Auxiliary Relations 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

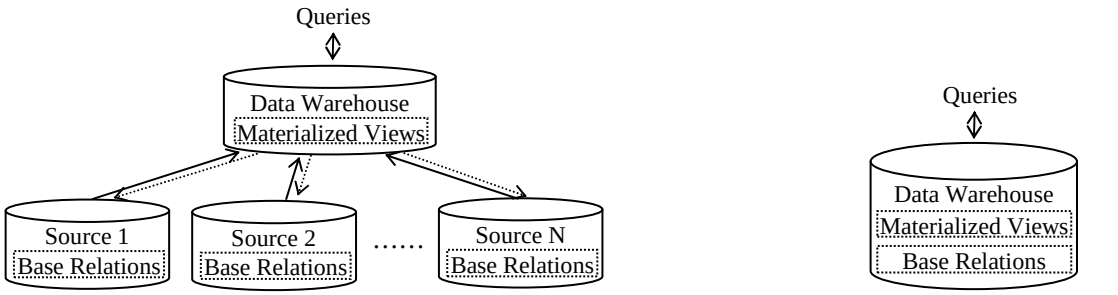
In a typical data warehouse, materialized views are used to speed up query execution. Upon updates to the base relations in the warehouse, these materialized views must also be updated. The need to update these materialized views can have a negative impact on performance. First, it requires materialized view updates in addition to the base relation updates, and second, it renders the materialized views unavailable for querying during the update window. This performance problem is exacerbated in parallel RDBMSs, where simple single-node updates to base relations can give rise to expensive all-node updates on the materialized views. In this paper, we investigate the use of auxiliary relations to speed up materialized join view maintenance in a parallel RDBMS. Both analytical results and an implementation of our approach in a commercial parallel RDBMS show that the use of auxiliary relations significantly decreases the system resources required for updates in the presence of materialized views, and that this benefit increases with the number of processors in the system.

1. Introduction

A data warehouse collects information from several, possibly widely distributed and loosely coupled, data sources. The collected information is integrated into a single database to be queried by the data warehouse clients. To provide acceptable query performance in the presence of complex queries over a large volume of data, a data warehouse is usually managed by a parallel RDBMS. In addition, a data warehouse usually makes use of many materialized views [CD97, HRU96]. There are two commonly used architectures for a data warehouse:

(1) Distributed Architecture: The base relations are stored at remote sources while the materialized views are stored at the centralized data warehouse repository.

(2) Centralized Architecture: Both base relations and materialized views are stored at the centralized data warehouse repository.



(a) distributed architecture (b) centralized architecture

Figure 1. Data Warehouse Architecture.

In a data warehouse, when base relations are updated, materialized views defined on the base relations must also be updated. These materialized views are typically maintained incrementally rather than recomputed from scratch in order to improve efficiency [GM99]. This incremental maintenance of materialized views degrades system performance in two ways. First, while these materialized views are being updated, they may not be available for querying. Second, updating these materialized views imposes additional demands on system resources above and beyond those required to simply update the base relations.

Recent trends in data warehousing make it imperative to reduce the burden imposed on the system by the update of materialized views. First, as the world moves to a 24×7 environment, there is no longer any "batch window" of downtime in which these updates can be hidden. Second, more and more businesses are finding it useful to run real-time [Kla] or "operational" data warehouses in which it is critical that the data in the warehouse be as up-to-date as possible. This second trend is reflected in most major RDBMS vendors' products and initiatives, including Oracle9i [Ora], NCR's active data warehouse [Win00], IBM's business intelligence system [BI], Microsoft's digital nervous system [Grz], and Compaq's zero-latency enterprise [ZLE].

In this paper we focus on an important class of materialized views called *join views*. Briefly, a join view on two relations *A* and *B* pre-computes a projection of the join of *A* and *B*, and facilitates queries based on this join. Updates to join views pose special challenges in parallel systems as it is likely that simple single-node updates to base relations give rise to expensive all-node operations in the presence of a join view.

We propose using auxiliary relations to speed materialized view maintenance for join queries in a parallel RDBMS. Conceptually, an auxiliary relation is a selection and projection of a base relation that is partitioned in a special way. It is, perhaps, counter-intuitive that adding still more relations, each of which must also be updated, can mitigate the update problem for materialized views. The reason adding more relations is effective is that through proper partitioning, auxiliary relations can be used to change expensive all-node update operations to single-node operations. While this may not improve the elapsed time for a given single-tuple update, in the context of multiple-tuple updates this can greatly reduce the total workload and thereby increase the performance of the system.

Auxiliary relations have been considered previously in the context of distributed data warehouses [QGM+96]. 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. To our knowledge, our present work is the first that considers the benefits of auxiliary relations in the context of a centralized data warehouse built upon a parallel RDBMS. Our results show that, unless the number of inserted tuples per transaction approaches the size in pages of the base relation, the auxiliary relation approach can substantially improve update performance.

2. Join Views in Parallel RDBMS

In this paper we consider join views in a parallel RDBMS with *L* data server nodes at which both base relations and materialized views are stored (the centralized data warehouse architecture). As we have

stated in the introduction, 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.

2.1. Join Views on Two Base Relations

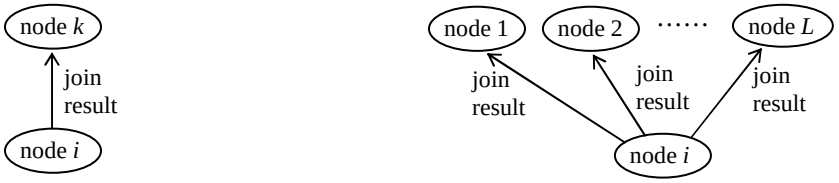
Suppose that there are two base relations, *A* and *B*. An example of a join view *JV* for relations *A* and *B* on join attributes *A.c* and *B.d* is the following:

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*. Here are two cases to illustrate the disadvantages of the naive join view maintenance method:

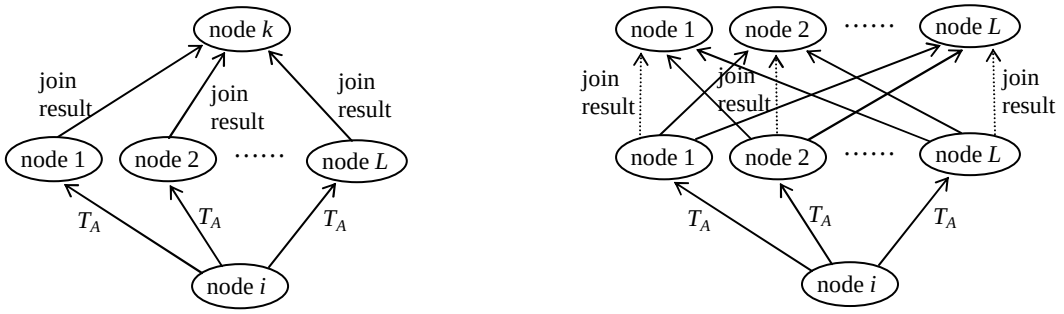
Case 1: Suppose that the base relations *A* and *B* are partitioned on the join attributes *A.c* and *B.d*, respectively. Figures 2a and 2b show the procedure to maintain *JV*. Tuple T_A is joined with the appropriate tuples of *B* at node *i*. If *JV* is partitioned on an attribute of *A*, the join 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) join view is partitioned on an attribute of *A* (b) join view is not partitioned on an attribute of *A*

Figure 2. 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 3a and 3b 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. Tuple T_A is redistributed to every node to search for the matching tuples of *B* for the join. 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) join view is partitioned on an attribute of *A* (b) join view is not partitioned on an attribute of *A*

Figure 3. 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. Without loss of generality, 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.) At each node, 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 copy 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 4 shows the base relations, auxiliary relations, and join view at one node of the parallel RDBMS.

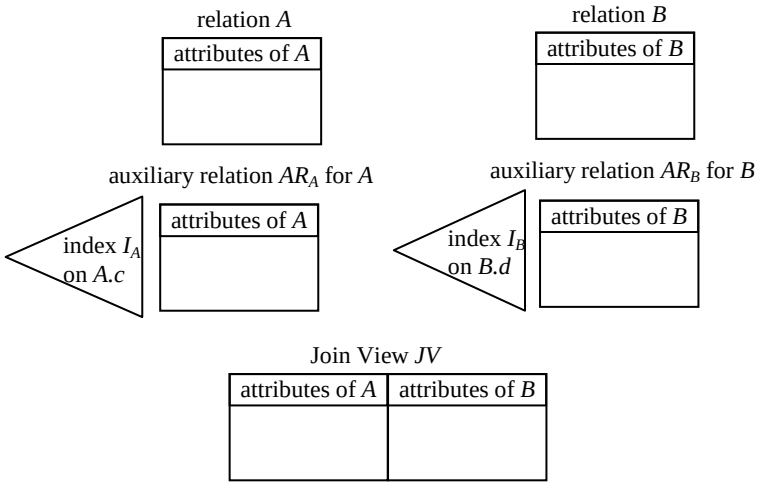
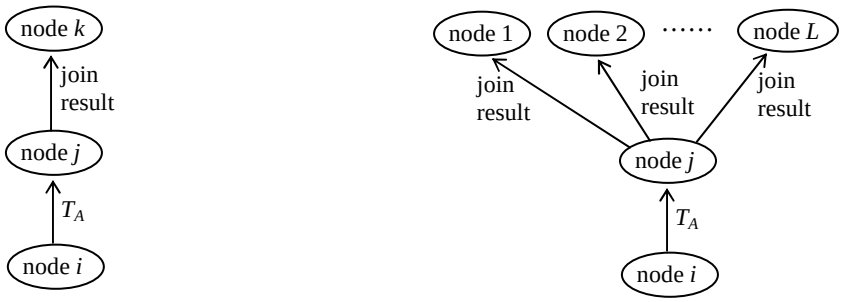


Figure 4. Base relations, auxiliary relations, and join view on a node of the parallel RDBMS.

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. 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 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 5a and 5b show this procedure.



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

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

Figure 5. 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.

These two points are perhaps the primary advantages of the auxiliary relation method. In the case of join views over joins on more than two relations, a third advantage can arise:

(3) The join work can be done utilizing the clustered index on the join attribute.

One might think that clustered indices could also be built and used for the naive method. While this is true for two-relation joins, as we will see in Section 2.2, this is not true for three-relation joins. For example, consider the join $A \bowtie B \bowtie C$. Here there will be two auxiliary relations for *B*, so one can be clustered for the join with *A*, the other for the join with *C*. In the naive method, there is no way that *B* can be simultaneously clustered for both joins (unless they are on the same attribute.)

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;
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;

update join view *JV*;

end transaction.

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*.

2.2. Extension to Multiple Base Relation Joins

Now we consider the situation that a join view is defined on multiple 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

We 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 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 [QW97]. 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 $C \bowtie A$.

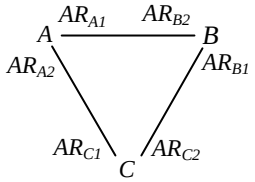


Figure 6. Auxiliary Relations for a join view defined on a complete join of three base relations.

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 the auxiliary relation AR_{B2}, then the join result tuples are joined with the auxiliary relation AR_{C2}.

(2) T_A is joined with the auxiliary relation AR_{B2}, then the join result tuples are joined with the auxiliary relation AR_{C1}.

(3) T_A is joined with the auxiliary relation AR_{C1}, then the join result tuples are joined with the auxiliary relation AR_{B1}.

(4) T_A is joined with the auxiliary relation AR_{C1}, then the join result tuples are joined with the auxiliary relation AR_{B2}.

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

2.3. 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 stated in the introduction, in [QGM+96], auxiliary views were proposed to make materialized views self-maintainable in a distributed data warehouse. [QGM+96] 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. Here is a brief review of the applicable techniques.

(1) If a join view has some selection condition on the base relation *A* in the "where" clause, such as:

create join view *JV* as
select *
from *A, B*
where *A.c=B.d* and *A.e=3*;

we only need to keep in the auxiliary relation AR_A those tuples of *A* that satisfy the selection condition. In the example above, we only need to keep in AR_A those tuples of *A* that satisfy *A.e=3*.

(2) If a join view does not contain all attributes of the base relation *A*, such as:

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

we only need to keep in the auxiliary relation AR_A those attributes of *A* that are necessary, e.g., the join attribute and the attributes appearing in the "select" clause. We can exclude the other unnecessary attributes. In the example above, we only need to keep in AR_A the attributes *c* and *e* of base relation *A*.

(3) Suppose that the join view is a key – foreign-key join, such as:

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

where *A.c* is a key of relation *A* and *B.d* is a foreign key of relation *B* that references *A.c*.

If a tuple T_A is inserted into relation *A*, there can be no matching tuple in relation *B* (or else that tuple would have violated the key-foreign key constraint before T_A was inserted). However, if a tuple T_B is inserted into relation *B*, there must be a matching tuple of relation *A* (or else relation *B* will violate the constraint after insertion). The case for deletion is similar. Thus, if we only consider insertion and deletion, we only need to keep the auxiliary relation AR_A and there will be no need for the auxiliary relation AR_B. However, if a tuple T_A of relation *A* is modified on a non-join attribute, to maintain the join

view *JV*, tuple T_A has to be joined with the appropriate tuples in the base relation *B*. In this case, the auxiliary relation AR_B is helpful. Thus there is a tradeoff:

(1) If we do not keep the auxiliary relation AR_B, we can save the storage overhead of AR_B. However, maintaining the join view *JV* will be costly in the case of updates to the base relation *A*.

(2) If we keep the auxiliary relation AR_B, we have the storage overhead of AR_B, but maintaining the join view *JV* will be efficient.

In a data warehousing environment, it is common for certain base relations to be infrequently modified, with virtually all updates being inserts or deletes. An example of this is the dimension tables of a star schema. In such a situation we can use the key-foreign key constraint analysis to eliminate the need for an auxiliary relation on the fact table. The space savings here could be substantial, since the dimension tables are typically a small fraction of the size of the fact table.

3. An Analytic Performance Model

In this section we propose a simple analytical model to gain insight into the performance advantage of the auxiliary relation 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 approach. In Section 4 we show that our model predicts trends fairly accurately where it overlaps with our experiments with a commercial parallel RDBMS.

Consider a join view $JV=A \bowtie B$. For simplicity and without loss of generality, 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 3a and 5a).

(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 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.

(7) The overhead of inserting a tuple into any table (base relation, auxiliary relation, join view) is a constant *INSERT*.

(8) $|\Delta A|$ tuples are inserted, and these tuples are uniformly distributed on the join attribute.

(9) For each tuple T_A, N join result tuples are generated in total.

(10) For each tuple T_A, the matching tuples T_B of base relation B reside at K of the L nodes.

(11) The $|\Delta A|$ new tuples T_A are inserted into base relation A in a single transaction.

This last restriction allows us to ignore the effect of lock contention in join view updates. Our results in this paper can be regarded as showing the performance that results if updates are grouped into batches.

Other researchers, for example [QW97], have proposed techniques to maintain two versions of the database such that one version is used for reader queries while another version is used for maintenance. In such a scenario batched updates is a reasonable assumption.

It is not obvious, however, how the relative performance of the naive and auxiliary relation methods would change if updates were done instead as many concurrent, single tuple transactions. On one hand, the auxiliary relation method converts all-node operations to single-node operations, which tends to increase concurrency; on the other hand, concurrency control conflicts may serialize some transactions, resulting in worse performance than that predicted by our model for both view maintenance methods. Exploring this tradeoff, and specialized locking schemes for this problem, is an interesting topic for future work.

Returning to our model, 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 irrelevant unproductive work in parallel with the useful update operations.)

For either join view maintenance method (naive or auxiliary relation), 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 that is required by the auxiliary relation method, and (b) the differences between the two 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*.

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

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

(2) 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.)

(3) 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.

For the auxiliary relation method,

(1) Sending tuple T_A to node j has overhead $SEND$.

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

(3) 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 $SEARCH$.)

(4) 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 $INSERT + 2 \times SEND + SEARCH$.

Compared to the naive method, 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). As L grows, the savings in $SEND$, $SEARCH$ and $FETCH$ are significant compared to the overhead of one extra $INSERT$. 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.

Note, however, that this model 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, or (iii) L I/Os for the naive method when index J_B is 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 both 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 $|\Delta A|/L = |\Delta A_i|$.

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), and $||B_i|| < |\Delta A_i| \times L$ (if index J_B is 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 method. This is because both have the same join cost (the scan of *B*), while the auxiliary relation has the extra overhead of the updates to the auxiliary relation. In the discussion of the experiments with the analytic model below, we discuss the implications of this fact when choosing a method for join view maintenance.

Experiments with Analytic Model

Setting $||B||=6{,}400$, $M=10$, and $N=10$, we present in Figures 7 ~ 11 the resulting performance of the auxiliary relation method and the naive method of join view maintenance. Figure 7 shows *TW* for a single tuple insert vs. the number of data server nodes. For the auxiliary relation method, *TW* is a constant. For the naive method, *TW* increases linearly with the number of data server nodes.

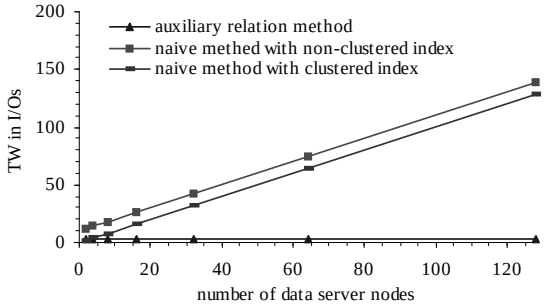


Figure 7. TW vs. number of data server nodes.

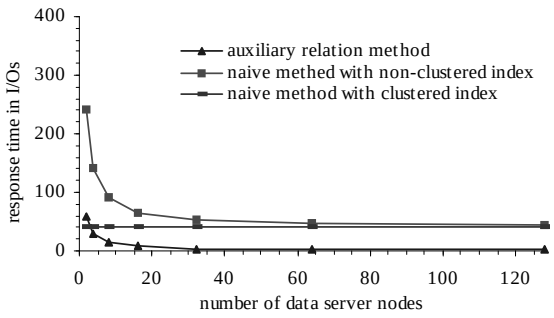


Figure 8. Execution time of one transaction with 40 tuples (index join).

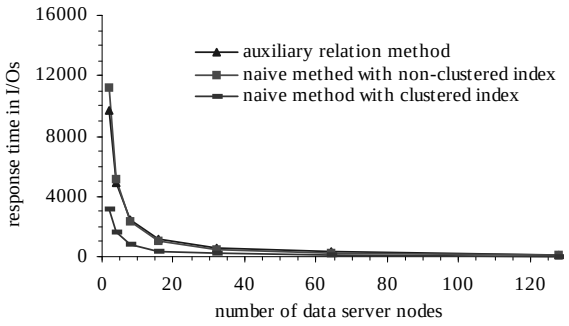


Figure 9. Execution time of one transaction with 6,500 tuples (sort merge join).

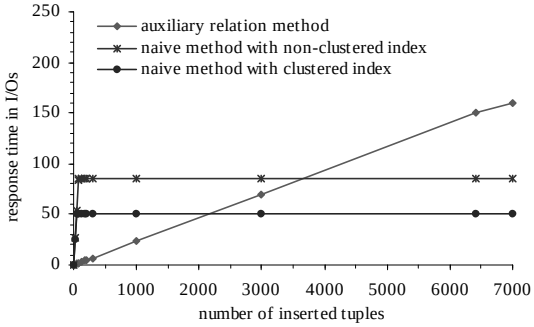


Figure 10. Execution time vs. tuples inserted (L=128).

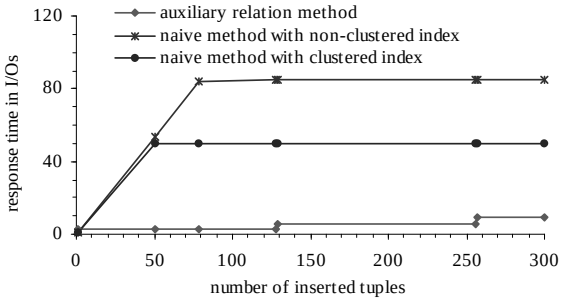


Figure 11. Execution time vs. tuples inserted - detail (L=128).

Figure 8 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).

Figure 9 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. 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 10 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, the execution time increases much more slowly. The join time of each

method 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 auxiliary relation method reaches this point much later than the naive method, 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 $|\Delta A|$ tuples. However, once again, as the number of inserted tuples approaches the number of pages of B, the auxiliary relation method is indeed worse than the naive method.

Figure 11 "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 $|\Delta A| \leq L$, the maximum number of inserted tuples seen by each node is 1. If $L < |\Delta A| \leq 2 \times L$, the maximum number of inserted tuples seen by each node is 2.

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.

4. Evaluation in the Teradata Parallel RDBMS

In this section, we investigate the performance of the auxiliary relation method for materialized view maintenance in NCR's Teradata Release V2R4 Version 4D. Our measurements were performed with the Teradata client application and server running on an Intel x86 Family 6 Model 5 Stepping 3 workstation with four 400MHz processors, 1GB main memory, six 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.

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

customer (custkey, name, address, nationkey, phone, acctbal, mktsegment, comment),
orders (orderkey, custkey, orderstatus, totalprice, orderdate, orderpriority, clerk, shippriority, comment),
lineitem (orderkey, partkey, suppkey, linenumber, quantity, entendedprice, discount, tax, returnflag, linestatus, shipdate, commitdate, receiptdate, shipinstruct, shipmode, comment).

The underscore indicates the partitioning attributes of the relations. In our tests, each *customer* tuple matches 1 *orders* tuple on the attribute *custkey*. Each *orders* tuple matches 4 *lineitem* tuples on the attribute *orderkey*.

	number of tuples	total size
customer	0.15M	25MB
orders	1.5M	178MB
lineitem	6M	764MB

Table 1. Test data set.

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 update method for join views, we used the following approach to see 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*, while *lineitem_1* was partitioned on *orderkey*. 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 predicated by the analytical model is shown in Figure 12. All the numbers in Figure 12 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 13. Figures 12 and 13 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.

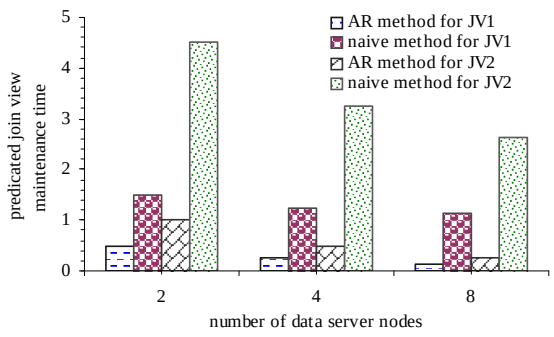


Figure 12. Predicated join view maintenance time.

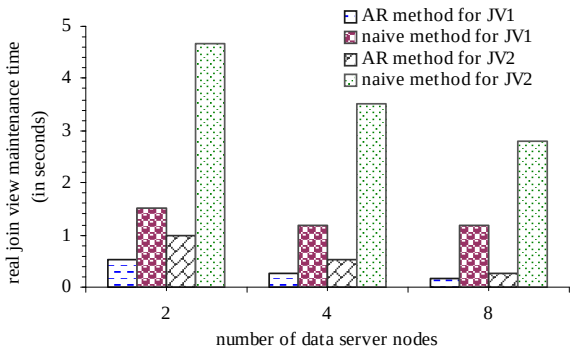


Figure 13. Real join view maintenance time.

We also ran experiments with large update transactions, where our analytic 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 analytic 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 and due to space constraints we do not present the large update results here.

The difficulty of duplicating in Teradata the analytic 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.

5. Conclusion

This paper proposes the use of auxiliary relations to speed materialized join view maintenance in a parallel RDBMS. We show, through an analytic model and through experiments with a commercial parallel RDBMS, that this approach can substantially improve efficiency by eliminating redundant all-node operations, replacing them with focused single-node operations. We show that this leads to superior performance for the auxiliary relation method unless the number of inserted tuples grows approximately equal to the size in pages of the base relations. There are a number of interesting problems remaining for future work.

As one example, it would be interesting to consider how auxiliary relations can be used to improve maintenance times for other kinds of materialized views such as aggregate views. There has been substantial work (e.g., [LQA97, RSS96]) on the use of complex materialized views to speed the maintenance of other complex materialized views. Previous work in this direction has considered only a uni-processor environment so that the effects of data partitioning (the main focus of our work) were not considered; it is possible that throwing a consideration of partitioning into the mix could create the need for new techniques.

Finally, although the experiments in this paper accurately model situations in which updates to the warehouse are grouped into batches, additional issues arise if the updates are instead submitted as many small, concurrent transactions. As we mentioned in Section 3, there is an interesting tradeoff here, since for single tuple update transactions, the auxiliary relation approach replaces all-node operations by single-node operations, which suggests that it will increase concurrency over the naive method. On the other hand, concurrency control conflicts will serialize some of the update transactions, thus perhaps giving a smaller speedup over the naive approach than that predicted by our model, which assumed that all the updates were gathered into a single transaction.

Acknowledgements

We would like to thank Grace Au for providing the implementation details of join view in Teradata, Patricia Bamrah for helpful comments on the manuscript.

References

[AAS+97] D. Agrawal, A.E. Abbadi, A.K. Singh, and T. Yurek. Efficient View Maintenance at Data Warehouses. SIGMOD Conf. 1997: 417-427.

[BCL89] J.A. Blakeley, N. Coburn, and P. Larson. Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates. TODS 14(3): 369-400, 1989.

[BI] The Road to Business Intelligence. http://www-4.ibm.com/software/data/busn-intel/road2bi.

[CD97] S. Chaudhuri, U. Dayal. An Overview of Data Warehousing and OLAP Technology. SIGMOD Record 26(1): 65-74, 1997.

[GM99] A. Gupta, I.S. Mumick. Materialized Views: Techniques, Implementations, and Applications. MIT Press, 1999.

[Grz] L. Grzanka. The Digital Nervous System and Beyond. http://www.enterprisebusiness.com/archives/vol1_issue1/cover.html.

[HRU96] V. Harinarayan, A. Rajaraman, and J.D. Ullman. Implementing Data Cubes Efficiently. SIGMOD Conf. 1996: 205-216.

[JMS95] H.V. Jagadish, I.S. Mumick, and A. Silberschatz. View Maintenance Issues for the Chronicle Data Model. PODS 1995: 113-124.

[Kla] G. Klaus. Real-time Data Warehousing and Data Mining for E-Commerce. http://ids.csom.umn.edu/faculty/wanninger/lectures/DataMining-6204Sp00.html.

[LQA97] W. Labio, D. Quass, and B. Adelberg. Physical Database Design for Data Warehouses. ICDE 1997: 277-288.

[Ora] Oracle Takes Aim At Real-Time Data Warehousing. http://www.informationweek.com/story/IWK20001120S0002.

[QGM+96] D. Quass, A. Gupta, I.S. Mumick, and J. Widom. Making Views Self-Maintainable for Data Warehousing. PDIS 1996: 158-169.

[QW97] D. Quass, J. Widom. On-Line Warehouse View Maintenance. SIGMOD Conf. 1997: 393-404.

[RSS96] K.A. Ross, D. Srivastava, and S. Sudarshan. Materialized View Maintenance and Integrity Constraint Checking: Trading Space for Time. SIGMOD Conf. 1996: 447-458.

[TPC] TPC Homepage. TPC-R benchmark, www.tpc.org.

[Val87] P. Valduriez. Join Indices. TODS 12(2): 218-246, 1987.

[Wid95] J. Widom. Research Problems in Data Warehousing. CIKM 1995: 25-30.

[Win00] R. Winter. B2B Active Warehousing: Data on Demand. http://www.teradatareview.com/fall00/winter.html. Teradata Review, 2000.

[ZLE] Compaq Zero Latency Enterprise Homepage. http://zle.himalaya.compaq.com/view.asp?PAGE=ZLE_HomeExt.