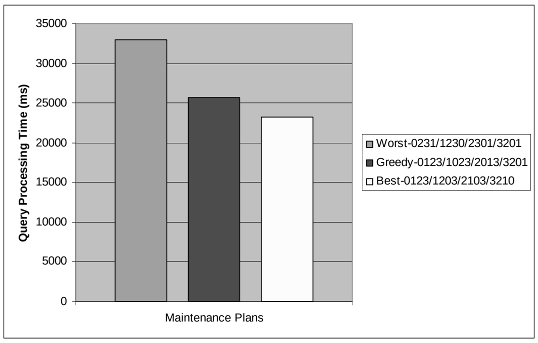

Figure 17: Greedy Search Estimated Cost

5 Related Work

Maintaining materialized views under source updates is one of the important issues in information integration and data warehousing [19]. A lot of algorithms have been proposed to maintain materialized views incrementally by issuing maintenance queries to the underlying data sources either using a compensation-based or a multiple version based approach [19, 20, 1, 3, 4].

In situations where the real time refresh of the view extent is not critical, changes to data sources can be buffered and propagated periodically to optimize the maintenance performance given a large number of updates. [14] proposed an asynchronous view maintenance algorithm using delta changes of data sources. [9] proposed a batch maintenance algorithm which can be applied to maintain a set of views. [10] introduced maintenance algorithms when both source data and schema changes are present. In our recent work [11], we explored the trade-off of the number of maintenance queries and the complexity of the query. There we have proposed the basic grouping maintenance strategy also employed by our current work. However, none of the above incorporates the dynamic nature of the real environment to further optimize the view maintenance process. Similar with [9], Posse [12] introduced a view maintenance optimization framework which only focuses on the order in which these delta changes are to be installed and maintained. While in our work, we explore the optimizations at an even lower level. That is, given a delta change, we study how to compose maintenance queries to calculate the maintenance result more efficiently.

Distributed query processing [8] is well studied also in particular for distributed environments. Garlic's optimizer [7] employs a rule-based approach which produces optimized plans using cost information provided by the corresponding wrappers. Various optimization techniques also have been proposed in the multiple query optimization work [15]. However, as we discussed earlier, more optimizations are available in a view maintenance framework due to the regularity of all (generated) maintenance queries that are dictated by the chosen maintenance strategy. Work on providing cost information and on developing cost models for data sources is also relevant. [18] proposed a regression algorithm to develop cost model for data sources. These works can be applied into our framework independently.

6 Conclusion

In this work, we explore two levels of trade-offs in the view maintenance strategies. That is, besides the inherent differences between maintenance strategies, such as the total number of maintenance queries vs. the complexity of each query, other factors such as the environmental settings including the network connection cost and the data source processing capabilities also will affect the view maintenance performance. We propose a cost-based view maintenance optimization framework by extending view maintenance strategies with corresponding cost models to generate efficient maintenance plans dynamically. The framework has been implemented in our TxnWrap view maintenance system. Experimental studies show that our framework generates efficient maintenance plans that optimize the view maintenance performance in terms of the total processing time significantly.

Although a lot of optimization techniques have been proposed both in distributed query processing and view maintenance areas, they were largely developed independently. Our work brings these two areas one step closer together and introduces a promising approach to further optimize the view maintenance performance given a non-homogenous environment. As one future step, a more adaptive view maintenance optimizer can be built by gathering cost information in a more timely manner.

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] L. M. Haas, D. Kossmann, E. L. Wimmers, and J. Yang. Optimizing Queries Across Diverse Data Sources. In *International Conference on Very Large Data Bases*, pages 276–285, 1997.

[8] D. Kossmann. The State of the Art in Distributed Query Processing. *ACM Computing Surveys (CSUR)*, 32(4):422–469, 2000.

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

[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, E. A. Rundensteiner, and D. Finkel. Optimizing View Maintenance Plans over Distributed Data Sources. In *Submission to CIKM'03*, Nov 2003.

[12] 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.

[13] D. Quass. Maintenance Expressions for Views with Aggregation. In *Proceedings of the Workshop on Materialized Views: Techniques and Applications*, June 1996.

[14] 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.

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

[16] C. Wang and M.-S. Chen. On the Complexity of Distributed Query Optimization. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 8(4):650–662, 1996.

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

[18] Q. Zhu, Y. Sun, and S. Motheramgari. Developing Cost Models with Qualitative Variables for Dynamic Multidatabase Environments. In *Proceedings of IEEE International Conference on Data Engineering*, pages 413–424, 2000.

[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.