A Comparison of Three Methods for Join View Maintenance in Parallel RDBMS

Gang Luo Jeffrey F. Naughton
Department of Computer Sciences
University of Wisconsin-Madison
1210 West Dayton Street
Madison, WI 53705
{gangluo, naughton}@cs.wisc.edu

Curt J. Ellmann Michael W. Watzke
NCR Advance Development Lab
5752 Tokay Boulevard, Suite 400
Madison, WI 53719
{Curt.Ellmann, Michael.Watzke}@ncr.com

Abstract

In a typical data warehouse, materialized views are used to speed up query execution. Upon updates to the base relations in the warehouse, these materialized views must also be maintained. The need to maintain these materialized views can have a negative impact on performance that is exacerbated in parallel RDBMSs, since simple single-node updates to base relations can give rise to expensive all-node operations for materialized view maintenance. In this paper, we present a comparison of three materialized join view maintenance methods in a parallel RDBMS, which we refer to as the naive, auxiliary relation, and global index methods. The last two methods improve performance at the cost of using more space. The results of this study show that the method of choice depends on the environment, in particular, the update activity on base relations and the amount of available storage space.

1. Introduction

Traditionally, data warehouses have been used to provide storage and analysis of large amounts of historical data. Recently, however, there has been a growing trend to use a data warehouse operationally, that is, to make real-time decisions about a corporation's day-to-day operations. This requires the database system to handle a mixed workload (adding real-time, online updates to a traditional data warehouse query workload.)

Most major RDBMS vendors have products and initiatives intended to address operational data warehousing, including Oracle's Oracle9i [6], NCR's active data warehouse [10], IBM's business intelligence system [1], Microsoft's digital nervous system [4], and Compaq's zero-latency enterprise [11]. This application of data warehouses raises a number of technical issues that are either not present, or are present to a lesser degree, in previous data warehouse applications. Of these, mixing updates with queries in the presence of materialized views is especially problematic.

For example, consider a parallel RDBMS with two base relations *A* and *B*, and an application in which there is a stream of updates to these relations. Suppose that each transaction updates one base relation and that each update is localized to one data server node. The throughput of the parallel RDBMS will be high. Now, however, suppose that in order to improve query performance, the DBA defines a materialized view over the join of *A* and *B*. As we will discuss in more detail in Section 2, even with no changes in the workload, the addition of this simple join view can bring what was a well-performing system to a crawl. This is because the addition of the join view converts what were simple single-node updates to expensive all-node operations. These all-node operations negate the throughput advantages of the parallel RDBMS, because instead of each node of the parallel RDBMS handling a fraction of the update stream, all nodes have to process every element of the update stream.

In this paper, we present three materialized join view maintenance methods in a parallel RDBMS: the naive method, the auxiliary relation method, and the global index method. The last two methods trade storage space for materialized view maintenance efficiency. That is, through the use of extra data structures, the auxiliary relation and global index methods reduce the expensive all-node operations that are required for materialized join view maintenance to single-node or few-node operations. In fact, the global index method of maintaining a join view is an "intermediate" method between the naive method and the auxiliary relation method:

(1) Global indices usually require less extra storage than auxiliary relations, while the naive method requires no extra storage.
(2) The global index method incurs less inter-node communication than the naive method, but more

inter-node communication than the auxiliary relation method.

(3) For each inserted (deleted, updated) tuple of a base relation, the join work needs to be done at (i) only one node for the auxiliary relation method, (ii) several nodes for the global index method, and (iii) all the nodes for the naive method.

Auxiliary relations have been considered previously in the context of distributed data warehouses [7]. There, they were used to make the materialized views "self-maintainable," that is, to ensure that updates at one distributed source could be propagated to the materialized view at the data warehouse without contacting other sources. Also, auxiliary relations are similar to copies of relations that are used to implement application specific partitioning in a parallel RDBMS. However, to the best of our knowledge, auxiliary relations have not been investigated in the literature as performance techniques for speeding materialized join view maintenance in a parallel RDBMS.

By "global index" we mean an index that maps from each value *x* in a non-partitioning attribute *c* of a relation to the global row ids of all the tuples that have value *x* in attribute *c*. Global indices are well known in practice and in the research literature [2]. Like auxiliary relations, to our knowledge the use of global indices for join view maintenance has not been discussed in the literature.

We investigate the performance of the three materialized join view maintenance methods with an analytical model. Also, we validate the analytical model for the naive and auxiliary relation methods with an implementation in a commercial parallel RDBMS.

The rest of this paper is organized as follows. We discuss join views and the three join view maintenance approaches in Section 2. Section 3 investigates the performance of the three join view maintenance methods. We conclude in Section 4.

2. Join View Maintenance Methods in a Parallel RDBMS

A join view stores and maintains the result from a join. We first consider join views on two base relations, and then we discuss join views on multiple base relations. In the remainder of this paper, we assume a parallel RDBMS with *L* data server nodes at which both base relations and materialized views are stored.

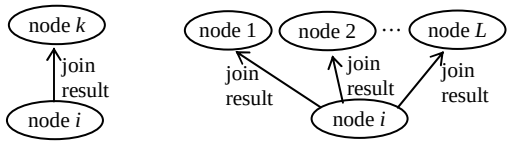
2.1. Join Views on Two Base Relations

Suppose that there are two base relations, *A* and *B*. An example of a join view *JV* for relations *A* and *B* on join attributes *A.c* and *B.d* is the following:

create join view *JV* as
select *
from *A*, *B*
where *A.c=B.d*
partitioned on *A.e*;

2.1.1. The Naive Maintenance Method. We begin by considering what happens if we propagate base relation updates using the obvious or "naive" approach in the absence of auxiliary relations. Consider how a join view *JV* is incrementally maintained when a tuple is inserted into a base relation in a parallel RDBMS. Assume that tuple T_A is inserted into base relation *A* at node *i*. Then to maintain *JV* we need to compute $T_A \bowtie B$, then insert the join result tuples into *JV*. Here are two cases to illustrate the disadvantages of the naive join view maintenance method:

Case 1: Suppose that the base relations *A* and *B* are partitioned on the join attributes *A.c* and *B.d*, respectively. Figures 1a and 1b show the procedure to maintain *JV*. Since base relation *B* is partitioned on the join attribute, tuple T_A only needs to be joined with the appropriate tuples of *B* at node *i*. If *JV* is partitioned on an attribute of *A*, the join result tuples (if any) are sent to some node *k* (node *k* might be the same as node *i*) to be inserted into *JV* based on the attribute value of T_A. If *JV* is not partitioned on an attribute of *A*, then the join result tuples need to be distributed to multiple nodes to be inserted into *JV*.

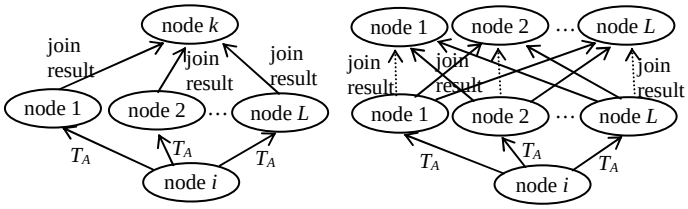


(a) *JV* is partitioned on an attribute of *A*

(b) *JV* is not partitioned on an attribute of *A*

Figure 1. Naive method of maintaining a join view (case 1).

Case 2: Suppose now that the base relations *A* and *B* are partitioned on the attributes *A.a* and *B.b*, which are not join attributes, respectively. Figures 2a and 2b show the procedure to maintain *JV*. The dashed lines represent cases in which the network communication is conceptual and no real network communication happens as the message is sent and received by the same node. Since base relation *B* is not partitioned on the join attribute, tuple T_A needs to be redistributed to every node to search for the matching tuples of *B* for the join, as we do not know at which nodes these matching tuples reside. If *JV* is partitioned on an attribute of *A*, the join result tuples (if any) are sent to some node *k* (node *k* might be the same as node *i*) to be inserted into *JV* based on the attribute value of T_A. If *JV* is not partitioned on an attribute of *A*, then the join result tuples need to be distributed to multiple nodes to be inserted into *JV*.



(a) *JV* is partitioned on an attribute of *A*

(b) *JV* is not partitioned on an attribute of *A*

Figure 2. Naive method of maintaining a join view (case 2).

In case 2, the naive method of maintaining a join view incurs substantial inter-node communication cost. Also, perhaps more importantly, a join needs to be done at every node, even though the base relation updates can be localized to a single node. We consider next how to eliminate both inefficiencies, especially the second one, by using auxiliary relations.

2.1.2. View Maintenance using Auxiliary Relations. We use auxiliary relations to overcome the shortcomings of the naive method of join view maintenance. In this section, we assume that neither base relation is partitioned on the join attribute. (If some base relation is partitioned on the join attribute, the auxiliary relation for that base relation is unnecessary.) In the parallel RDBMS, besides the base relations *A* and *B* and the join view *JV*, we maintain two auxiliary relations: AR_A for *A* and AR_B for *B*. Relation AR_A (AR_B) is a selection and projection of relation *A* (*B*) that is partitioned on the join attribute *A.c* (*B.d*). We maintain a clustered index I_A on *A.c* for AR_A (I_B on *B.d* for AR_B). Figure 3 shows the base relations, auxiliary relations, and join view at one node of the parallel RDBMS. For simplicity, we first assume that AR_A (AR_B) is a copy of relation *A* (*B*) that is partitioned on *A.c* (*B.d*), then we show how to minimize the storage overhead of auxiliary relations in Section 2.1.3.

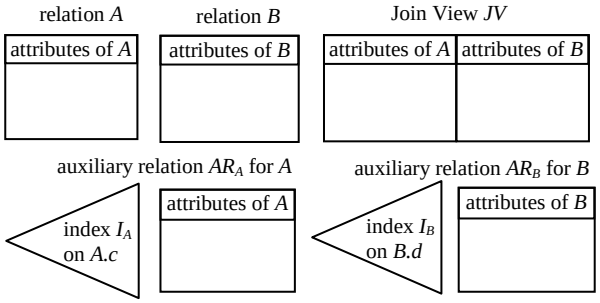
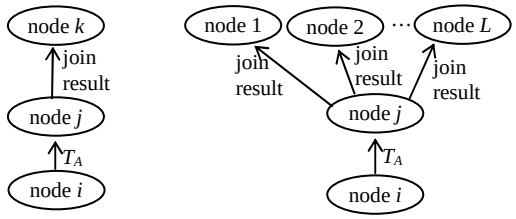


Figure 3. Base relations, auxiliary relations, and join view at a node of the parallel RDBMS.

When a tuple T_A is inserted into relation *A* at node *i*, it is also redistributed to some node *j* (node *j* might be the same as node *i*) based on its join attribute value $T_A.c$. Tuple T_A is inserted into the auxiliary relation AR_A at node *j*. Then T_A is joined with the appropriate tuples in the auxiliary relation AR_B (instead of base relation *B*) at node *j* utilizing the index I_B. If *JV* is partitioned on an attribute of *A*, the join result tuples (if any) are sent to some node *k* (node *k* might be the same as node *j*) to be inserted into *JV* based on the attribute value of T_A. If *JV* is not partitioned on an attribute of *A*, then the join result tuples need to be distributed to multiple nodes to be inserted into *JV*. Figures 4a and 4b show this procedure.



(a) *JV* is partitioned on an attribute of *A*

(b) *JV* is not partitioned on an attribute of *A*

Figure 4. Maintaining a join view using auxiliary relations.

The steps needed when a tuple T_A is deleted from or updated in the base relation *A* are similar to those needed in the case of insertion. Compared to the naive method, the auxiliary relation method of maintaining a join view has the following advantages:

(1) It saves substantial inter-node communication.

(2) For each inserted (deleted, updated) tuple of base relation *A*, the join work needs to be done at only one node rather than at every node.

In the naive method of maintaining a join view, the work needed when the base relation *A* is updated is as follows:

begin transaction
update base relation *A*;
update join view *JV*; (expensive)
end transaction.

For comparison, when we use the auxiliary relation method to maintain a join view, the work that needs to be done when the base relation *A* is updated is as follows:

begin transaction
update base relation *A*;
update auxiliary relation AR_A; (cheap)
update join view *JV*; (cheap)
end transaction.

If the update size is a small fraction of the base relation size, the extra work of updating the auxiliary relation AR_A is dominated by the advantages brought by the auxiliary relations in updating the join view *JV*.

In the above, we have considered the situation in which the base relation *A* is updated. The situation in which base relation *B* is updated is the same except we switch the roles of *A* and *B*.

2.1.3. Minimizing Storage Overhead. In the worst case, the auxiliary relation method requires substantial extra storage, as each auxiliary relation is a copy of some base relation. However, a more careful examination shows that the storage overhead required by the auxiliary relations can be reduced in many cases. As we have stated above, in [7], auxiliary views were proposed to make materialized views self-maintainable in a distributed data warehouse. [7] also proposed a systematic algorithm to minimize the storage overhead of auxiliary views. Their techniques for reducing storage overhead can be used in our auxiliary relation method. These techniques also apply to the global index method discussed below in Section 2.1.4. The main idea in [7] is not to include the unnecessary tuples and attributes of the base relations in the auxiliary relations. Applied to our scenario, an auxiliary relation is a selection and projection of a base relation that is partitioned in a special way. That is, an auxiliary relation AR_R of base relation R can be written as $AR_R = \pi(\sigma(R))$.

As an example, if join view *JV1* is defined as follows:

create join view *JV1* as
select *A.e, A.f, B.h*
from *A, B*
where *A.c=B.d*;

we only need to keep in the auxiliary relation AR_{A1} attributes *c*, *e*, and *f* of base relation *A*. If there is another join view *JV2* defined as follows:

create join view *JV2* as
select *A.e, A.g, C.p*
from *A, C*
where *A.c=C.q*;

we need to keep in the auxiliary relation AR_{A2} attributes *c*, *e*, and *g* of base relation *A*. However, there is redundancy between auxiliary relations AR_{A1} and AR_{A2}: both attributes *c* and *e* are stored in auxiliary relations AR_{A1} and AR_{A2}. If there are many join views defined on base relation *A* in the parallel RDBMS, the redundancy among those auxiliary relations of the same base relation *A* that are defined for different join views may be substantial. It is likely that the parallel RDBMS may not have enough disk space to store all of them. Also, when base relation *A* is updated, updating all the auxiliary relations of base relation *A* will be costly. One potential solution is to keep only one auxiliary relation AR_A for all the join views that use the same join attribute *A.c*, where AR_A is partitioned on the join attribute *A.c* and contains all the tuples and attributes of base relation *A*. In this case, auxiliary relation AR_A will require the same amount of storage as base relation *A*. If base relation *A* is very large (it may contain many attributes and many tuples), the parallel RDBMS still may not have enough disk space to store auxiliary relation AR_A.

2.1.4. View Maintenance Using Global Indices. The global index method for join view maintenance in general will require less space than the auxiliary relation method, although this space savings may come at the cost of some efficiency in processing join view maintenance. In this section, we assume that neither base relation is partitioned on the join attribute. (If some base relation is partitioned on the join attribute, the global index for that base relation is unnecessary.) In the parallel RDBMS, besides the base relations *A* and *B* and the join view *JV*, we maintain two global indices: GI_A for *A* and GI_B for *B*. Global index GI_A is an index on the join attribute *A.c*. It is partitioned on *A.c*. Each entry of the global index GI_A is of the form (value of *A.c*, list of global row ids), where the list of global row ids contains all the global row ids of the tuples of relation *A* whose attribute *A.c* is of that value. Each global row id is of the form (node id, local row id at the node). We define the global index GI_A to be distributed clustered (non-clustered) if the base relation *A* is clustered (non-clustered) on the join attribute *A.c* at each node. This technique applies to both base relation *A* and base relation *B*. Figure 5 shows the base relations, global indices, and join view at one node of the parallel RDBMS.

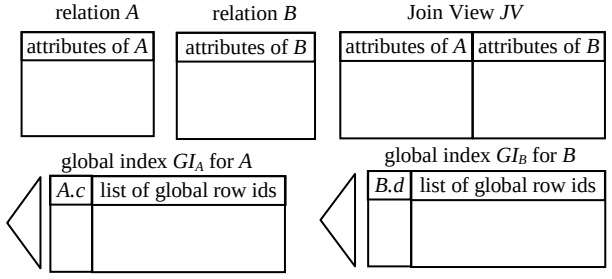
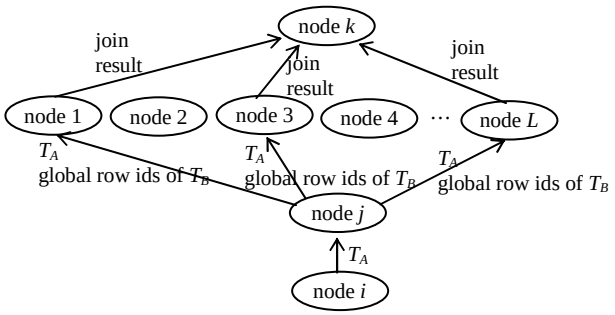
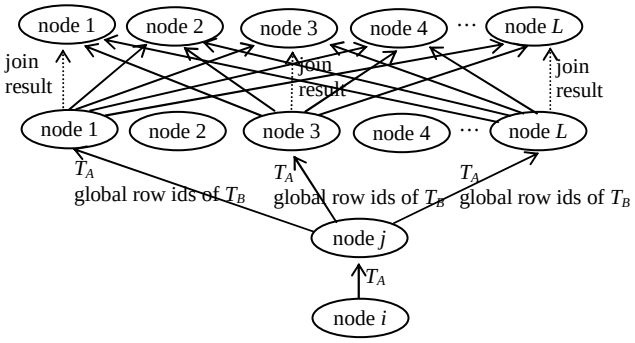


Figure 5. Base relations, global indices, and join view at a node of the parallel RDBMS.

When a tuple T_A is inserted into relation *A* at node *i*, it is also redistributed to some node *j* (node *j* might be the same as node *i*) based on its join attribute value $T_A.c$. A new entry containing the global row id of tuple T_A is inserted into the global index GI_A at node *j*. We search the global index GI_B at node *j* to find the list of global row ids for those tuples T_B of relation *B* that satisfy $T_B.d=T_A.c$. Suppose these tuples T_B reside at *K* of the *L* nodes. For each of the *K* nodes, T_A with the global row ids of those tuples T_B residing at that node is sent there. Then T_A is joined with those tuples T_B there. If *JV* is partitioned on an attribute of *A*, the join result tuples (if any) are sent to some node *k* (node *k* might be the same as node *j*) to be inserted into *JV* based on the attribute value of T_A. If *JV* is not partitioned on an attribute of *A*, then the join result tuples need to be distributed to multiple nodes to be inserted into *JV*. Figures 6a and 6b show the procedure.



(a) join view is partitioned on an attribute of *A*



(b) join view is not partitioned on an attribute of *A*

Figure 6. Maintaining a join view using global indices.

The steps needed when a tuple T_A is deleted from or updated in the base relation *A* are similar to those needed in the case of insertion. Thus, when we use the global index method to maintain a join view, the work that needs to be done when the base relation *A* is updated is as follows:

begin transaction
  update base relation *A*;
  update global index GI_A; (cheap)
  update join view *JV*; (moderate)
end transaction.

In the above, we have considered the situation in which the base relation *A* is updated. The situation in which base relation *B* is updated is the same except we switch the roles of *A* and *B*.

2.2. Extension to Multiple Base Relation Joins

Now we consider the situation that a join view is defined on more than two base relations. Suppose that a join view *JV* is defined on base relations R_1, R_2, …, and R_n. Then the auxiliary relation method works as follows:

For each base relation R_i $(1 \le i \le n)$
  For each base relation R_k that is joined with R_i in the join view definition
    Keep an auxiliary relation of R_i that is partitioned on the join attribute of $R_i \bowtie R_k$ unless R_i is partitioned on the join attribute of $R_i \bowtie R_k$.

When a base relation R_i $(1 \le i \le n)$ is updated, we do the following operations to maintain the join view:

(1) Update all the auxiliary relations of R_i accordingly.
(2) For each base relation R_j $(j \ne i, 1 \le j \le n)$
    Select a proper auxiliary relation of R_j (or R_j itself) based on the join conditions.
(3) Compute the changes to the join view according to the updates to R_i and the auxiliary (base) relation of R_j $(j \ne i, 1 \le j \le n)$ determined above.
(4) Update the join view.

The above algorithm also applies to the global index method.

The following is an example illustrating how this algorithm works. Consider a join view *JV* that is defined on $A \bowtie B \bowtie C$. For simplicity, we assume that no base relation is partitioned on the join attribute. (Again, if some base relation is partitioned on the join attribute, there is no need for an auxiliary relation on that base relation.) We keep the following auxiliary relations:

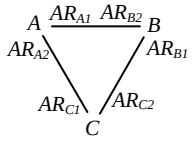
(1) AR_A for relation *A*, partitioned on the join attribute of $A \bowtie B$.
(2) AR_{B1} for relation *B*, partitioned on the join attribute of $A \bowtie B$.
(3) AR_{B2} for relation *B*, partitioned on the join attribute of $B \bowtie C$.
(4) AR_C for relation *C*, partitioned on the join attribute of $B \bowtie C$.

To maintain *JV* when some base relation is updated, we distinguish between three cases:

(1) If base relation *A* is updated, the same updates are propagated to the auxiliary relation AR_A. We use AR_{B1} and AR_C to maintain *JV*.
(2) If base relation *B* is updated, the same updates are propagated to the auxiliary relations AR_{B1} and AR_{B2}. We use AR_A and AR_C to maintain *JV*.
(3) If base relation *C* is updated, the same updates are propagated to the auxiliary relation AR_C. We use AR_{B2} and AR_A to maintain *JV*.

In the case of a join view defined on two base relations, the auxiliary relation method of maintaining join views is straightforward to implement using a query rewriting approach similar to [8]. However, if a join view is defined on multiple base relations, there are many choices as to how to use the auxiliary relations, and an optimization problem results. For example, consider a join view that is defined on the complete join of three base relations *A*, *B*, and *C*, where each base relation is joined to another on some join attribute. Assume that no base relation is partitioned on the join attribute. Then we need to keep the following auxiliary relations:

(1) AR_{A1} for relation A and AR_{B2} for relation B, both partitioned on the join attributes of $A \bowtie B$.
(2) AR_{B1} for relation B and AR_{C2} for relation C, both partitioned on the join attributes of $B \bowtie C$.
(3) AR_{C1} for relation C and AR_{A2} for relation A, both partitioned on the join attributes of $C \bowtie A$.



If a tuple T_A is inserted into the base relation A, there are four possible ways to compute the corresponding changes to the join view JV:
(1) T_A is joined with AR_{B2}, then the join result tuples are joined with AR_{C2}.
(2) T_A is joined with AR_{B2}, then the join result tuples are joined with AR_{C1}.
(3) T_A is joined with AR_{C1}, then the join result tuples are joined with AR_{B1}.
(4) T_A is joined with AR_{C1}, then the join result tuples are joined with AR_{B2}.

The optimization problem arises because it is impossible to state which alternative is best without considering relational statistics.

3. Performance of Different Join View Maintenance Methods

In this section we evaluate the performance of the three join view maintenance methods, first with an analytical model, and then with experiments in a commercial parallel RDBMS.

3.1. Analytical Model

We first propose a simple analytical model to gain insight into the performance advantage of the auxiliary relation / global index method vs. the naive method in maintaining materialized views. The goal of this model is not to accurately predict exact performance numbers in specific scenarios. Rather, it is to identify and explore some of the main trends that dominate in the auxiliary relation / global index approach. In Section 3.3 we show that our model for the naive and auxiliary relation methods predicts trends fairly accurately where it overlaps with our experiments with a commercial parallel RDBMS.

Consider a join view $JV=A \bowtie B$. We only analyze the case that the join view, JV, is partitioned on an attribute of relation A (the case in which the join view is partitioned on an attribute of B is symmetric.) Furthermore, we assume that neither the base relation A nor the base relation B is partitioned on the join attribute. We make the following simplifying assumptions in this model:
(1) Nodes i, j, and k are different from each other (Figures 2a, 4a, and 6a).
(2) Base relation A (B) has an index J_A (J_B) on the join attribute.
(3) The join view JV is partitioned on an attribute of relation A, and there is an index on this attribute.
(4) The network overhead of sending one message from one node to another node is a constant $SEND$, regardless of the message size and the network structure.
(5) In the auxiliary relation method, the overhead of searching the index once at each node is a constant $SEARCH$. If n ($n>0$) tuples T_B of base relation B are found to match a tuple T_A through index search at that node, the overhead of fetching these n tuples T_B and joining them with the tuple T_A is regarded as free. This is because the index on the join attribute of base relation B is clustered and these n tuples T_B are stored together in the index entry. (We are assuming that all n tuples fit on a single page. The model could be easily extended to capture cases where T_A joins with more tuples than fit on a single page; however, this would not change the conclusions that we draw from our model.)
(6) In the global index method, the overhead of searching the global index once at each node is a constant $SEARCH$. The overhead of fetching the entry of the global index is regarded as free. (Again, we are assuming that each entry of the global index fits on a single page.)
(7) In the naive method, the overhead of searching the index once at each node is a constant $SEARCH$. At one node, suppose n ($n>0$) tuples T_B of base relation B are found to match a tuple T_A through index search. Then the overhead of fetching these n tuples T_B and joining them with the tuple T_A is (i) $n \times FETCH$, if index J_B is non-clustered or (ii) regarded as free, if index J_B is clustered.
(8) The overhead of inserting a tuple into any table (base relation, auxiliary relation, global index, join view) is a constant $INSERT$.
(9) $|\Delta A|$ tuples are inserted, and these tuples are uniformly distributed on the join attribute.
(10) For each tuple T_A, N join result tuples are generated in total.
(11) For each tuple T_A, the matching tuples T_B of base relation B reside at K of the L nodes.
(12) The $|\Delta A|$ new tuples T_A are inserted into base relation A in a single transaction.

3.1.1. Total Workload. For each tuple T_A, we use as the cost metric the total workload TW, which we define to be the sum of the work done over all the nodes of the

parallel RDBMS. This is a useful basic metric because while other metrics, such as response time, can be derived from it, the reverse is not true (response time alone can hide the fact that multiple nodes may be doing unproductive work in parallel with the useful update operations.)

For any of the three join view maintenance methods (naive, auxiliary relation, and global index), the same updates must be performed on the base relations and on the join view. Because of this, in our model we omit the cost of these updates. Then the costs that must be captured are (a) the extra update of the auxiliary relation (global index) that is required by the auxiliary relation (global index) method, and (b) the differences among the three methods in the cost of the joins that are required to determine the result tuples that need to be inserted into the join view. We now turn to quantify those costs, which we refer to as *TW*.

(1) For the naive method, upon an insertion of a tuple T_A,

(a) Sending tuple T_A to each node has overhead $L \times SEND$.

(b) Joining tuple T_A with the appropriate tuples of base relation B at each node to generate all the N join result tuples has overhead (i) $L \times SEARCH + N \times FETCH$, if index J_B is non-clustered or (ii) $L \times SEARCH$, if index J_B is clustered. (Here again we are assuming that in the clustered index case, all the joining tuples are found on the leaf page reached at the end of the search.)

(c) The N join result tuples are generated at K of the L nodes. Sending these join result tuples to node k has overhead $K \times SEND$.

Thus for the naive method, the total workload *TW* for each tuple T_A is (i) $(L+K) \times SEND + L \times SEARCH + N \times FETCH$, if index J_B is non-clustered or (ii) $(L+K) \times SEND + L \times SEARCH$, if index J_B is clustered.

(2) For the auxiliary relation method,

(a) Sending tuple T_A to node j has overhead *SEND*.

(b) Inserting tuple T_A into auxiliary relation AR_A at node j has overhead *INSERT*.

(c) Joining tuple T_A with the appropriate tuples of base relation B at node j to generate all the N join result tuples has overhead *SEARCH*. (Again, we assume that because the index is clustered, the joining tuples are all found on the same leaf page reached by the *SEARCH*.)

(d) Sending the join result tuples from node j to node k has overhead *SEND*.

So for the auxiliary relation method, the total workload *TW* for each tuple T_A is $INSERT + 2 \times SEND + SEARCH$.

(3) For the global index method,

(a) Sending tuple T_A to node j has overhead *SEND*.

(b) Inserting a new entry for tuple T_A into global index GI_A at node j has overhead *INSERT*.

(c) Searching global index GI_B to find the list of global row ids has overhead *SEARCH*. Those tuples T_B of base relation B that correspond to these global row ids reside at K of the L nodes.

(d) Sending tuple T_A and the global row ids of those tuples T_B to the K nodes has overhead $K \times SEND$.

(e) At the K nodes, joining tuple T_A with those tuples T_B and generating the N join result tuples has overhead (i) $N \times FETCH$, if global index GI_B is distributed non-clustered or (ii) $K \times FETCH$, if global index GI_B is distributed clustered. Recall that if global index GI_B is distributed clustered, base relation B is clustered on the join attribute at each node. In this case, we assume that all the matching tuples T_B of base relation B reside at one page at each node. Note that if there are several global indices for the same base relation B, at most one global index can be distributed clustered as base relation B can be clustered for at most one attribute at each node. For example, for a join view JV' that is defined on $A \bowtie B \bowtie C$, global indices GI_{B1} (for $A \bowtie B$) and GI_{B2} (for $B \bowtie C$) cannot be both distributed clustered unless both joins $A \bowtie B$ and $B \bowtie C$ are on the same join attribute.

(f) Sending the join result tuples from the K nodes to node k has overhead $K \times SEND$.

For the global index method, the total workload *TW* for each tuple T_A is (i) $INSERT + (1+2 \times K) \times SEND + SEARCH + N \times FETCH$, if global index GI_B is distributed non-clustered or (ii) $INSERT + (1+2 \times K) \times SEND + SEARCH + K \times FETCH$, if global index GI_B is distributed clustered. Note that $K \le min(N,L)$.

Compared to the naive method, (1) the auxiliary relation method incurs an extra *INSERT*, while saving $(L+K-2)$ *SEND*s, $(L-1)$ *SEARCH*s, and N *FETCH*s (if index J_B is non-clustered); (2) the global index method incurs an extra *INSERT* and K *FETCH*s (if GI_B is distributed clustered), while saving $(L-K-1)$ *SEND*s and $(L-1)$ *SEARCH*s. As L grows, (1) for the auxiliary relation method, the savings in *SEND*, *SEARCH*, and *FETCH* are significant compared to the overhead of one extra *INSERT*; (2) for the global index method, the savings in *SEND* and *SEARCH* are significant compared to the overhead of one extra *INSERT* and K extra *FETCH*s (if GI_B is distributed clustered). In a typical parallel RDBMS, the time spent on *SEND* is much smaller than the time spent on *SEARCH*, *FETCH*, and *INSERT*. In the following, we only consider the time spent on *SEARCH*, *FETCH*, and *INSERT*. For simplicity, we will assume that *SEARCH* takes one I/O, *FETCH* takes one I/O, and

INSERT takes two I/Os. Our conclusions would remain unchanged by small variations in these assumptions.

3.1.2. Response Time. The model in Section 3.1.1 is accurate only if the join method is index nested loops, for which the cost is directly proportional to the number of tuples inserted. If $|\Delta A|$ is large enough, an algorithm such as sort merge may perform better than index nested loops. To explore this issue, we extend our model to handle this case. We use sort merge join as an alternative to index nested loops here; we believe our conclusions would be the same for hash joins. The point is that for both sort-merge and hash join, the join time is dominated by the time to scan a relation, and unless the number of modified tuples is a sizeable fraction of the base relations, the join time is independent of the number of modified tuples.

Let $\|x\|$ denote the size of x in pages. Let M denote the size of available memory in pages. In addition, we make the following simplifying assumptions:

(1) We use the number of page I/Os to measure the performance. Then the total workload *TW* for each tuple T_A is (i) 3 I/Os for the auxiliary relation method, (ii) $(L+N)$ I/Os for the naive method when index J_B is non-clustered, (iii) L I/Os for the naive method when index J_B is clustered, (iv) $(3+N)$ I/Os for the global index method when GI_B is distributed non-clustered, or (v) $(3+K)$ I/Os for the global index method when GI_B is distributed clustered.

(2) Tuples of relation B are evenly distributed both on the partitioning attribute and on the join attribute so that at each node i, the size of auxiliary relation AR_B in pages is equal to the size of relation B in pages. Both of them are denoted as $\|B_i\|=\|B\|/L$.

(3) ΔA_i can be held entirely in memory.

Given these assumptions, *TW* for the three methods for the multiple-tuple insertion is just $|\Delta A|$ times the *TW* for a single-tuple update. Calculating the response time is more interesting. We can express the response time (in number of I/Os) for each update method by considering the work that is done by each node in parallel.

(1) At each node i, for the naive method,
  (a) If the join method of choice is sort merge, then
    (i) if index J_B is non-clustered, the sort merge join time is dominated by the time of sorting B_i and is approximated by $\|B_i\| \times log_M \|B_i\|$ I/Os;
    (ii) if index J_B is clustered, the sort merge join time is dominated by the time of scanning B_i and is approximated by $\|B_i\|$ I/Os.
  (b) If the join method of choice is the index join algorithm, the index join time is approximated by $|\Delta A| \times (L+N)/L = |\Delta A_i| \times (L+N)$ I/Os (if index J_B is non-clustered) or $|\Delta A| \times L/L = |\Delta A_i| \times L$ I/Os (if index J_B is clustered).

(2) At each node i, for the auxiliary relation method,
  (a) If the sort merge join algorithm is the join method of choice, the sort merge join time is dominated by the time of scanning B_i and is approximated by $\|B_i\|$ I/Os, as auxiliary relation AR_B is clustered on the join attribute.
  (b) If index nested loops is the algorithm of choice, the index join time is approximated by $|\Delta A|/L = |\Delta A_i|$ I/Os.
  (c) The number of updates to the auxiliary relation is $|\Delta A|/L = |\Delta A_i|$.

(3) At each node i, for the global index method,
  (a) If the join method of choice is sort merge, then
    (i) if GI_B is distributed non-clustered, the sort merge join time is dominated by the time of sorting B_i and is approximated by $\|B_i\| \times log_M \|B_i\|$ I/Os;
    (ii) if GI_B is distributed clustered, the sort merge join time is dominated by the time of scanning B_i and is approximated by $\|B_i\|$ I/Os.
  (b) If the join method of choice is the index join algorithm, the index join time is approximated by $|\Delta A| \times (1+N)/L = |\Delta A_i| \times (1+N)$ I/Os (if GI_B is distributed non-clustered) or $|\Delta A| \times (1+K)/L = |\Delta A_i| \times (1+K)$ I/Os (if GI_B is distributed clustered).
  (c) The number of updates to the global index is $|\Delta A|/L = |\Delta A_i|$.

If $|\Delta A|$ is large enough that $\|B_i\| < |\Delta A_i|$, $\|B_i\| \times log_M \|B_i\| < |\Delta A_i| \times (L+N)$ (if index J_B is non-clustered), $\|B_i\| < |\Delta A_i| \times L$ (if index J_B is clustered), $\|B_i\| \times log_M \|B_i\| < |\Delta A_i| \times (1+N)$ I/Os (if GI_B is distributed non-clustered), and $\|B_i\| < |\Delta A_i| \times (1+K)$ (if GI_B is distributed clustered) are satisfied, then the sort merge join algorithm is preferable to index nested loops.

The above analysis shows that when sort-merge is the join algorithm of choice, the naive join view maintenance algorithm with clustered index actually outperforms the auxiliary relation / global index method. This is because each has the same join cost (the scan of B), while the auxiliary relation / global index method has the extra overhead of the updates to the auxiliary relation / global index. In the discussion of the experiments with the analytical model below, we discuss the implications of this fact when choosing a method for join view maintenance.

3.2. Experiments with Analytical Model

Setting $\|B\|=6{,}400$, $M=10$, $N=10$ (except in Figure 8), and $K=min(N,L)$, we present in Figures 7 ~ 12 the resulting performance of the auxiliary relation method, the global index method, and the naive method of join view maintenance. Figure 7 shows *TW* for a single tuple insert

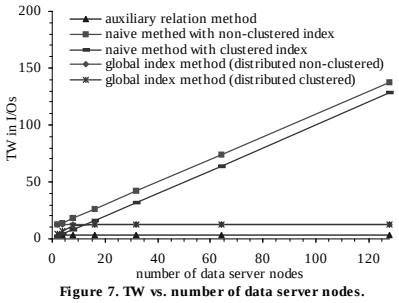


Figure 7. TW vs. number of data server nodes.

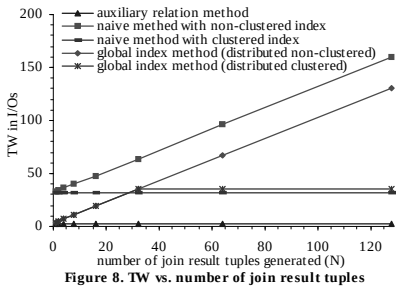


Figure 8. TW vs. number of join result tuples generated (L=32).

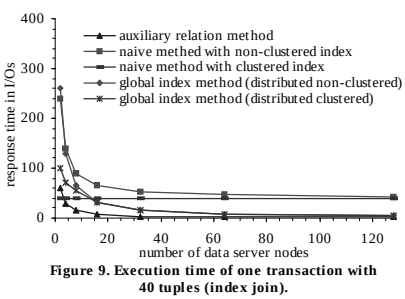


Figure 9. Execution time of one transaction with 40 tuples (index join).

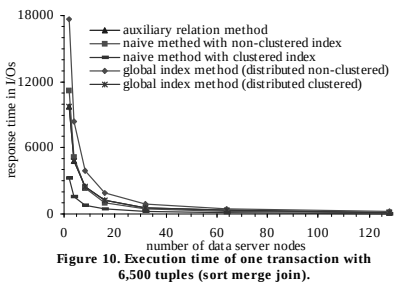


Figure 10. Execution time of one transaction with 6,500 tuples (sort merge join).

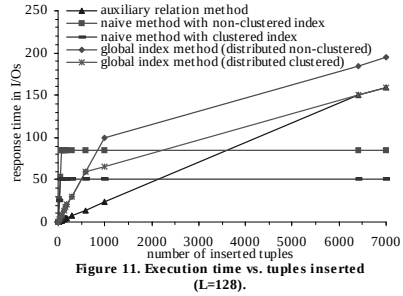


Figure 11. Execution time vs. tuples inserted (L=128).

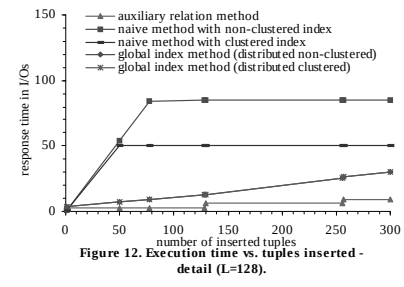


Figure 12. Execution time vs. tuples inserted - detail (L=128).

vs. the number of data server nodes. For the auxiliary relation method, *TW* is a small constant 3. For the naive method, *TW* increases linearly with the number of data server nodes. For the global index method, *TW* quickly reaches a constant 13 (*K* becomes *N* when *L* becomes larger than *N*), while this constant is greater than the constant for the auxiliary relation method.

Figure 8 shows *TW* for a single tuple insert vs. the number of join result tuples generated (*N*). When the number of join result tuples generated for the inserted tuple is small, *TW* for the global index method is close to *TW* for the auxiliary relation method. When the number of join result tuples generated for the inserted tuple is large, *TW* for the global index method is close to *TW* for the naive method. In other words, the global index method is an "intermediate" method between the naive method and the auxiliary relation method. When there are only a few matching tuples for a given join attribute value in the other base relation *B*, the overhead of the global index method is close to that of the auxiliary relation method. When there are many matching tuples for a given join attribute value in the other base relation *B*, the overhead of the global index method is close to that of the naive method.

Figure 9 shows the execution time of one transaction with 40 inserted tuples, where the join method of choice is the index join algorithm. The execution time of the auxiliary relation method ($3\times|\Delta A|/L$) decreases rapidly with more data server nodes. This is because in the auxiliary relation method, on average each node will see $|\Delta A|/L$ inserted tuples, whereas in the naive method, each node sees all $|\Delta A|$ inserted tuples. The execution time of the naive method ($|\Delta A|\times L/L=|\Delta A|$) is a constant when index J_B is clustered. Recall that this is because in our model, we assumed that in the clustered index case all joining tuples are found on the leaf page reached at the end of the *SEARCH* operation. When index J_B is non-clustered, the execution time of the naive method approaches that constant with more data server nodes ($|\Delta A|\times(L+N)/L$ approaches $|\Delta A|$ as *L* grows). The execution time of the global index method ($(3+K)\times|\Delta A|/L$ or $(3+N)\times|\Delta A|/L$) decreases rapidly with more data server nodes, while the decreasing rate is smaller than that of the auxiliary relation method. This is because in the global index method, on average each node will see $|\Delta A|\times K/L$ inserted tuples.

Figure 10 shows the execution time of one transaction with 6,500 inserted tuples, where the join method of choice is the sort merge join algorithm. Here we see that the naive method with a clustered index performs better than the auxiliary relation method. Also, the naive method performs better than the global index method. Note that there is nothing special about the number 6,500 other than that it is greater than the number of pages in base relation *B*. This indicates that if the expected update transaction inserts a number of tuples approximately equal to the number of pages in the base relation *B*, the naive method with clustered base relations is the method of choice.

It is an interesting empirical question whether or not such large update transactions are likely. Anecdotal evidence suggests that they are not – data warehouses typically store data from several years of operation, so it seems highly unlikely that individual update transactions

(of which there are presumably many each day) insert more than a very small fraction of the warehoused data. However, this is not something that can be proven by an abstract argument; rather, it must be decided on a case by case basis in the "real world."

Figure 11 shows the execution time of one transaction where the number of inserted tuples varies from 1 to 7,000. For the naive method, the execution time increases rapidly with the number of inserted tuples. For the auxiliary relation method and the global index method, the execution time increases much more slowly. The join time of any of the three methods reaches a constant when the number of inserted tuples is large enough for the sort merge join method to become the join method of choice. The global index method reaches this point much later than the naive method, and much earlier than the auxiliary relation method. This is due to the fact that in the auxiliary relation method and global index method, on average each node will see $|\Delta A|/L$ and $|\Delta A| \times K/L$ inserted tuples, respectively, whereas in the naive method, each node sees all $|\Delta A|$ tuples. However, once again, as the number of inserted tuples approaches the number of pages of B, the auxiliary relation (global index) method is indeed worse than the naive method.

Figure 12 "zooms in" on the execution time of one large transaction where the number of inserted tuples varies from 1 to 300. We notice that the execution time of the auxiliary relation method has a step-wise behavior. This is because the execution time of the auxiliary relation method depends on the maximum number of inserted tuples seen by each node. Assuming an even distribution, the maximum number of inserted tuples seen by each node for the auxiliary relation method is $\lceil \Delta A/L \rceil$, where $\lceil x \rceil$ is the ceiling function (e.g., $\lceil 1.3 \rceil$=2). For example, if $|\Delta A| \le L$, the maximum number of inserted tuples seen by each node is 1. If $L < |\Delta A| \le 2 \times L$, the maximum number of inserted tuples seen by each node is 2. The execution time of the global index method has a similar step-wise behavior that is not obvious on the Figure. This is due to the fact that assuming an even distribution, the maximum number of inserted tuples seen by each node for the global index method is $\lceil \Delta A \times K/L \rceil$.

It is straightforward to apply the above analytical model to the situation of a join view on multiple base relations. Experiments with this model did not provide any insight not already given by the two-relation model, so we omit them here.

3.3. Evaluation of the Auxiliary Relation Method in a Parallel RDBMS

We now turn to describe experiments we performed on NCR's Teradata Release V2R4 Version 4D. Our measurements were performed with the DBMS client application and server running on an Intel x86 Family 6 Model 5 Stepping 3 workstation with four 400MHz processors, 1GB main memory, eight 8GB disks, and running the Microsoft Windows NT 4.0 operating system. We allocated a processor and a disk for each data server, so there were at most four data servers on each workstation. We only tested the naive method and the auxiliary relation method for join view maintenance, as Teradata does not currently support the global index method.

The three relations used for the tests followed the schema of the standard TPC-R Benchmark relations [9]:

customer (custkey, acctbal, …),
orders (orderkey, custkey, totalprice, …),
lineitem (orderkey, partkey, suppkey, entendedprice, discount, …).

The underscore indicates the partitioning attributes of the relations. In our tests, each *customer* tuple matches one *orders* tuple on the attribute *custkey*. Each *orders* tuple matches 4 *lineitem* tuples on the attribute *orderkey*.

Table 1. Test data set I.

	number of tuples	total size
customer	0.15M	25MB
Orders	1.5M	178MB
Lineitem	6M	764MB

We wanted to test the performance of insertion into the *customer* relation in the presence of join views. We chose two join views for testing:

(1) *JV1* was the join result of *customer* and *orders* based on the join attribute *custkey*:

```
create join view JV1 as
select c.custkey, c.acctbal, o.orderkey, o.totalprice
from orders o, customer c
where c.custkey=o.custkey;
```

(2) *JV2* was the join result of *customer*, *orders*, and *lineitem* based on the join attributes *custkey* and *orderkey*.

```
create join view JV2 as
select c.custkey, c.acctbal, o.orderkey, o.totalprice,
       l.discount, l.extendedprice
from orders o, customer c, lineitem l
where c.custkey=o.custkey and
      o.orderkey=l.orderkey;
```

As the *customer* relation was partitioned on the join attribute, it required no auxiliary relation. The join view maintenance consists of three steps: updating the base relation, computing the changes to the join view, and updating the join view. As the first step and the third step were the same for the naive method and the auxiliary relation method, we only measured the time spent on the second step.

Because Teradata does not currently support the auxiliary relation maintenance method for join views, we used the following approach to gain insight into how it

would perform if implemented. We evaluated the performance of join view maintenance when 128 tuples were inserted into the *customer* relation (these tuples each have one matching tuple in the *orders* relation) in the following way:

(1) We created a non-clustered index on the *custkey* attribute of the *orders*, and another non-clustered index on the *orderkey* attribute of the *lineitem* relation.

(2) We created a new relation *delta_customer* that had the same schema and was partitioned in the same way as *customer*.

(3) We inserted 128 tuples into *delta_customer*.

(4) We created two relations *orders_1* and *lineitem_1* as auxiliary relations for *orders* and *lineitem*. They had the same schema and the same content as that of the relations *orders* and *lineitem*. The relation *orders_1* was partitioned on the *custkey* attribute, while *lineitem_1* was partitioned on the *orderkey* attribute. In Teradata, this means that a clustered index was automatically built on the *custkey* attribute of *orders_1*; similarly, Teradata automatically built a clustered index on the *orderkey* attribute of *lineitem_1*.

(5) We measured the execution time of the following two SQL statements:

```
select c.custkey, c.acctbal, o.orderkey, o.totalprice
from orders o, delta_customer c
where c.custkey=o.custkey;

select c.custkey, c.acctbal, o.orderkey, o.totalprice,
        l.discount, l.extendedprice
from orders o, delta_customer c, lineitem l
where c.custkey=o.custkey and
   o.orderkey=l.orderkey;
```

These two SQL statements implemented the naive method for maintaining join views *JV1* and *JV2*, respectively, while 128 tuples were inserted into the base relation *customer*. To implement the auxiliary relation method for maintaining join views *JV1* and *JV2*, we replaced *orders* and *lineitem* with *orders_1* and *lineitem_1*, respectively, in the two SQL statements.

We ran the SQL statements on 2-node, 4-node, and 8-node configurations, where each node was a data server. The 8-node configuration was the largest available hardware configuration.

The join view maintenance time predicted by the analytical model is shown in Figure 13. All the numbers in Figure 13 are scaled by a constant factor (the time unit is 128 I/Os) so only the relative ratios between them are meaningful. The experimental join view maintenance time is shown in Figure 14. Figures 13 and 14 match well. The speedup gained by the auxiliary relation (AR) method over the naive method for materialized view maintenance increases with the number of data server nodes.

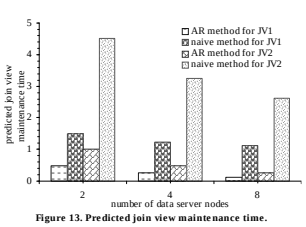


Figure 13. Predicted join view maintenance time.

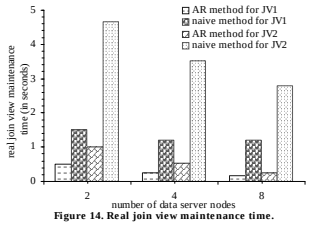


Figure 14. Real join view maintenance time.

We also ran experiments with large update transactions, where our analytical model predicts that the naive algorithm with clustered base relations performs well. Unfortunately, in the version of Teradata we tested, it was impossible to test the naive method with clustered indices, because clustered indices must be on partitioning attributes. We did indeed observe the trend that the performance of the naive and auxiliary relation methods became comparable; however, the analytical model was less accurate for large updates than for small. This is likely due to the impact of buffering throughout the system – with large insert transactions substantial fractions of the base and auxiliary relations end up getting cached in main memory. For these reasons we do not present the large update results here.

The difficulty of duplicating in Teradata the analytical model results for large updates does not affect our conclusions. The model is accurate for reasonably sized updates; these are the ones that are common in practice and also are the ones for which the auxiliary relation method dramatically outperforms the naive method.

4. Conclusion

This paper compares three join view maintenance methods in a parallel RDBMS: naive, auxiliary relation, and global index. We show through an analytical model that if the update size is small with respect to the base relation size, the auxiliary relation and global index methods can substantially improve efficiency by eliminating expensive all-node operations, replacing them with focused single-node or few-node operations. We also validate the analytical model for the naive and auxiliary relation methods through experiments with a commercial parallel RDBMS.

There are many factors that influence the performance of the three join view maintenance methods, e.g., the update activity on base relations and the amount of available storage space. For this reason, it is impossible to say that one method is always the best. In fact, for a given workload, it is complicated to decide which method is the best to use. Our analytical model could form the basis for a cost model that would enable a system to choose the best approach automatically.

Moreover, in many cases, it is possible that a hybrid method will outperform any of the three methods. For example, we could adopt the following heuristics:

(1) We only build auxiliary relations or global indices for those most frequently updated join views.
(2) We only build auxiliary relations or global indices for those join attributes that are shared by multiple join views.
(3) If there is enough storage space, we build auxiliary relations; otherwise we build global indices.
(4) If there are many join views defined on a base relation R where the same attribute $R.c$ is used as a join attribute, we build only one auxiliary relation AR_R or global index GI_R on $R.c$ containing all the tuples of base relation R. That is, we do not use the storage overhead saving techniques in [7] to build one auxiliary relation or global index on $R.c$ for each join view.

Making these heuristics more rigorous and a thorough evaluation of these hybrid strategies is an interesting area for future work.

Acknowledgements

We would like to thank Stephen Brobst, Grace Au, and Ambuj Shatdal for useful discussions, Patricia Bamrah for helpful comments on the manuscript. This work was supported by the NCR Corporation and also by NSF grants CDA-9623632 and ITR 0086002.

References

[1] The Road to Business Intelligence. http://www-4.ibm.com/software/data/busn-intel/road2bi.
[2] D. Choy, C. Mohan. Locking Protocols for Two-Tier Indexing of Partitioned Data. Proc. International Workshop on Advanced Transaction Models and Architectures, 1996.
[3] A. Gupta, I.S. Mumick. Materialized Views: Techniques, Implementations, and Applications. MIT Press, 1999.
[4] L. Grzanka. The Digital Nervous System and Beyond. http://www.enterprisebusiness.com/archives/vol1_issue1/cover.html.
[5] G. Klaus. Real-time Data Warehousing and Data Mining for E-Commerce. http://ids.csom.umn.edu/faculty/wanninger /lectures/DataMining-6204Sp00.html.
[6] Oracle Takes Aim At Real-Time Data Warehousing. http://www.informationweek.com/story/IWK20001120S0002.
[7] D. Quass, A. Gupta, I.S. Mumick, and J. Widom. Making Views Self-Maintainable for Data Warehousing. PDIS 1996: 158-169.
[8] D. Quass, J. Widom. On-Line Warehouse View Maintenance. SIGMOD Conf. 1997: 393-404.
[9] TPC Homepage. TPC-R benchmark, www.tpc.org.
[10] R. Winter. B2B Active Warehousing: Data on Demand. http://www.teradatareview.com/fall00/winter.html. Teradata Review, 2000.
[11] Compaq Zero Latency Enterprise Homepage. http://zle.himalaya.compaq.com/view.asp?PAGE=ZLE_HomeExt.