TECHNIQUES FOR OPERATIONAL DATA WAREHOUSING

by

Gang Luo

A dissertation submitted in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN – MADISON

2004

Abstract

Traditionally, data warehouses have been used to analyze historical data. Recently, there has been a growing trend to use data warehouses to support real-time decision-making about an enterprise's day-to-day operations. The needs for improved query and update performance are two challenges that arise from this new application of a data warehouse. To address these needs, new data warehouse functionality is needed including: (1) better access to early query results while queries are running and (2) making the information stored in a data warehouse as fresh as possible.

For the first problem, we introduce a non-blocking parallel hash ripple join algorithm to support interactive queries in a parallel DBMS. Compared to previous work, our parallel hash ripple join algorithm (1) combines parallelism with sampling to speed convergence, and (2) maintains good performance in the presence of memory overflow. We demonstrate the performance of our approach with a prototype implementation in a parallel DBMS.

For the second problem, we propose two techniques to improve the efficiency of immediate materialized view maintenance. We identify two challenges for immediate materialized view maintenance:

(1) In parallel RDBMSs, simple single-node updates to base relations can give rise to expensive all-node operations for materialized view maintenance.

(2) Immediate materialized view maintenance with transactional consistency, if enforced by generic concurrency control mechanisms, can result in low levels of concurrency and high rates of deadlock.

To address the first challenge, 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.

To address the second challenge, we extend previous high concurrency locking techniques to apply to materialized view maintenance, and show how this extension can be implemented even in the presence of indices on the materialized view.

Acknowledgements

I would like to thank my advisor, Professor Jeffrey Naughton, for his help and guidance throughout my graduate study. He has always been patient with me and given me complete freedom in pursuing my research interests. He also assisted me in obtaining the necessary financial support, a portion of which was generously provided by NCR (Teradata) Corporation through a NCR Research Fellowship. Without Jeff's continuing encouragement and extreme patience, I would not have been able to complete this dissertation work successfully. The past four years of working with him has had and will continue to have a great impact on my professional career.

I would also like to thank Professor David Dewitt and Professor Raghu Ramakrishnan for their valuable comments and suggestions on my dissertation work. It has been a great honor to work with them in the prestigious Wisconsin Database group. Thanks are also due to Professor Jude Shavlik and Professor Yibin Pan for being on my dissertation committee.

This dissertation would not have been completed without the help and encouragement from my colleagues and friends. Curt Ellmann, Peter Haas, and Michael Watzke spent a great deal of time on co-authoring papers with me. Grace Au, Carrie Ballinger, Patricia Bamrah, Carrie Boone, Stephen Brobst, Josef Burger, Phillip Gibbons, Joseph Hellerstein, Ye Hong, Dan Hu, Kang Jaewoo, Henry Korth, Dakun Lin, David Lomet, Xiaohua Lu, Jiangbin Luo, Binh Ly, C. Mohan, Yibin Pan, Kevin O'Connor, Anita Richards, Pankaj Shatadal, Ambuj Shatdal, Qi Su, Dongbin Tao, Feng Tian, Liantao Wang, Yuan Wang, Honggang Zhang, and Min Zhong provided much valuable discussion and/or assistance in writing the papers.

I would like to specially thank my wife, Haiyan Chen, for her patience and support during my graduate study. I would also like to thank my son, Chenkai Luo, for being there and making everything I do meaningful.

This dissertation is in memory of Yongjing Cheng, who helped me develop the initial interest in academics when I was still a teenager.

Contents

List of Tables

List of Figures

Chapter 1: Introduction

Traditionally, data warehouses have been used to provide storage and analysis of large amounts of historical data. In a typical data warehouse, updates occur in batches at regular time intervals (e.g., every night). At all other times, the data warehouse is regarded as a "read-only" database, where users can pose long-running decision support queries.

Recently, 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. Most major RDBMS vendors have products and initiatives intended to address operational data warehousing, including Oracle's 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].

This new 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. In this dissertation, we investigate two such issues: (1) providing better access to early query results while queries are running and (2) improving update performance. More specifically, for the second issue, we investigate techniques for making the information stored in a data warehouse as fresh as possible. Of course, these two issues are not the only ones that arise from operational data warehousing. However, we believe that they are key issues that need to be resolved before an ideal operational data warehouse can be built.

The first part of this dissertation deals with providing better access to early query results while queries are running. As observed in [HHW97], for a complex decision support query, in many cases, the user is only interested in getting an approximate answer quickly. Based on the approximate answer, the user may abort, refine, and re-execute the query. Hence, it would be desirable to let the user know the approximate answer as early as possible. In [HHW97], Hellerstein et al. have proposed an online aggregation interface that is suitable for this purpose and permits the user to observe the progression of the submitted decision support query. Supporting online aggregation requires non-blocking join algorithms that can provide statistical guarantees [HHW97, HH99]. For this purpose, people have proposed a family of join algorithms called the ripple join algorithms [HH99], among which the hash ripple join algorithm had the best performance.

The existing hash ripple join algorithm suffers from the following two problems:

(1) The speed that the approximate answers converge can be slow if the number of tuples that satisfy the join predicate is small or if there are many groups in the output.

(2) If memory overflows (for example, because the user allows the algorithm to run to completion for an exact answer), the hash ripple join algorithm degenerates to block ripple join and performance suffers.

To solve these two problems, we propose a parallel hash ripple join algorithm that (a) combines parallelism with sampling to speed convergence, and (b) maintains good performance in the presence of memory overflow. We demonstrate the performance of our parallel hash ripple join algorithm with both an analytical model and a prototype implementation in a parallel DBMS.

The second part of this dissertation is about making the information stored in a data warehouse as fresh as possible. We focus our attention on improving the efficiency of immediate materialized join view maintenance in a parallel RDBMS. More specifically, we show that in a parallel RDBMS, simple single-node updates to base relations can give rise to expensive all-node join operations for materialized view maintenance. These all-node join operations are inefficient and can degrade the throughput of the parallel RDBMS.

To solve this problem, we propose two materialized join view maintenance methods that can change expensive all-node join operations to cheap single-node (few-node) join operations: the auxiliary relation method and the global index method. Our main idea is to trade storage space for materialized join view maintenance efficiency. We demonstrate the performance advantages of these two materialized join view maintenance methods with an analytical model. Also, we validate the analytical model with an implementation in a commercial parallel RDBMS.

The third part of this dissertation is also about improving the efficiency of immediate materialized join view maintenance. However, instead of resource usage, we focus our attention on the concurrency of immediate materialized join view maintenance transactions. More specifically, we show that if we use generic concurrency control mechanisms, immediate materialized aggregate join view maintenance can introduce many lock conflicts and/or deadlocks on the materialized aggregate join view. These lock conflicts and/or deadlocks can significantly degrade the throughput of the RDBMS.

To solve this problem, we propose a set of new locking protocols for materialized aggregate join views, and show how they can be implemented even in the presence of indices on the materialized aggregate join views. Our main idea is to utilize the associativity and commutativity of the materialized aggregate join view maintenance operations while avoiding certain anomaly. We give an example of the potential performance advantages of our locking protocols through a simulation study in a commercial RDBMS.

The rest of this dissertation is organized as follows. Chapter 2 introduces the parallel hash ripple join algorithm. Chapter 3 presents a comparison of three materialized join view maintenance methods in a parallel RDBMS. Chapter 4 describes the locking protocols for materialized aggregate join views. Finally, Chapter 5 concludes this dissertation.

Chapter 2: A Scalable Hash Ripple Join Algorithm

2.1 Introduction

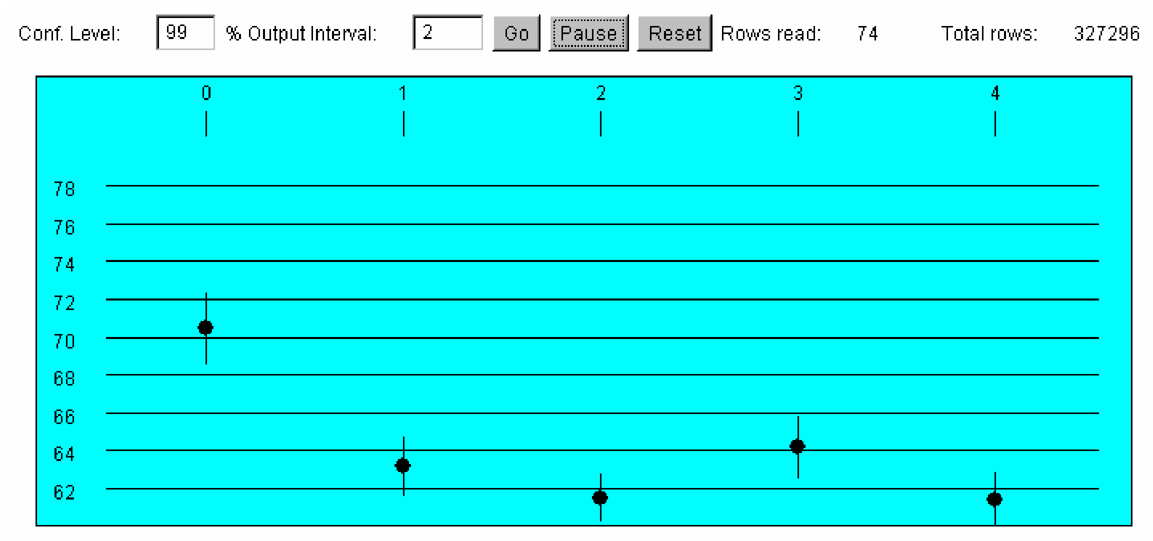


Figure 2.1 An online aggregation interface for a query of the form: select avg(temperature) from t group by site.

Online aggregation was proposed by Hellerstein et al. [HHW97] as a technique to enable users to obtain approximate answers to complex queries far more quickly than the exact answer can be computed. The basic idea is to sample tuples from the input relations and compute a continually-refining running estimate of the answer, along with a “confidence interval” that indicates the precision of the running estimate; such confidence intervals typically are displayed as error bars in a graphical user interface such as shown in Figure 2.1. The precision of the running estimate improves as more and more input tuples are processed. In [HH99], Haas and Hellerstein proposed a family of join algorithms, which they termed “ripple joins,” to support online aggregation for join-aggregate queries. A typical query handled by these algorithms is the following:

> select online *A.e*, *avg(B.f)*
>
> from *A*, *B*
>
> where *A.c* = *B.d*
>
> group by *A.e*;

Among the ripple join algorithms proposed by Haas and Hellerstein, the hash ripple join algorithm had the best performance. They showed that the algorithm can converge very quickly to a good approximation of the exact answer, and provided formulas for computing running estimates and confidence intervals.

Although the hash ripple join rapidly gives a good estimate for many join-aggregate problem instances, the convergence can be slow if the number of tuples that satisfy the join predicate is small or if there are many groups in the output. This is not a property of the algorithm itself, but is inherent to statistical estimation. The issue is that many input tuples must be processed before a single relevant sample is produced. For example, in our example query above, suppose that *A.e* has 500 distinct values. Then there will be 500 averages to be estimated, and each join tuple of *A.e* will contribute to only one out of the 500. Thus if we need 100 samples to generate a satisfactory estimate of one of the averages, we will need to generate 50,000 join tuples, and the hash ripple join algorithm may not converge quickly enough. As an even more extreme example, consider a join that returns only a single tuple. In this case, all of the query-processing effort will be spent on finding this tuple, and there will be no benefit at all to sampling.

In this chapter we propose a new algorithm, the parallel hash ripple join algorithm, to investigate whether parallelism can be combined with sampling in order to extend the range of queries that are amenable to online processing. While there is a long tradition of using parallelism to speed up join algorithms, it was not clear to us at the outset that parallelism could be used to speed up ripple joins, in which we are estimating the answer rather than computing it exactly. Through an implementation in a parallel DBMS we show that it is indeed possible – in our experiments we observed speedup and scaleup properties that closely match those of the traditional parallel hybrid hash join algorithm [SD89].

Applying parallelism to ripple joins raises some interesting and non-trivial statistical issues. This is in contrast to the case for, say, traditional hash joins, in which (at least algorithmically) a uniprocessor hash join generalizes in a straightforward way to a multiprocessor hash join. Our general approach is to use stratified sampling techniques that are similar in spirit to [Coc77]. In our setting, the strata must be defined very carefully to ensure that taking a simple random sample from each input relation at each “source” node (where the tuples are originally stored) produces a simple random sample from each stratum at each “join” node (where the tuples are joined). It turns out, perhaps contrary to intuition, that there must be many strata corresponding to each join node. Moreover, in contrast to classical stratified sampling as in [Coc77], samples from different strata are not in general independent, so that the classical confidence-interval formulas must be modified.

Another issue is that the parallel hash ripple join introduces a new source of statistical error. Briefly, at any point during the execution of the algorithm, the global running estimate depends upon (a) the current estimates of the aggregates for each stratum, and (b) estimates of the size of each stratum. The error from (b) of course does not arise in a uniprocessor ripple join. In this chapter, as a first step, we analyze the error when the sizes of the strata are known exactly, and highlight some practical situations in which this will indeed be the case. For those cases in which the sizes of the strata cannot be known in advance, our algorithm is still correct, but our current analysis is not strong enough to allow the system to display valid “error bars” during execution. Extending our analysis to provide these error bars in the most general case appears to be a difficult open problem in statistics.

Our parallel hash ripple join algorithm also extends the original hash ripple join algorithm presented in [HH99] to provide better performance when memory overflows during the computation. If one expects that a user will always stop a query after a reasonably precise estimate has been computed, there is probably no need for this, for with modern memory sizes it seems unlikely that memory will overflow before this point. We think, however, it is possible that in some cases a user will want to let a ripple join algorithm continue until it has computed the exact answer. In such cases memory overflow is likely, and the hash ripple join algorithm presented in [HH99] will degenerate to the much slower block ripple join. One desirable property of the algorithm in [HH99] is that it continues to process tuples from the input relations in a random and independent way throughout. That is, upon memory overflow, [HH99] maintains independence and randomness at the expense of performance.

In this chapter we suggest making the opposite tradeoff. That is, upon memory overflow, we sacrifice guarantees of randomness and independence in order to guarantee good performance. It is not that we are deliberately introducing inaccuracies in the estimate; rather, as tuples are staged through memory and disk we may introduce correlations that break the assumptions of randomness required for the computation of confidence intervals. Our rationale is that memory is expected to overflow when users are running the algorithm to completion, not when they are still “watching” the estimate and waiting for the error bounds to become acceptable. We show through an analytic model that over a wide range of memory sizes, our hash ripple join algorithm is dramatically faster than the block ripple join algorithm upon memory overflow.

2.2 Related Work

Figure 2.2 illustrates the original hash ripple join algorithm [HH99].

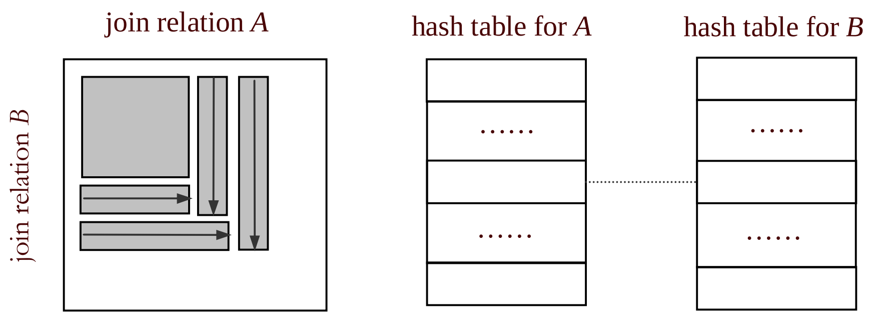


Figure 2.2 The original hash ripple join algorithm.

The original two-table hash ripple join [HH99, WA91] uses two hash tables, one for each join relation, that at any given point contain the tuples seen so far. At each sampling step, one previously unseen tuple is randomly retrieved from one of the two join relations. The join algorithm first decides which join relation is the source of the tuple. Then the tuple is joined with all matches in the hash table built for the other relation. Also, the tuple is inserted into its hash table so that tuples from the other join relation that arrive later can be joined correctly. As the hash tables grow in size, memory may overflow. When this occurs, the algorithm in [HH99] falls back to the block ripple join algorithm. At each step, the block ripple join algorithm retrieves a new block of one relation, scans all the old tuples of the other relation, and joins each tuple in the new block with the corresponding tuples there.

Ripple join algorithms for online aggregation are similar in some ways to the XJoin algorithm [UF99], which dynamically adjusts the algorithm's behavior in accordance with changes in the run time environment. XJoin was proposed for adaptive query processing [HFC$^+$00], where tuples are assumed to be arriving over a wide area network such as the Internet. To deal with this environment, the XJoin's behavior is complex, and if one attempts to use the XJoin for online aggregation, it will be difficult to make statistical guarantees.

In other work dealing with query processing over unpredictable and slow networks, [IFF$^+$99] propose the incremental left flush and the incremental symmetric flush hash join algorithms. Like the Xjoin algorithm, these algorithms are complex and do not lend themselves to statistical analyses. Furthermore, these algorithms both block in certain situations, which also makes them problematic for online aggregation.

2.3 Parallel Hash Ripple Join Algorithm

2.3.1 Overview of the Parallel Hash Ripple Join Algorithm

Suppose we want to equijoin two relations *A* and *B* on attributes *A.c* and *B.d* as in the following SQL query from the introduction:

```
select online A.e, avg(B.f)
from A, B
where A.c = B.d
group by A.e;
```

We first present the parallel hash ripple join algorithm, and then we show how to use it to support online aggregation in a parallel RDBMS in Section 2.3.3.

Originally, the tuples of *A* and *B* are stored at a set of source nodes according to some initial partitioning strategy (such as hash, range, or round-robin partitioning). A split vector, which maps join attribute values to processors, is used to redistribute the tuples of *A* and *B* during join processing. The goal of redistribution is to allocate the tuples of the join relations so that each join node performs roughly equal work during the execution of the algorithm.

A traditional parallel hybrid hash join algorithm [SD89] is performed in two phases. First, the algorithm redistributes the tuples of *A* (the build relation) to the nodes where the join will run, where some are added to the in-memory hash tables as they arrive, while others are spooled to disk. Then the tuples of *B* (the probe relation) are redistributed to the same nodes, and the hash tables built in the first phase are probed for some tuples of *B*, while the remainder of the *B* tuples are also spooled to disk.

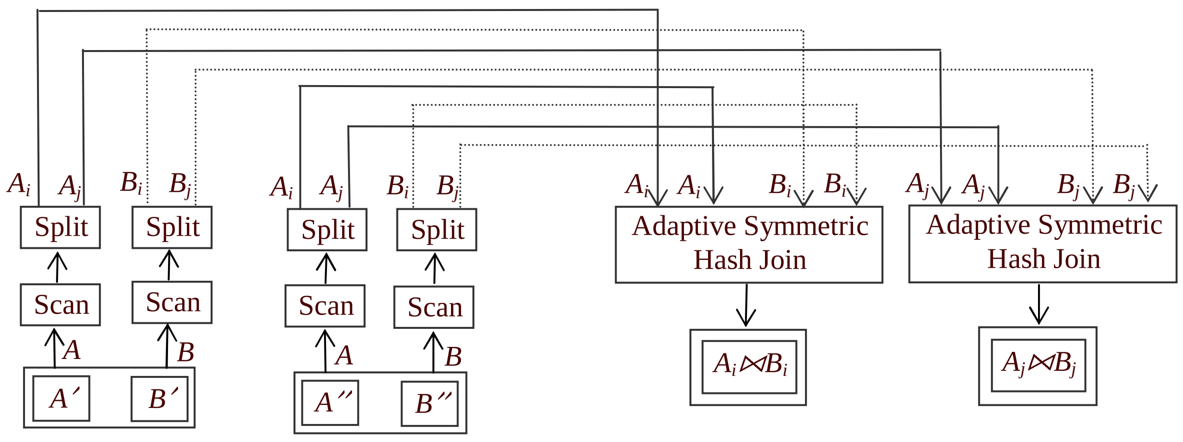


Figure 2.3 Dataflow network of operators for the parallel hash ripple join algorithm.

Finally, the disk-resident portions of *A* and *B* are joined. The result is that no output tuples are produced until the build relation is completely redistributed and then the second phase begins. This situation must be avoided in online aggregation because we would like to produce tuples as quickly as possible for the downstream aggregate operators. In a parallel ripple join algorithm, redistribution of the tuples should occur simultaneously with the join. Thus our parallel hash ripple join algorithm does the following three things simultaneously by multi-threading at each node, as shown in Figure 2.3:

1. Thread *RedisA* redistributes the tuples of *A* according to the split vector.
2. Thread *RedisB* redistributes the tuples of *B* according to the split vector.
3. Thread *JoinAB* performs the local join of the incoming tuples of *A* and *B* from redistribution using the adaptive symmetric hash join algorithm described below.

(The algorithms described in this chapter simplify when the tuples of *A* and/or *B* are already at the appropriate join nodes and need not be redistributed.)

To ensure that the tuples of *A* and *B* arrive at the nodes randomly from redistribution, we need to access them randomly before redistribution. One way to do so is to use a random sampling operator at each node as the input to the redistribution operator. In some applications scanning via a random sampling operator will be too slow so, as proposed in [HHW97], we can utilize a heap scan for heap files, or an index scan if there is an index such that there is no correlation between the aggregate attribute and the indexed attribute. Alternatively, as an engineering approximation to "pure" sampling, the data can be stored in random order on disk, so that sampling reduces to scanning; in this scheme the data on disk must periodically be randomly permuted (using, e.g., an online reorganization utility) to prevent the samples from becoming "stale" [HHW97].

At each node we maintain two hash tables on the join attributes, H_A for *A* and H_B for *B*, using the same hash function *H*. Usually, symmetric hash join requires that both the hash tables H_A and H_B can be held in memory [WA91], which may not always be possible. Consequently, we revise the symmetric hash join algorithm to fit our needs, the result of which we call the adaptive symmetric hash join algorithm.

2.3.2 Dealing with Memory Overflow

During the join processing, tuples are stored in hash tables H_A and H_B at each node. The hash table H_A (H_B) is divided into buckets. Each bucket E_A (E_B) is divided into a memory part, MP_A (MP_B), and a disk part, DP_A (DP_B). For performance reasons, one page of the disk part DP_A (DP_B), which we denote by P_A (P_B), is kept in memory as a write buffer. We call each pair of hash table buckets, E_A and E_B, with the same hash value the hash table bucket pair E_{AB}. This is conceptual; in practice, due to the limited memory size and the large number of hash table bucket pairs, we need to group many buckets of a hash table together to share the same memory part, disk part, and write buffer. The hash table buckets that are grouped together should be the same for the two hash tables.

Figure 2.4 shows the hash tables at a specific node:

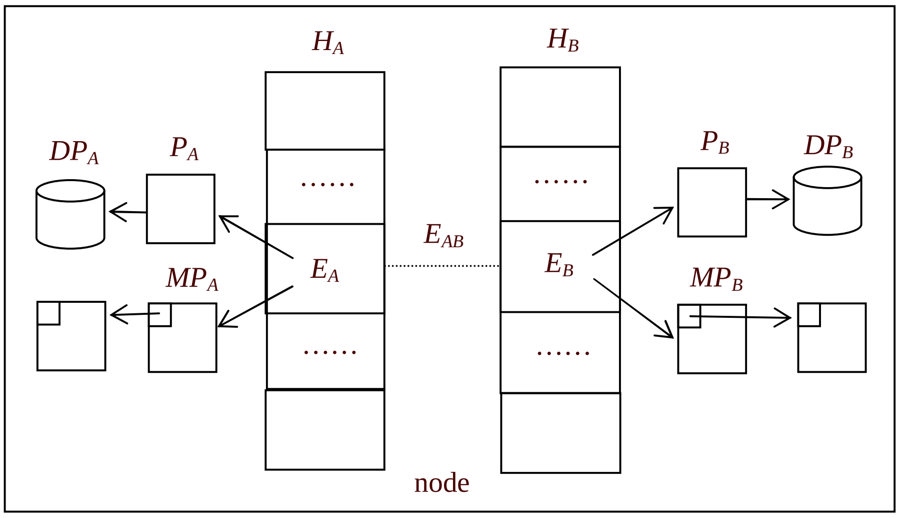


Figure 2.4 Hash tables at a node.

The adaptive symmetric hash join at each node is composed of two stages.

2.3.2.1 First Stage: Redistribution Phase

The goal of the first stage is to redistribute and join as many of the tuples of *A* and *B* as possible with the available memory. The first stage is completed when all the tuples of both relations have been redistributed.

Initially, for each hash table bucket pair, E_{AB}, MP_A in E_{AB} contains one page and MP_B in E_{AB} contains one page. At the node, we organize the other memory pages that can be allocated to MP_A and MP_B into a buffer pool *BP*.

Stage 1-1: memory-resident redistribution phase

When a tuple, T_A of A or T_B of B, comes from redistribution, we use the hash function H to find the corresponding hash table bucket pair E_{AB}.

If the tuple is T_A,

T_A is inserted into MP_A and joined with the tuples in MP_B.

If MP_A becomes full,

if the buffer pool BP is not empty, a page is allocated from BP to MP_A;

if the buffer pool BP is empty, then MP_A in E_{AB} becomes full first, and we enter stage 1-2 for this hash table bucket pair E_{AB}.

If the tuple is T_B, then we perform the operations described above, except that we switch the roles of A and B.

Stage 1-2: memory overflow redistribution phase

When a tuple T_A of A or T_B of B comes from redistribution, we use the hash function H to find the corresponding hash table bucket pair E_{AB}.

If at stage 1-1 MP_A in E_{AB} became full first, then:

If the tuple is T_A, it is written to the write buffer P_A. Whenever P_A becomes full, we write P_A to DP_A. Thus P_A can accept tuples from A again.

If the tuple is T_B, it is joined with the tuples in MP_A. If MP_B is not yet full, then T_B is inserted into MP_B. Otherwise T_B is written to the write buffer P_B. Whenever P_B becomes full, we write P_B to DP_B. Thus P_B can accept tuples from B again.

If at stage 1-1 MP_B in E_{AB} became full first, then we perform the operations described above, except that we switch the roles of A and B.

As a special case, for a given hash table bucket pair, E_{AB}, by the time MP_A (MP_B) in E_{AB} becomes full first at stage 1-1, if all tuples of B (A) have arrived from redistribution, DP_B (DP_A) in E_{AB} will be empty. At stage 1-2, whenever a tuple T_A of A (T_B of B) comes from redistribution, we only need to join it with the appropriate tuples in MP_B (MP_A) without writing it to DP_A (DP_B). In this way, we avoid the work for that E_{AB} at the second stage.

2.3.2.2 Second Stage: Disk Reread Phase

When all the tuples of A and B have arrived from redistribution, we enter the second stage for the node. That is, all the hash table bucket pairs enter stage 1-2 at different times, but they enter the second stage at the same time. At that time, for a given hash table bucket pair E_{AB}, if MP_A (MP_B) in E_{AB} became full first at stage 1-1, then all the tuples in E_B (E_A) have been joined with the tuples in MP_A (MP_B). We only need to join the tuples in E_B (E_A) with the tuples in DP_A (DP_B). We assume throughout that the DP_A (DP_B) part of each hash table bucket pair can fit in memory; the overall memory requirements of the algorithm are discussed in detail in Section 2.3.5.

The second stage proceeds as follows. We select, one at a time and in random order, those hash table bucket pairs whose DP_A parts or DP_B parts have been used at the first stage. (If the hash table bucket pairs are grouped together, we actually need to select the groups of hash table bucket pairs instead of the individual bucket pairs one by one.) For each hash table bucket pair, we perform the following operations:

Initialize an in-memory hash table H_{DP} that uses a hash function H' different from H.

If at stage 1-1 MP_A in E_{AB} became full first, then:

Stage 2-1 The tuples in DP_A (including the tuples in P_A) are read from the disk into memory. At the same time, they are joined with the tuples in MP_B and inserted into H_{DP} according to the hash values of H' for their join attributes.

Stage 2-2 The tuples in DP_B (including the tuples in P_B) are read from the disk into memory. At the same time, they are joined with the tuples in DP_A utilizing the hash function H' and the hash table H_{DP}.

If at stage 1-1 MP_B in E_{AB} became full first, then we perform the same operations described above except that we switch the roles of A and B.

Free the in-memory hash table H_{DP}.

2.3.2.3 Work Done at Different Stages

For a hash table bucket pair E_{AB}, suppose that at stage 1-1 MP_A in E_{AB} became full first. Then the stages at which the work is done for joining the tuples in E_A and tuples in E_B are shown in Figure 2.5:

		join relation *A*	
		MP_A	DP_A
join relation *B*	MP_B	stage 1-1 & stage 1-2	stage 2-1
	DP_B	stage 1-2	stage 2-2

Figure 2.5 Work done at each stage for a hash table bucket pair.

We expect our parallel hash ripple join algorithm to be commonly used in two modes:

(1) Exploratory mode: It is highly likely that users will abort the online query execution at stage 1-1 when both of the hash tables H_A and H_B are in memory, long before the query execution is finished. So, usually all the operations are done in the memory phase and our algorithm does not need to deal with the disk phase, which helps ensure high performance.

(2) Exact result mode: If users do not abort the query execution, our parallel hash ripple join algorithm has the advantage that the disk phase is handled efficiently without compromising the performance of the memory phase.

2.3.2.4 Random Selection Algorithm

In some cases, users may be still looking for an approximate answer upon memory overflow. To partially fulfill this requirement, we may generate the join result tuples in a nearly random order at the second stage of the algorithm. Note that we are not claiming true statistical randomness here; rather, we are using heuristics to try to avoid correlation in the memory overflow case. As mentioned in the previous subsection, we do this by selecting those hash table bucket pairs whose DP_A parts or DP_B parts have been used at the first stage individually on a random and non-repetitive basis. If the join attributes have no correlation with the aggregate attribute, we only need to select the hash table bucket pairs sequentially at the second stage. Otherwise, we can use a random shuffling algorithm [Knu98] to rearrange the hash table bucket pairs prior to selection.

2.3.3 Supporting Online Aggregation

Ripple joins [HH99] were proposed to support online aggregation [HHW97]. To support online aggregation in a parallel environment, we use the strategy of the Centralized Two Phase aggregation algorithm [SN95], as shown in Figure 2.6 (see below). First, each data server node does aggregation on its local partition of the relations. Then these local results are sent to a centralized query coordinator node from time to time to be merged for the global online results. In this way, we can support operators such as *SUM*, *COUNT*, *AVG*, *VARIANCE*, *STDEV*, *COVARIANCE*, and *CORRELATION*.

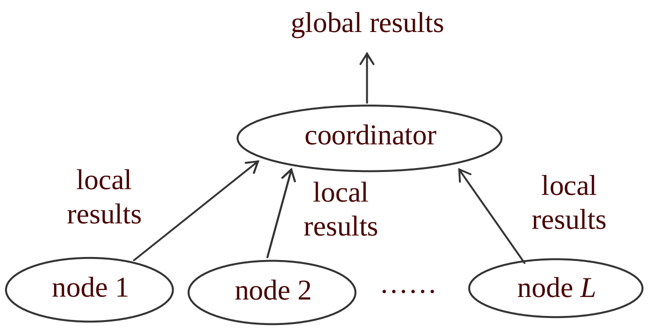
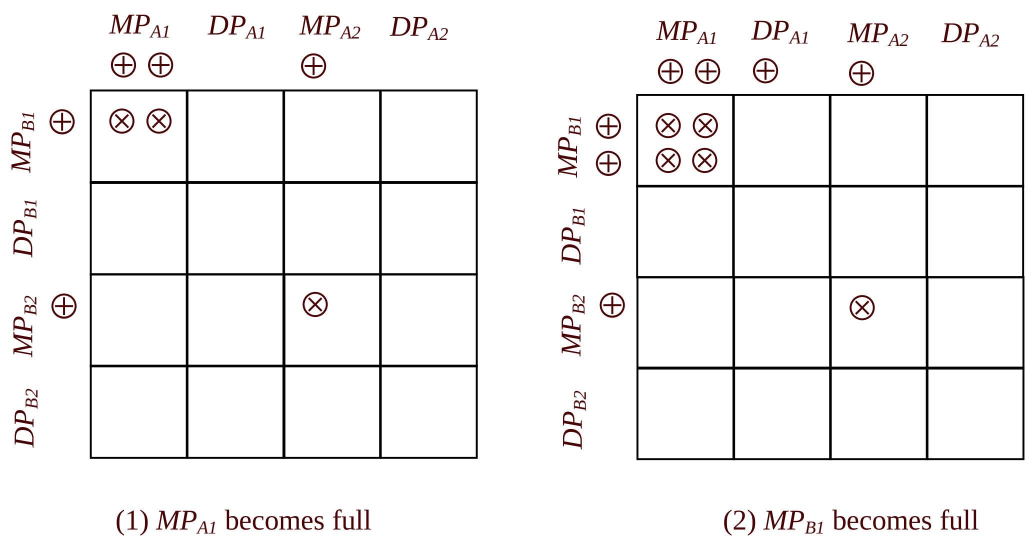
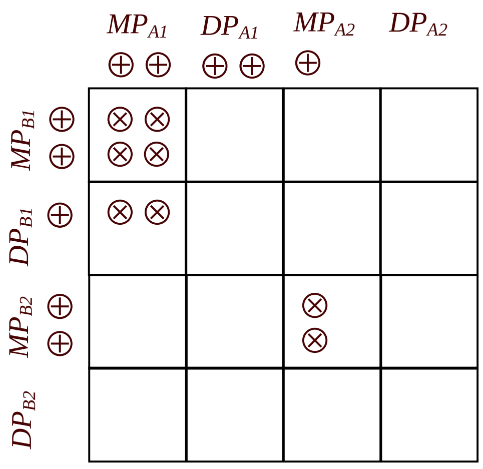


Figure 2.6 Computing global results from local results.

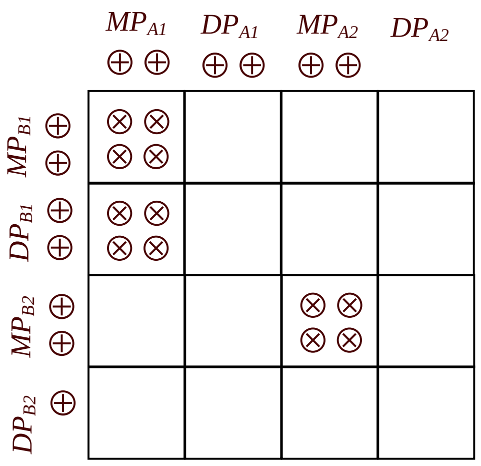
2.3.4 Running Join Example

We illustrate the operation of a join with a running example as follows. We only consider one node, at which each hash table has two buckets. Then we have the following parts: MP_{A1}, DP_{A1}, MP_{A2}, DP_{A2}, MP_{B1}, DP_{B1}, MP_{B2}, and DP_{B2}. Suppose MP_{A1} becomes full first, then MP_{B1}, then MP_{B2}, and finally MP_{A2}. Notice that no work needs to be done for joining the tuples in MP_{Ai} / DP_{Ai} and MP_{Bj} / DP_{Bj} when $i \neq j$. Let $\oplus$ denote the tuples arrived, and $\otimes$ denote the join work done. Then the progress of the join may look as shown in Figure 2.7.

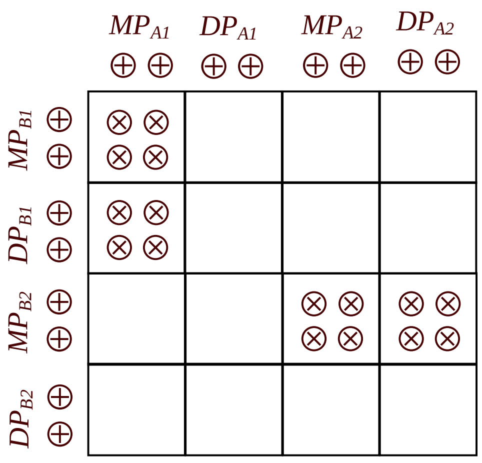




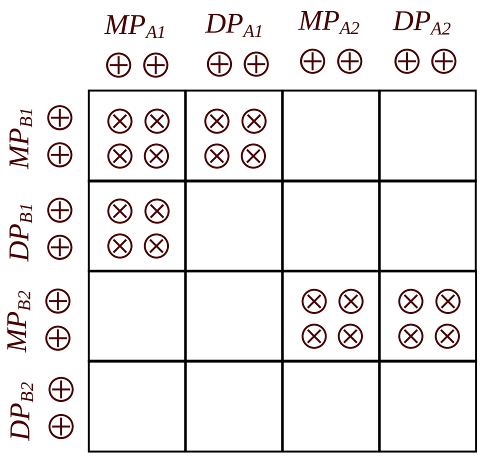
(3) MP_{B2} becomes full



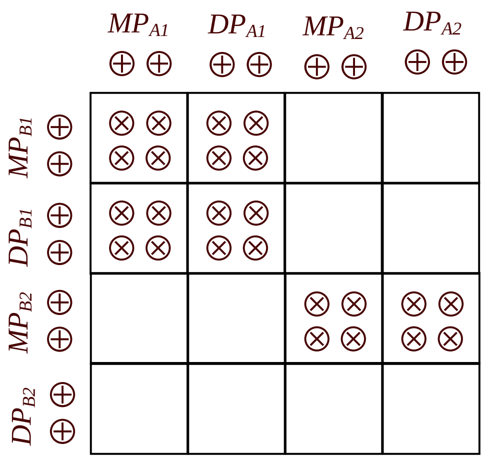
(4) MP_{A2} becomes full



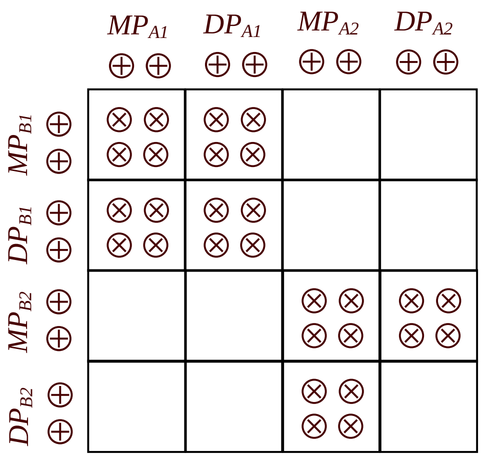
(5) all tuples of A and B have arrived from redistribution



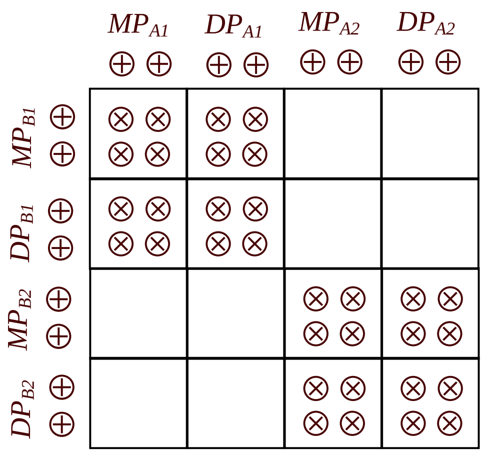
(6) DP_{A1} is read from the disk into memory



(7) DP_{B1} is read from the disk into memory



(8) DP_{B2} is read from the disk into memory



(9) DP_{A2} is read from the disk into memory

Figure 2.7 Work done in different phases of the running example.

2.3.5 Memory Requirements

The total memory requirement of the parallel hash ripple join algorithm is the combined size of all the memory parts (MP_A and MP_B), all the write buffers (P_A and P_B), and H_{DP}. The size of H_{DP} is roughly equal to the size of the largest disk part (either DP_A or DP_B). Assume there are H hash table bucket pairs (groups of hash table bucket pairs) at the node, and all the tuples are evenly distributed among the hash table bucket pairs. Let $||x||$ denote the size of x in pages and M denote the size of available memory in pages. The memory size M is sufficient if

$$M \geq H(||MP_A||+||MP_B||+||P_A||+||P_B||)+||H_{DP}||.$$

Since $||H_{DP}|| \approx max(||A||, ||B||)/H$, $||MP_A|| \geq 1$, $||MP_B|| \geq 1$, $||P_A|| \geq 1$, and $||P_B|| \geq 1$, it follows that the minimal sufficient memory size is $M=4H+max(||A||, ||B||)/H$. Thus the parallel hash ripple join algorithm requires that the sizes of the input relations satisfy the condition $max(||A||, ||B||) \leq H(M-4H)$. If this condition cannot be satisfied, the parallel hash ripple join algorithm falls back to the block ripple join algorithm at each node upon memory overflow.

2.3.6 Analytical Performance Model

To gain insight into the algorithms' behavior upon memory overflow, we use a simple analytical model. Table 2.1 shows the system parameters used. The original hash ripple join algorithm only works in the uniprocessor environment. Our parallel hash ripple join algorithm also works in the uniprocessor environment by degenerating to the adaptive symmetric hash join algorithm. To make the comparison fair, we only consider the uniprocessor environment. We assume that $||A||=||B||$, $H=\sqrt{||A||}/2$, and the time required to join a tuple with tuples of the other relation is a constant.

$\|\|A\|\|$	size of relation *A* in pages
$\|\|B\|\|$	size of relation *B* in pages
S	number of tuples/page
join	time required to join a tuple with tuples of the other relation
IO	time required to read/write a page
BLOCK	block size of the block ripple join algorithm in pages

Table 2.1 System parameters used in the analytical model.

We differentiate between the time to run to completion for the original hash ripple join algorithm [HH99] and our parallel hash ripple join algorithm in three cases:

(1) If $\|A\|+\|B\|\le M$, i.e., both join relations can fit into memory, then both the memory requirement of the parallel hash ripple join algorithm and that of the original hash ripple join algorithm are met. Both algorithms read the tuples of the join relations from the disk only once, so the execution time of either algorithm is

$$(\|A\|+\|B\|)\times IO+(\|A\|+\|B\|)\times S\times join$$

(2) If $\|A\|+\|B\|>M$ and $max(\|A\|, \|B\|)\le H(M-4H)$, then the memory requirement of the parallel hash ripple join algorithm is met but that of the original hash ripple join algorithm is not. For the parallel hash ripple join algorithm, upon memory overflow, each tuple of a join relation is written to the disk at most once, and read from the disk at most twice. The execution time of the parallel hash ripple join algorithm is:

$$(M+(\|A\|+\|B\|-M)\times 3)\times IO+(\|A\|+\|B\|)\times S\times join$$

$$=(3(\|A\|+\|B\|)-2M)\times IO+(\|A\|+\|B\|)\times S\times join\,.$$

In contrast, the original hash ripple join algorithm falls back to the block ripple join algorithm in this situation. Upon memory overflow, each block of relation A needs to be joined with from $M/2$ to $\|B\|$ pages of tuples of B, so the average number of disk I/Os for a given block of A is approximately $BLOCK+(M/2+\|B\|)/2$. There are $(\|A\|-M/2)/BLOCK$ such blocks of A. For relation B it is the same except that we switch the roles of A and B in the formulas. Thus the execution time of the original hash ripple join algorithm is:

$$(M+\frac{\|A\|-M/2}{BLOCK}\times(BLOCK+\frac{M/2+\|B\|}{2})+$$

$$\frac{\|B\|-M/2}{BLOCK}\times(BLOCK+\frac{M/2+\|A\|}{2}))\times IO+(\|A\|+\|B\|)\times S\times join$$

$$= (\|A\| + \|B\| + \frac{\|A\| \times \|B\| - M^2/4}{BLOCK}) \times IO + (\|A\| + \|B\|) \times S \times join.$$

(3) If *max(||A||, ||B||)>H(M-4H)*, then neither the memory requirement of the parallel hash ripple join algorithm nor that of the original hash ripple join algorithm is met. Both algorithms fall back to the block ripple join algorithm upon memory overflow. Thus the execution time of either of them is:

$$(\|A\| + \|B\| + \frac{\|A\| \times \|B\| - M^2/4}{BLOCK}) \times IO + (\|A\| + \|B\|) \times S \times join.$$

Setting the system parameters as shown in Table 2.2, we present in Figure 2.8 (see below) the resulting performance of the parallel hash ripple join algorithm (PHRJ) and the original hash ripple join algorithm (OHRJ).

$\|\|A\|\|$	2000
$\|\|B\|\|$	2000
S	40
join	400 microseconds
IO	30 milliseconds
BLOCK	60
page size	8000 bytes

Table 2.2 System parameter settings.

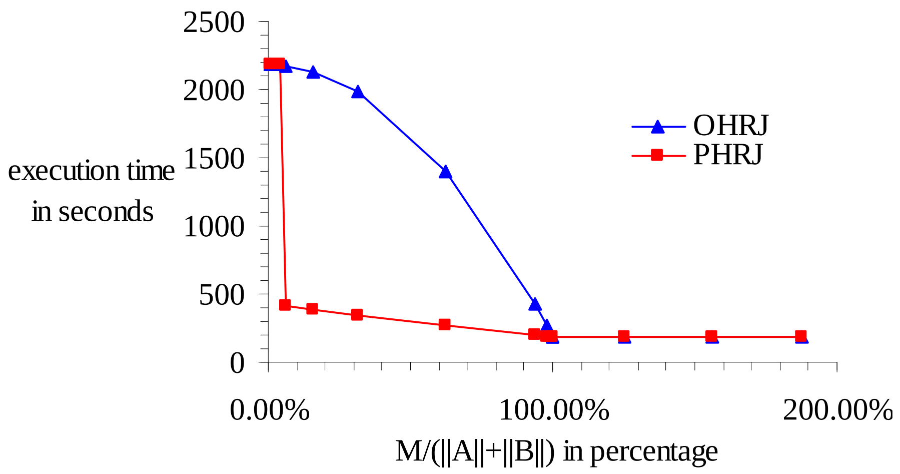


Figure 2.8 Execution time of join algorithms.

2.4 Statistical Issues

During Stage 1-1, the parallel hash ripple join algorithm supports the same online aggregation functionality as the original hash ripple join algorithm in [HH99]. Specifically, the algorithm permits running estimates and associated confidence intervals (“error bars”) to be displayed to the user; see, for example, Figure 2.1. In this section we describe the statistical methodology used to obtain these quantities.

Given the structure of the parallel hash join algorithm, it is natural to try to compute statistics locally and asynchronously at each join node and then combine these local results into a global running estimate and confidence interval, as outlined in Section 2.3.3. Classical stratified sampling techniques [Coc77] work in exactly this manner: the population to be sampled is divided into disjoint strata, sampling and estimation are performed independently in each stratum, and then global estimates are computed as a weighted sum of the local estimates, where the weight for the estimate from the *i*-th stratum is the stratum size divided by the population size.

In our setting, the strata must be chosen carefully. The most obvious choice in an L-node multiprocessor is to have the strata be the Cartesian products $A_i \times B_i$, for i ranging from 1 to L, where A_i (resp., B_i) is the set of tuples of A (resp., B) that are sent to node i for join processing. A moment's thought, however, shows that this choice is not correct, because even if the tuples being redistributed constitute random samples from their source nodes, the tuples arriving at a join node i do not, in general, constitute a growing simple random sample of A_i and B_i. To see this, suppose that there are two source nodes, say j and k, and the rate of tuple arrivals at node i is the same for each source node. Also suppose that at some point 1000 tuples of A_i have arrived at node i for processing, and consider two possible cases: in the first case there are about 500 tuples from each of nodes j and k, and in the second case all 1000 tuples are from node j. If the 1000 tuples are a true simple random sample from A_i, then both cases should be equally likely (because every sample of 1000 tuples from A_i is equally likely). Of course, the probability that the second case occurs is 0, whereas the first case occurs with positive probability. Our solution to this problem, as described in Section 2.4.1, is to define many strata, one for each (source node, join node) pair. This solution leads to another complication: unlike in classical stratified sampling, estimates computed from strata at the same join node are not statistically independent, so that the classical stratified sampling formulas are not applicable. We show, however, that the classical formulas can be extended to handle the correlations between estimates, leading to extensions of the estimation formulas in [HH98, HH99] to the setting of parallel sampling.

Defining the strata in the foregoing manner provides statistically valid random samples at each join node. However, as mentioned in the introduction, there is another problem. The estimation formulas for stratified sampling assume that the size of each stratum is known. There are cases in which the sizes of the strata are known precisely. For example, if detailed distribution statistics on the join attribute are maintained at each source node. Another important case is the "in-place" join, where relations A and B are already partitioned on the join attribute. In practice, for performance reasons, DBA's for parallel RDBMSs try to choose partitioning strategies so that the most common joins are indeed in-place joins. However, in general the sizes of the strata can only be estimated.

Developing estimation formulas that take into account both the uncertainty in the sizes of the strata and the uncertainty in the estimates at each node is a daunting task that requires the solution of some open statistical problems. Accordingly, as a first step, we provide estimation formulas that are valid when the sizes of the strata are indeed known precisely. This simplification lets us obtain some insight into the statistical performance of the parallel hash ripple join algorithm. When strata sizes are unknown, our algorithm is correct (in that the running estimate converges to the true answer as more and more tuples are processed) but we cannot provide statistically meaningful error bars for the user as the algorithm progresses.

2.4.1 Assumptions and Notation

We focus on queries of the form

select *op(expression)*

from *A*, *B*

where *predicate*;

where *op* is one of *SUM*, *COUNT*, or *AVG*. All of our formulas extend naturally to the case of multiple tables. When *op* is equal to *COUNT*, we assume that *expression* reduces to the SQL "*" identifier. The predicate in the query can in general consist of conjunctions and/or disjunctions of boolean expressions involving multiple attributes from both *A* and *B*; we make no simplifying assumptions about the joint distributions of the attributes in either of these relations. We restrict attention to "large-sample" confidence intervals; see [Haa97, Haa00, HH98] for a discussion of other types of confidence intervals, as well as methods for dealing with *GROUP BY* and *DISTINCT* clauses.

Denote by A_{ij} the set of tuples of A that are stored on source node j and sent to join node i for join processing, and let $|A_{ij}|$ be the size of A_{ij} in tuples. Similarly define B_{ij} and $|B_{ij}|$. We allow i and j to be equal, so that source and join nodes may coincide. Assume for ease of exposition that any local predicates are processed at the join node and not at the source nodes. If local predicates are processed at the source nodes (actually a more likely scenario, for performance reasons), then the calculations given below are valid as stated, provided that A_{ij} is interpreted as the set of tuples that would be sent from node j to node i for processing if local predicates were ignored. As noted in the beginning of this section, we make the simplifying assumption that each quantity $|A_{ij}|$ and $|B_{ij}|$ is known precisely. For example, given detailed distribution statistics on the join attribute at each source node, together with the split vector entries, each $|A_{ij}|$ and $|B_{ij}|$ can readily be computed. For reasons of efficiency, it may be desirable to pre-compute each $|A_{ij}|$ and $|B_{ij}|$ and store this set of derived statistics with the base tables.

2.4.2 Running Estimator and Confidence Interval for *SUM*

First consider a fixed join node i. Suppose that there are L_i (resp., M_i) source nodes sending tuples of A (resp., B) to node i. Also suppose that n_{ij} tuples from A_{ij} and m_{ik} tuples from B_{ik} have been processed for each $1 \le j \le L_i$ and $1 \le k \le M_i$, and denote by $A_{ij}(n_{ij})$ and $B_{ik}(m_{ik})$ the respective sets of tuples. Under our assumptions, each set $A_{ij}(n_{ij})$ can be viewed as a simple random sample from A_{ij}, and similarly for each set $B_{ik}(m_{ik})$. As before, set $A_i = \bigcup_{j=1}^{L_i} A_{ij}$ and $B_i = \bigcup_{k=1}^{M_i} B_{ik}$, and observe that

$$A_i \bowtie B_i = \bigcup_{j=1}^{L_i} \bigcup_{k=1}^{M_i} A_{ij} \bowtie B_{ik} ,$$

where each join $A_{ij} \bowtie B_{ik}$ can be viewed as being computed by means of a standard (nonparallel) hash ripple join.

2.4.2.1 Running Estimator

For fixed *j*, *k*, observe that, as in [HH99],

$$\hat{\mu}_{ijk} = \frac{1}{n_{ij} m_{ik}} \sum_{(a,b)\in A_{ij}(n_{ij})\times B_{ik}(m_{ik})} expression_p(a,b)$$

is an unbiased and strongly consistent estimator of

$$\mu_{ijk} = \frac{1}{|A_{ij}||B_{ik}|} \sum_{(a,b)\in A_{ij}\times B_{ik}} expression_p(a,b),$$

where *expression*$_p$*(a,b)* equals *expression(a,b)* if *(a,b)* satisfies the *WHERE* clause, and equals 0 otherwise. Here "strongly consistent" means that, with probability 1, the estimator $\hat{\mu}_{ijk}$ converges to the true value μ_{ijk} as more and more tuples are sampled. "Unbiased" means that the estimator $\hat{\mu}_{ijk}$ would be equal on average to μ_{ijk} if the sampling and estimation process were repeated over and over. Setting $w_{ijk}=|A_{ij}||B_{ik}|/|A_i||B_i|$ for $1\le j\le L_i$ and $1\le k\le M_i$, it follows easily that

$$\hat{\mu}_i = \sum_{j=1}^{L_i}\sum_{k=1}^{M_i} w_{ijk}\hat{\mu}_{ijk}$$

is an unbiased estimator of

$$\mu_i = \frac{1}{|A_i||B_i|} \sum_{(a,b)\in A_i\times B_i} expression_p(a,b).$$

It then follows that, with $w_i=|A_i||B_i|$ for each *i*, the estimator $\hat{\mu} = \sum_i w_i\hat{\mu}_i$ is unbiased for

$$\mu = \sum_i w_i\mu_i = \sum_{(a,b)\in A\times B} expression_p(a,b),$$

which is the overall quantity that we are trying to estimate.

2.4.2.2 Confidence Interval

Recall that a *100p%* confidence interval for an unknown parameter μ is a random interval of the form $I=[L, U]$ such that $P(\mu\in I)=p$. Typically, a confidence interval has the symmetric form $I=[\hat{\mu}-\varepsilon, \hat{\mu}+\varepsilon]$, where $\hat{\mu}$ is an estimator of μ based on a random sample, and ε is also a quantity computed from the sample. In this formulation, ε is a measure of the precision of $\hat{\mu}$: with probability *100p%*, the estimator $\hat{\mu}$ is within $\pm\varepsilon$ of the true value μ. In our setting, $\hat{\mu}$ is the running estimator of the *SUM* query, the parameter μ is the true answer to the query based on all of the data, and ε is the length of the "error bar" on either side of the point estimate as in, e.g., Figure 2.1.

To obtain formulas for large-sample confidence intervals, we assume as an approximation that sampling is performed with replacement; the error in this approximation is negligible provided that we sample only a small fraction of the data. Then successive samples drawn from a specified set A_{ij} or B_{ij} can be viewed as independent and identically distributed observations. We also make the technical assumption that for each A_{ij} there exist positive constants c_{ij} and d_{ij} such that, with probability 1, $n_{ij}(k)/k\to c_{ij}$ and $m_{ij}(k)/k\to d_{ij}$ as $k\to\infty$, where $n_{ij}(k)$ (resp., $m_{ij}(k)$) is the number of tuples from A_{ij} (resp., B_{ij}) that have been processed after k tuples have been processed throughout the entire system. As a practical matter, we require that there be no large disparities between or among the c_{ij}'s and d_{ij}'s. This will certainly be the case when relations A and B are initially partitioned in a round robin manner and both processor speeds and transmission times between nodes are homogeneous throughout the system.

As before, suppose that n_{ij} tuples from A_{ij} and m_{ik} tuples from B_{ik} have been processed for each $1\le j\le L_i$ and $1\le k\le M_i$. Using arguments as in [HNS+96], the results in [HH98, HH99] can be extended in an algebraically tedious but straightforward manner to show that, provided each n_{ij} and m_{ik} is large, the estimator $\hat{\mu}$ defined above is approximately normally distributed with mean μ and variance $\sigma^2 = \mathrm{Var}[\hat{\mu}]$. As discussed in [HH99, Sec. 5.2.2], this asymptotic normality is not a simple consequence of the usual central limit theorem for independent and identically distributed (i.i.d.) random variables — the terms that are added together to compute $\hat{\mu}$ are far from being statistically independent. Given the foregoing extension of the usual central limit theorem, standard algebraic manipulations then show that the random interval $I = [\hat{\mu} - z_p\hat{\sigma}, \hat{\mu} + z_p\hat{\sigma}]$ is an approximate *100p%* confidence interval for μ. Here $\hat{\sigma}^2$ is any strongly consistent estimator of σ^2, and z_p is the unique number such that the area under the standard normal curve between $-z_p$ and z_p is equal to p. The crux of the problem, then, is to determine the form of σ^2 and identify a strongly consistent estimator $\hat{\sigma}^2$.

To this end, observe that the samples $\{A_{ij}(n_{ij}), B_{ik}(m_{ik}): i,j,k \geq 1\}$ are mutually independent, so that $\hat{\mu}_i$ is independent of $\hat{\mu}_j$ for $i \neq j$. It follows that $\sigma^2 = Var[\hat{\mu}] = \sum_i w_i^2 Var[\hat{\mu}_i]$. Now fix i and observe that $Var[\hat{\mu}_i] = \sum_{j,k,j',k'} w_{ijk} w_{ij'k'} Cov[\hat{\mu}_{ijk}, \hat{\mu}_{ij'k'}]$. Each term $Cov[\hat{\mu}_{ijk}, \hat{\mu}_{ij'k'}]$ can be computed as follows. For $B_0 \subseteq B$ and $a \in A$, denote by $\mu_1(a;B_0)$ the average of *expression$_p$(a,b)* over $b \in B_0$, and similarly denote by $\mu_2(b;A_0)$ the average of *expression$_p$(a,b)* over $a \in A_0$ for $b \in B$ and $A_0 \subseteq A$. Next, denote by $\sigma^{(1)}_{i,j,k,k'}$ the covariance of the pairs $\{(\mu_1(a;B_{ik}), \mu_1(a;B_{ik'})) : a \in A_{ij}\}$ and by $\sigma^{(2)}_{i,j,j',k}$ the covariance of the pairs $\{(\mu_2(b;A_{ij}), \mu_2(b;A_{ij'})) : b \in B_{ik}\}$. Also denote by $\sigma^{(1)}_{ijk}$ the variance of the numbers $\{\mu_1(a;B_{ik}) : a \in A_{ij}\}$ and by $\sigma^{(2)}_{ijk}$ the variance of the numbers $\{\mu_2(b;A_{ij}) : b \in B_{ik}\}$. Then straightforward calculations show that

$$Cov[\hat{\mu}_{ijk}, \hat{\mu}_{ij'k'}] = \begin{cases} 0 & if\ \ j \neq j' \ and\ k \neq k'; \\ n_{ij}^{-1}\sigma_{i,j,k,k'}^{(1)} & if\ \ j = j' \ and\ k \neq k'; \\ m_{ik}^{-1}\sigma_{i,j,j',k}^{(2)} & if\ \ j \neq j' \ and\ k = k'; \\ n_{ij}^{-1}\sigma_{ijk}^{(1)} + m_{ik}^{-1}\sigma_{ijk}^{(2)} + O((n_{ij}m_{ik})^{-1}) & if\ \ j = j' \ and\ k = k'. \end{cases}$$

Note that in the case $j=j'$ and $k=k'$, the formula for $Cov[\hat{\mu}_{ijk}, \hat{\mu}_{ij'k'}]$ is essentially the same as that given in [HH99]. An estimator $\hat{\sigma}^2$ for σ^2 can be computed as above, with each A_{ij} replaced by the sample $A_{ij}(n_{ij})$ and each B_{ik} replaced by $B_{ik}(m_{ik})$. Calculations as in [HNS⁺96] show that $\hat{\sigma}^2$ is indeed strongly consistent for σ^2. A development along the lines of [HH98] leads to efficient and stable numerical procedures for computing the various estimates described above.

2.4.3 Running Estimator and Confidence Interval for *COUNT* and *AVG*

Point estimates and confidence intervals for *COUNT* queries are computed almost exactly as described for *SUM* queries, but with $expression_p(a,b)$ replaced by $one_p(a,b)$, where $one_p(a,b)$ equals 1 if (a,b) satisfies the *WHERE* clause, and equals 0 otherwise.

We now consider running estimates for *AVG* queries. Denote by μ_s the answer to the *AVG* query when *AVG* is replaced by *SUM*, and by μ_c the answer to the *AVG* query when *AVG* is replaced by *COUNT*. Observe that the answer μ_a to the *AVG* is simply the *SUM* divided by the *COUNT*: $\mu_a=\mu_s/\mu_c$. As in [HH99], a natural estimator of μ_a is therefore $\hat{\mu}_a = \hat{\mu}_s / \hat{\mu}_c$, where $\hat{\mu}_s$ and $\hat{\mu}_c$ are the respective estimators of μ_s and μ_c as in Section 2.4.2.1. The estimator $\hat{\mu}_a$ is strongly consistent for μ_a; this assertion follows from the strong consistency of $\hat{\mu}_s$ for μ_s and $\hat{\mu}_c$ for μ_c. Although $\hat{\mu}_a$ is biased, the bias is typically negligible except when the samples are very small. (Indeed, the bias decreases as

$O(n^{-1})$, where *n* is the number of samples, as opposed to the $O(n^{-1/2})$ rate at which the confidence-interval length decreases.)

Several approaches are available for obtaining confidence intervals for *AVG* queries. As in [HH99], we can apply standard results on ratio estimation to find that when each n_{ij} and m_{ik} is large, the estimator $\hat{\mu}_a$ is distributed approximately according to a normal distribution with mean μ_a and variance $\sigma^2 = (\sigma_s^2 - 2\mu_a\sigma_{sc} + \mu_a^2\sigma_c^2)/\mu_c^2$, where $\sigma_s^2 = \mathrm{Var}[\hat{\mu}_s]$, $\sigma_c^2 = \mathrm{Var}[\hat{\mu}_c]$, and $\sigma_{sc} = Cov[\hat{\mu}_s, \hat{\mu}_c]$. Both σ_s^2 and σ_c^2 can be estimated as described in Section 2.4.2.2, and σ_{sc}^2 can be estimated in a similar manner, yielding an estimate $\hat{\sigma}^2$ of σ^2, and hence a confidence interval. The details of estimating σ_{sc}^2 are somewhat cumbersome and we omit them for brevity.

An alternative approach computes separate *100q%* confidence intervals for μ_s and μ_c as above, where *q=(1+p)/2*, and combines the intervals using Bonferroni's inequality. This latter inequality asserts that $P(C \text{ and } D) \geq 1 - P(\bar{C}) - P(\bar{D})$ for any events *C* and *D*, where $\bar{X}$ denotes the complement of an event *X*. Thus the probability that the two *100q%* confidence intervals simultaneously contain μ_s and μ_c, respectively, is at least $1 - 2(1-q) = p$. The resulting simultaneous bounds on the possible values of μ_s and μ_c then lead directly to bounds on the possible values of μ_a; by construction, these latter bounds hold with probability at least *p*. This approach is used in [Haa00] to obtain the symmetric *100p%* confidence interval $[\hat{\mu}_a - \varepsilon^*, \hat{\mu}_a + \varepsilon^*]$, where $[\hat{\mu}_s - \varepsilon_s, \hat{\mu}_s + \varepsilon_s]$ and $[\hat{\mu}_c - \varepsilon_c, \hat{\mu}_c + \varepsilon_c]$ are the initial *100q%* confidence intervals, and

$$\varepsilon^* = \frac{\hat{\mu}_c\varepsilon_s + |\hat{\mu}_s|\,\varepsilon_c}{\hat{\mu}_c(\hat{\mu}_c - \varepsilon_c)}.$$

A variant of this technique yields confidence intervals that are asymmetric about the point estimator, but are shorter than the symmetric intervals given above. Set $L_x = \hat{\mu}_x - \varepsilon_x$ and $U_x = \hat{\mu}_x + \varepsilon_x$, where *x* equals *s* or *c*. Then the confidence interval for μ_a is $[L_a, U_a]$, where

$$(U_a, L_a) = \begin{cases} (U_s / L_c, \quad L_s / U_c) & if \quad U_s \geq L_s \geq 0; \\ (U_s / U_c, \quad L_s / L_c) & if \quad L_s \leq U_s \leq 0; \\ (U_s / L_c, \quad L_s / L_c) & if \quad U_s > 0 > L_s. \end{cases}$$

In general, techniques based on Bonferroni's inequality are somewhat less burdensome computationally than techniques based on large-sample results for ratios, but yield longer confidence intervals.

2.5 Performance

In this section, we present results from a prototype implementation of the parallel hash ripple join algorithm in the parallel ORDBMS TOR 2.0.02. Our measurements were performed with the database client application and server running on four NCR WorldMark 4400 workstations, each with four 400MHz processors, 1GB main memory, six 8GB disks. We allocated a processor and a disk for each data server, so there were at most four data servers on each workstation. Before we ran each test, we restarted the computers to ensure a cold buffer pool.

The relations used for the benchmarks were based on the standard Wisconsin Benchmark relations [BDT83]. We have two join relations: *A* and *B*. The relevant fields from their common schema is shown as follows:

```
create table A
( unique1      bigint not null,
  unique2      integer not null,
```

…11 more integer attributes,

… three 52 character string attributes

);

In order to get more meaningful results, 500,000 and 50,000 tuple versions of the standard relations were constructed for *A* and *B*, respectively. We use the *unique1* attribute as the aggregate attribute and the *unique2* attribute as the join attribute. To prevent numerical overflow when doing aggregation, the type of the *unique1* attribute was changed from integer to bigint, which corresponds to the 8-byte long long type in C. Thus each relation consists of one 8-byte bigint attribute, twelve 4-byte integer attributes, and three 52-byte string attributes. Assuming no storage overhead, the length of each tuple is 212 bytes. The sizes of the relations *A* and *B* are approximately 106 and 10.6 megabytes, respectively. The *unique1* attribute of *A* was uniformly distributed between 0 and 499,999. There is no correlation between the aggregate attribute *unique1* and the join attribute *unique2*. No index was created for the attributes. At each node, we set the memory that a join operator can utilize to 22 megabytes.

2.5.1 Speedup Experiments

The most important performance metric for online aggregation is the rate at which the real-time results continuously produced by the online data exploration system converge to the final precise results.

We ran the following query on 1-node, 2-node, 4-node, 8-node, and 16-node configurations (where each node is a data server) with a global query coordinator using the parallel hash ripple join (PHRJ) algorithm and the classical blocking parallel hybrid hash join (PHHJ) algorithm. We chose the

PHHJ algorithm for comparison because it performs the best among all the four traditional blocking parallel join algorithms in [SD89].

select online *avg (A.unique1)*

from *A*, *B*

where *A.unique2* = *B.unique2*;

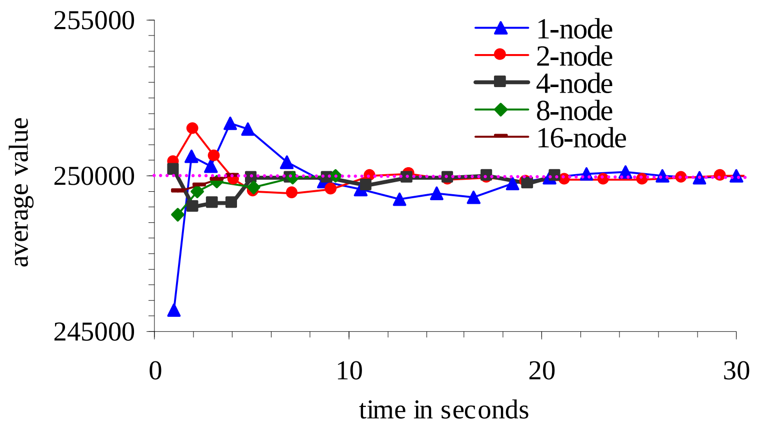


Figure 2.9 Average value computed over time (high join selectivity).

At the beginning, we generated the *unique2* attributes of both relations such that each tuple of *A* matches exactly one tuple of *B*. For the PHRJ algorithm. Figure 2.9 shows how the computed average value converges to the final precise result over time, with the final precise result indicated by the horizontal dotted line. Figure 2.10 (see below) shows how the number of join result tuples generated increases over time.

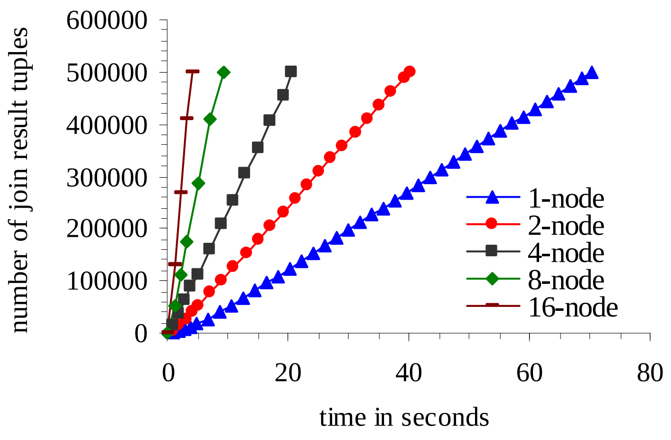


Figure 2.10 Join result tuples generated over time (high join selectivity).

Because relatively many tuples satisfy the join predicate and we are estimating for a single group (no "group by" clause), only a small number of samples need to be taken from the two join relations to generate enough join result tuples for a good approximation. The approximate answer converges to the final precise answer within seconds even in the uniprocessor environment.

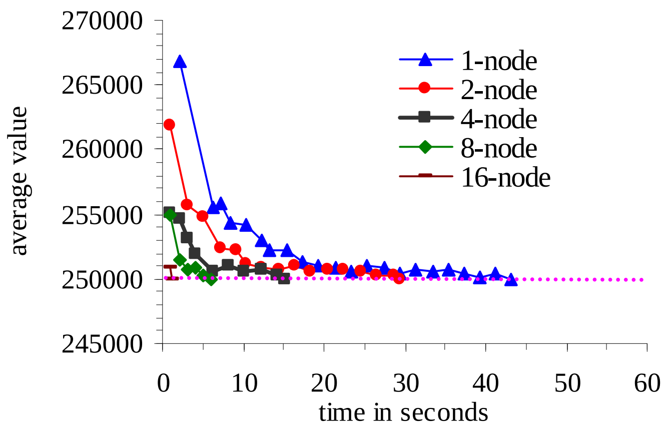


Figure 2.11 Average value computed over time (low join selectivity).

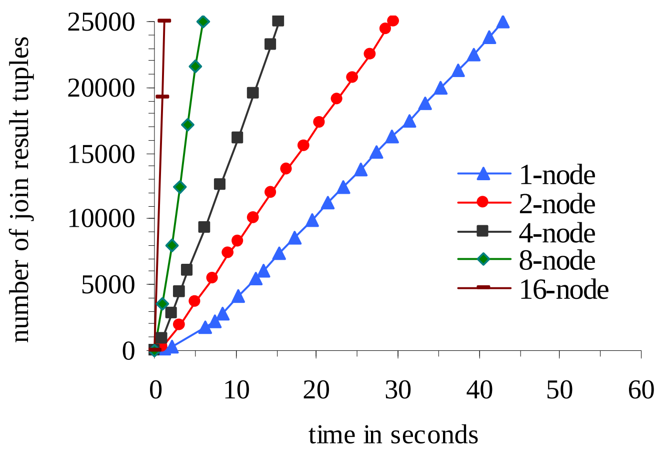


Figure 2.12 Join result tuples generated over time (low join selectivity).

In the remainder of this section, we reduced the number of tuples that satisfy the join predicate by a factor of 20 in order to present a more challenging problem statistically (note that in terms of difficulty of the estimation problem, this is similar to doing a “group by” on an attribute with 20 distinct values). That is, we generated the *unique2* attributes of both relations such that each tuple of *A* matches at most one tuple of *B* on *unique2*, and on average 1/20 of all the tuples of *A* have such a match. The corresponding results are shown in Figure 2.11 (see above) and Figure 2.12, respectively. Note that now, in contrast to the results in Figure 2.9, the multiple-node configurations converge noticeably quicker than the single node configurations.

Table 2.3 (see below) shows the time at which memory overflow occurs in the five configurations. We can see that memory overflow does not greatly influence the speed of generating the join result tuples. The average value that the PHRJ algorithm estimates achieves a relatively small tolerance within seconds, well before memory overflow occurs. Even after the memory overflows the join result tuples are still generated steadily, and the computed average value still approximates the final result very well.

1-node	2-node	4-node	8-node	16-node
8.1	10.3	11.8	none	none

Table 2.3 Time (seconds) that memory overflow occurs.

In each configuration, all nodes fetch tuples from the join relations and do the join work in parallel, so one might think that memory overflow would occur at the same time for all five configurations. In fact, as the number of nodes increases, the number of different values that the join attribute can take at each node decreases and the local join selectivity at each node increases. Thus in the same amount of time, a larger percentage of time is spent on joining rather than fetching tuples from the join relations at each node, and memory overflow occurs later. The delayed memory overflow lengthens the period in which our PHRJ algorithm can make strong statistical guarantee.

Table 2.4 shows the execution to completion time of the PHRJ algorithm and the traditional PHHJ algorithm [SD89] in the five configurations.

	1-node	2-node	4-node	8-node	16-node
PHRJ	43	29	15	6	3
PHHJ	37	19	10	4	2

Table 2.4 Execution to completion time (seconds) of the two parallel join algorithms (speedup).

In all five configurations, the blocking PHHJ algorithm produced the final result about 13% to 33% faster than our PHRJ algorithm. This is mainly due to the fact that our algorithm needs to build two hash tables, one for each join relation, while the blocking algorithm only needs to build one hash table for the inner join relation. For the PHRJ algorithm, the speed at which the join result tuples are generated in these five configurations is nearly proportional to the number of nodes used.

The PHRJ algorithm produces a reasonably precise approximation within seconds. It is up to two orders of magnitude faster than the time required by the PHHJ algorithm (which does not produce results until the join is nearly completed) to produce exact answers.

2.5.2 Scale-up Experiments

We ran the query on 1-node, 2-node, 4-node, 8-node, and 16-node configurations. Compared with that of the 1-node configuration (106M & 10.6M), the sizes of the relations for the other configurations were increased proportionally to the number of data server nodes. Thus the average workload at each node, which was measured by the sizes of the join relations there, was kept the same for the scale-up experiments.

Figure 2.13 shows the execution to completion time of the two parallel join algorithms in the five configurations. In all five configurations, the blocking PHHJ algorithm produced the final result about 13% to 32% faster than our PHRJ algorithm.

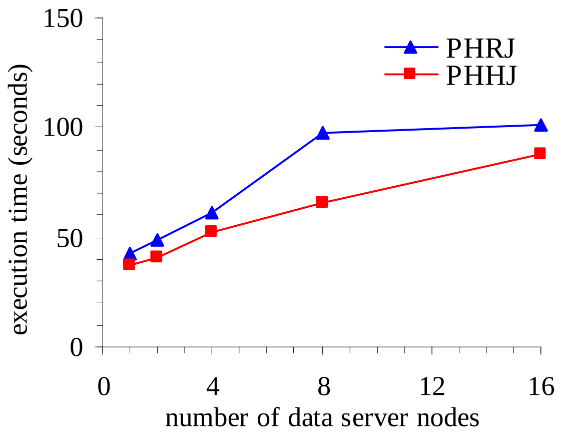


Figure 2.13 Execution to completion time of the two parallel join algorithms (scale-up).

2.6 Summary and Conclusions

The two primary tools that permit online exploration of massive data sets are sampling and parallel processing. In this chapter we initiate an effort to combine these tools, and introduce the parallel hash ripple join algorithm. This efficient non-blocking join algorithm permits extension of online aggregation techniques to a much broader collection of queries than could previously be handled.

Our analytic model suggests that in cases where memory overflows, the parallel hash ripple join algorithm is as much as a factor of five faster than the previously proposed hash ripple join algorithm (which degenerates to the block ripple join algorithm) in the uniprocessor environment. Furthermore, our implementation in a parallel DBMS shows that the parallel hash ripple join algorithm is able to use multiple processors to extend the speedup and scale-up properties of the traditional parallel hybrid hash join algorithm to the realm of online aggregation.

To complement the new join algorithm, we extend the results in [HH99] to provide formulas for running estimates and associated confidence intervals that account for the complexities of sampling in a parallel environment. Such formulas are a crucial ingredient of an interactive user interface that permits early termination of queries when the approximate answer is sufficiently precise.

There is substantial scope for future work on parallel ripple joins. For example, the development of the confidence interval formulas in Section 2.4.1 used the assumption that each $|A_{ij}|$ and $|B_{ij}|$ is known precisely. When this not the case, it appears possible to essentially estimate each $|A_{ij}|$ and $|B_{ij}|$ on the fly. The required modification of the confidence interval formulas to reflect the resulting increase in uncertainty is quite complex. As another example, the original uniprocessor hash ripple join in [HH99] permits the relative sampling rates from the various input relations to adapt over time to the statistical properties of the data. The goal is to achieve sampling rates that optimally trade off decreases in confidence interval length against times between successive updates of the point estimate and confidence interval. Such adaptive sampling also is possible in the parallel processing context, but the implementation issues, cost models, statistical formulas, and optimization methods are much more complex. We intend to pursue these issues in future work.

Chapter 3: A Comparison of Three Methods for Join View Maintenance in Parallel RDBMS

3.1 Introduction

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. 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 3.2, 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 chapter, 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 [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. 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 [CM96]. 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 chapter is organized as follows. We discuss join views and the three join view maintenance approaches in Section 3.2. Section 3.3 investigates the performance of the three join view maintenance methods. We conclude in Section 3.4.

3.2 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 chapter, we assume a parallel RDBMS with *L* data server nodes at which both base relations and materialized views are stored.

3.2.1 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;
```

3.2.1.1 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 3.1a and 3.1b 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*.

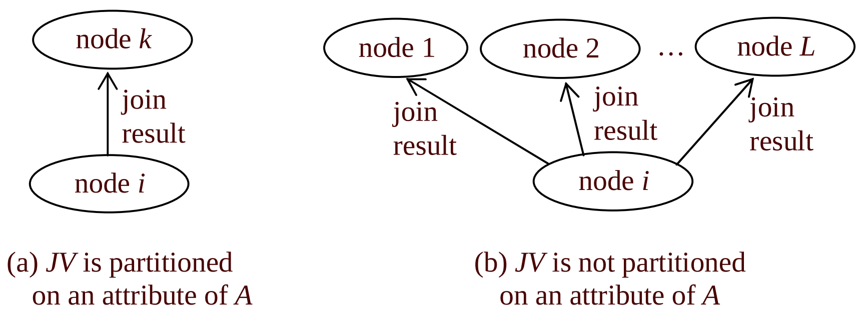


Figure 3.1 Naive method of maintaining a join view (case 1).

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 3.2a and 3.2b (see below) 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*.

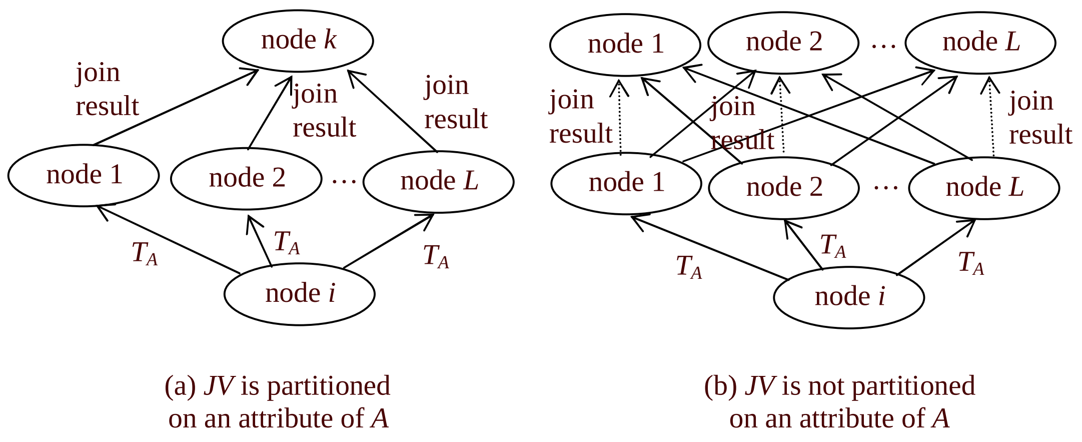


Figure 3.2 Naive method of maintaining a join view (case 2).

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.

3.2.1.2 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 3.3 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 3.2.1.3.

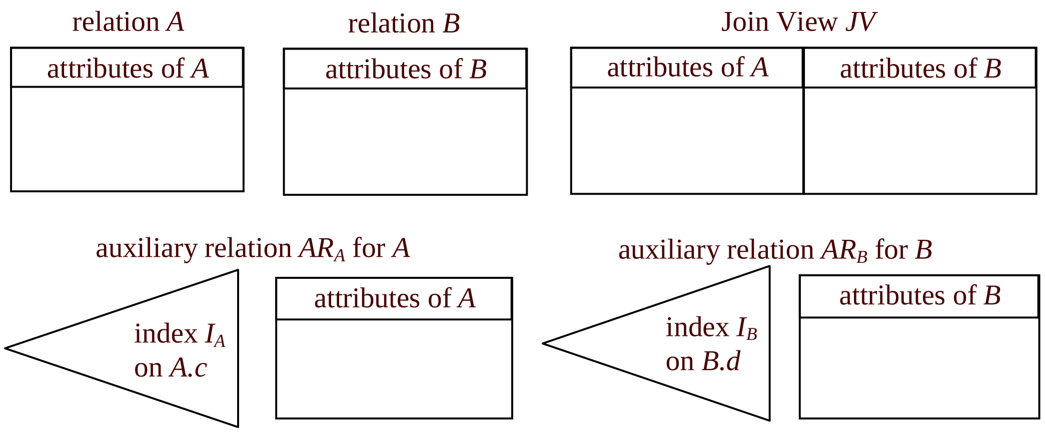


Figure 3.3 Base relations, auxiliary relations, and join view at a node of the parallel RDBMS.

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 3.4a and 3.4b show this procedure.

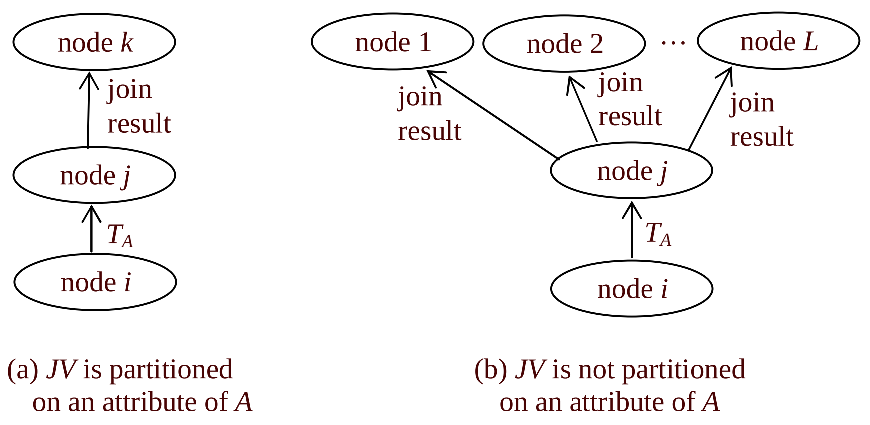


Figure 3.4 Maintaining a join view using auxiliary relations.

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*.

3.2.1.3 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 [QGM$^+$96], auxiliary views were proposed to make materialized views self-maintainable in a distributed data warehouse. [QGM$^+$96] also proposed a systematic algorithm to minimize the storage overhead of auxiliary views. Their techniques for reducing storage overhead can be used in our auxiliary relation method. These techniques also apply to the global index method discussed below in Section 3.2.1.4. The main idea in [QGM$^+$96] 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.

3.2.1.4 View Maintenance Using Global Indices

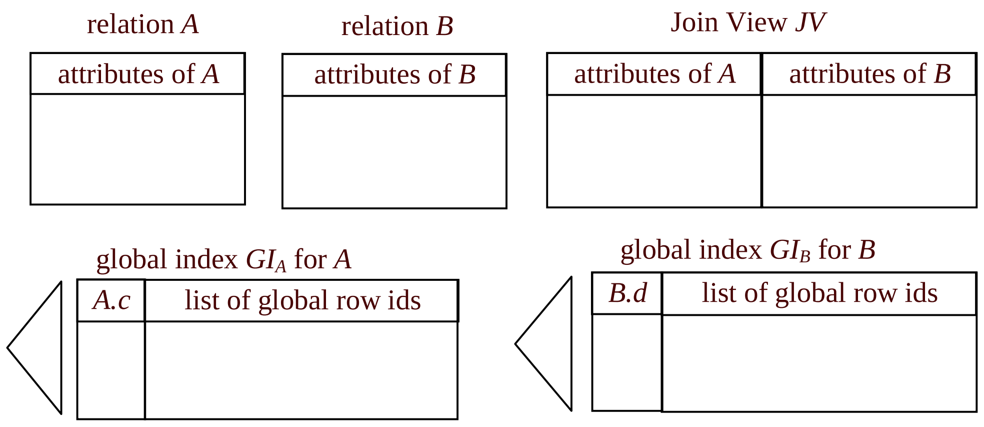
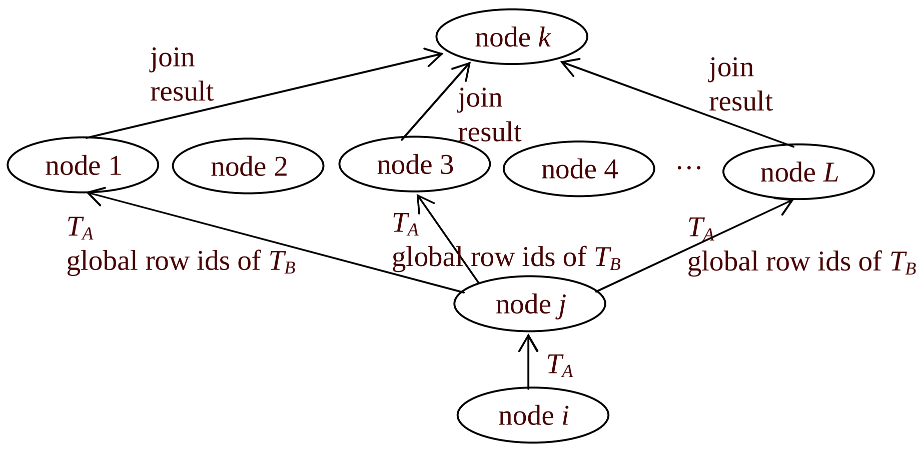


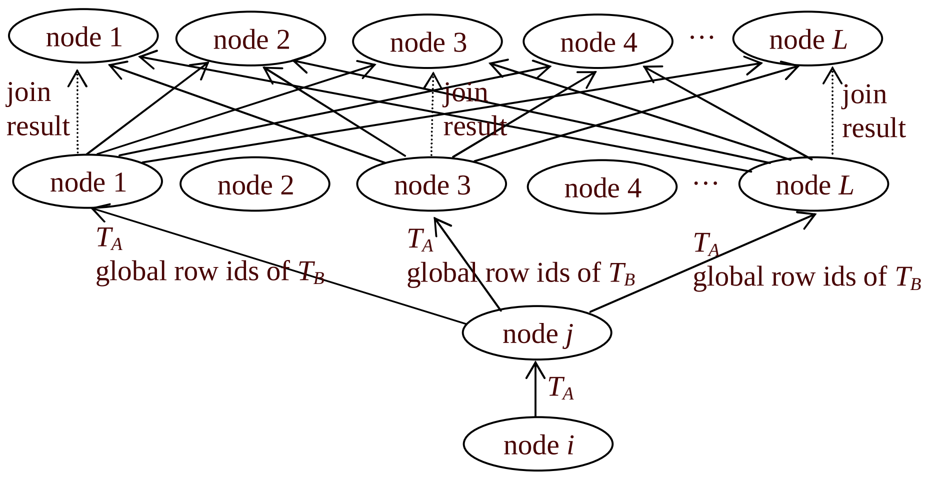
Figure 3.5 Base relations, global indices, and join view at a node of the parallel RDBMS.

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 3.5 (see above) shows the base relations, global indices, and join view at one 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 3.6a and 3.6b 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 3.6 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*.

3.2.2 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.

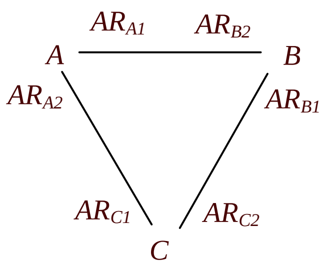
(3) If base relation C is updated, the same updates are propagated to the auxiliary relation AR_C. We use AR_{B2} and AR_A to maintain JV.

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 [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 $C \bowtie A$.



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.

3.3 Performance of Different Join View Maintenance Methods

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

3.3.1 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 3.3.3 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 3.2a, 3.4a, and 3.6a).

(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.

3.3.1.1 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 $K \leq min(N,L)$.

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.

3.3.1.2 Response Time

The model in Section 3.3.1.1 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.

3.3.2 Experiments with Analytical Model

Setting $||B||=6{,}400$, $M=10$, $N=10$ (except in Figure 3.8), and $K=min(N,L)$, we present in Figures 3.7 ~ 3.12 the resulting performance of the auxiliary relation method, the global index method, and the naive method of join view maintenance. Figure 3.7 (see below) shows *TW* for a single tuple insert 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.

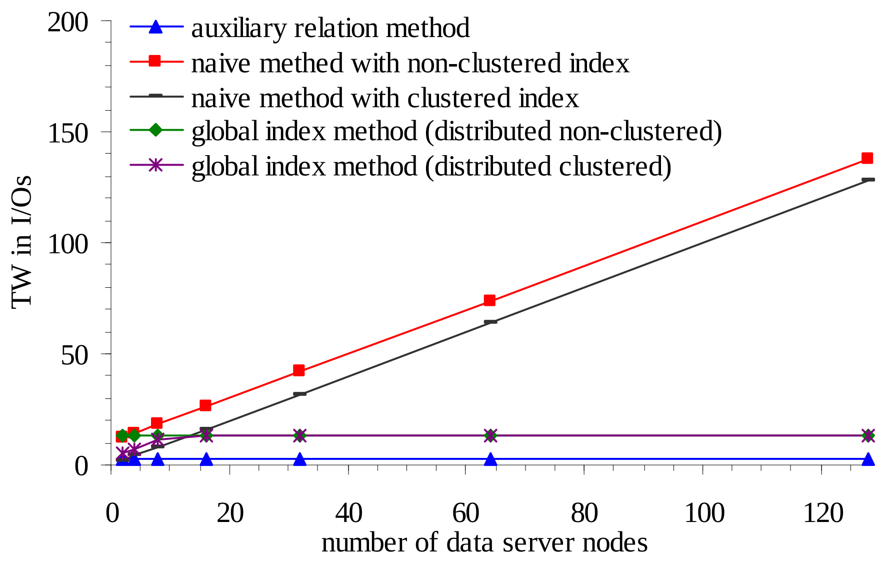


Figure 3.7 TW vs. number of data server nodes.

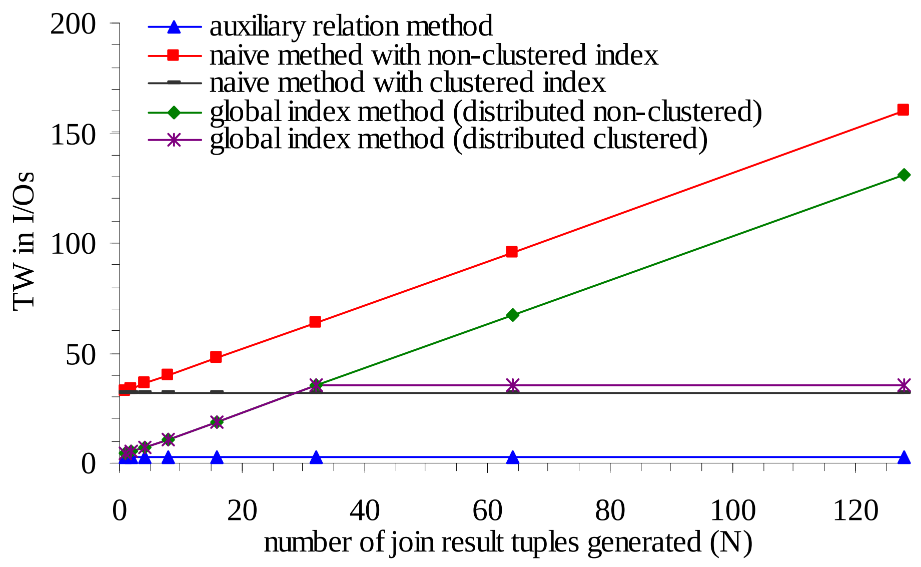


Figure 3.8 TW vs. number of join result tuples generated (L=32).

Figure 3.8 (see above) 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.

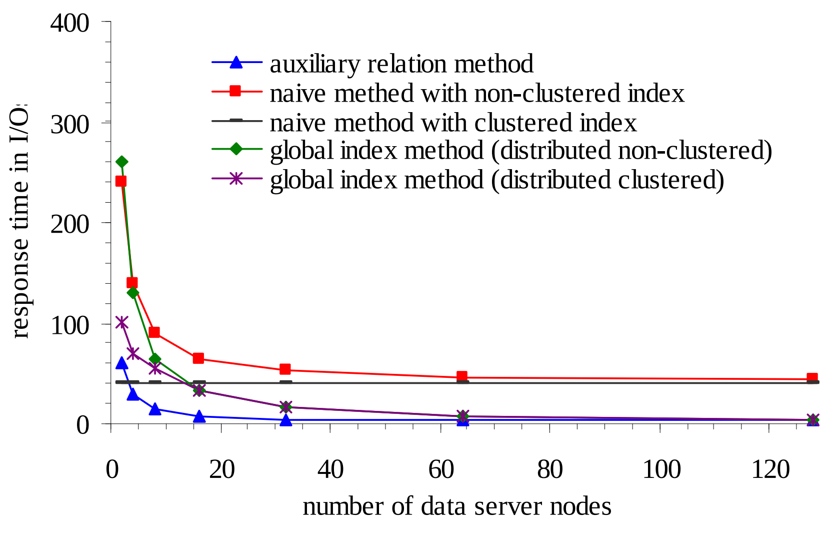


Figure 3.9 Execution time of one transaction with 40 tuples (index join).

Figure 3.9 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.

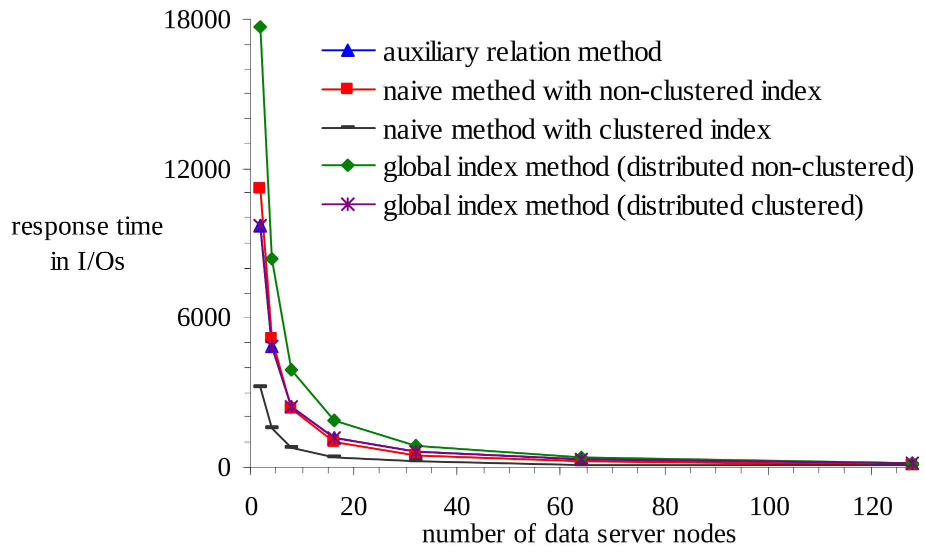


Figure 3.10 Execution time of one transaction with 6,500 tuples (sort merge join).

Figure 3.10 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."

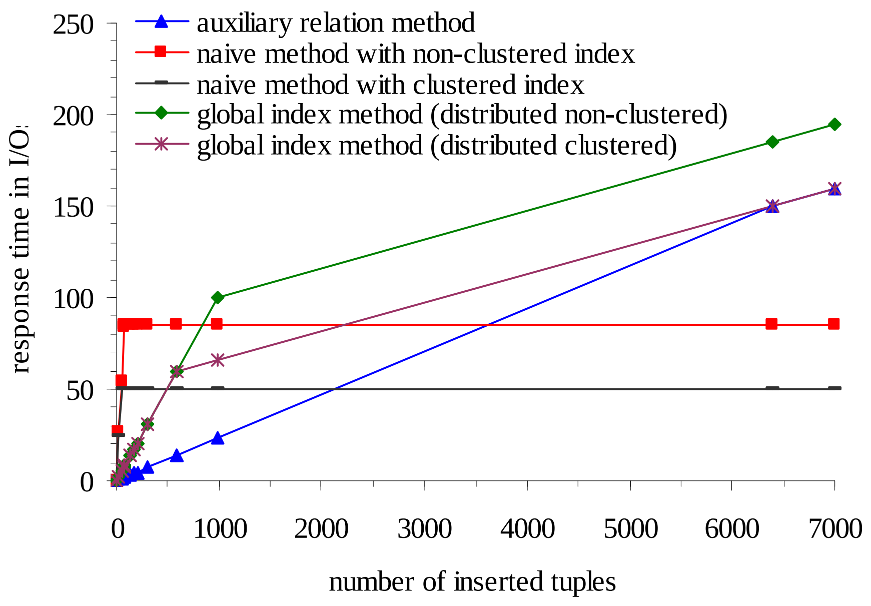


Figure 3.11 Execution time vs. tuples inserted (L=128).

Figure 3.11 (see above) 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.

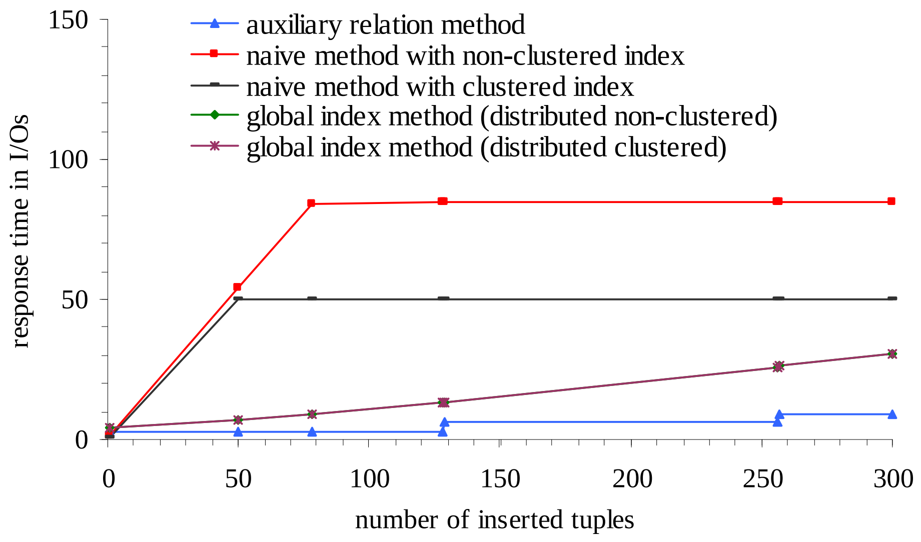


Figure 3.12 Execution time vs. tuples inserted - detail (L=128).

Figure 3.12 (see above) "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.

3.3.3 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 NCR WorldMark 4400 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 [TPC]:

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*.

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

Table 3.1 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 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. Before we ran each test, we restarted the computers to ensure a cold buffer pool.

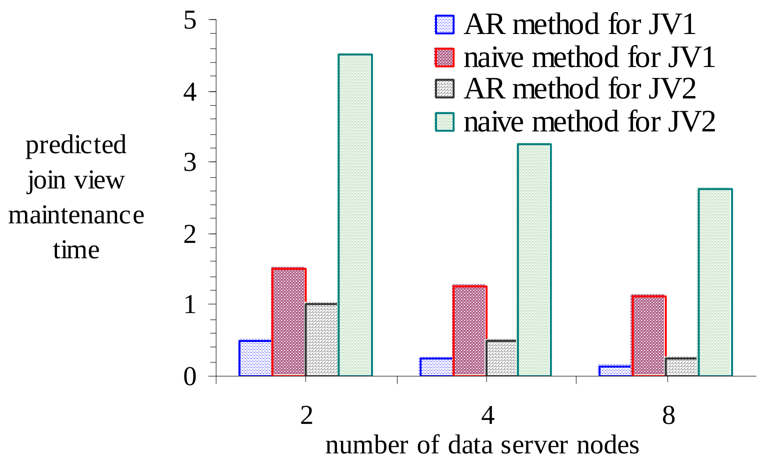


Figure 3.13 Predicted join view maintenance time.

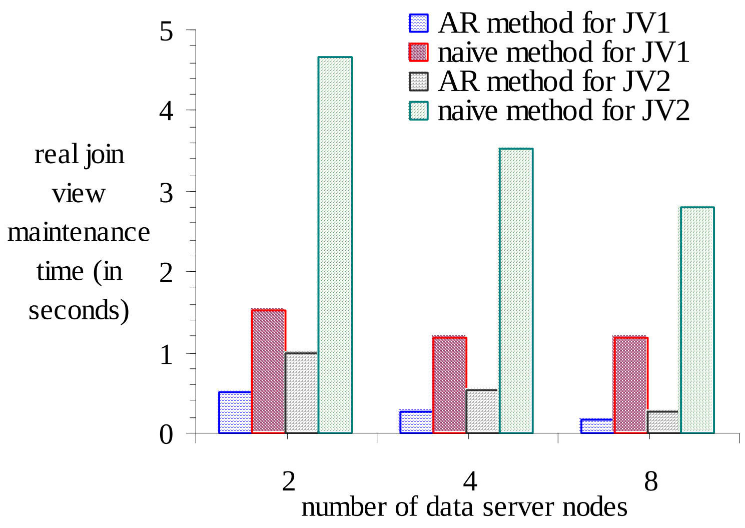


Figure 3.14 Measured join view maintenance time.

The join view maintenance time predicted by the analytical model is shown in Figure 3.13 (see above). All the numbers in Figure 3.13 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 3.14. Figures 3.13 and 3.14 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.

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.

3.4 Conclusion

This chapter 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 [QGM$^+$96] 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.

Chapter 4: Locking Protocols for Materialized Aggregate Join Views

4.1 Introduction

Although materialized view maintenance has been well-studied in the research literature [GM99], with rare exceptions, to date that published literature has ignored concurrency control. In fact, if we use generic concurrency control mechanisms, immediate materialized aggregate join view maintenance becomes extremely problematic — the addition of a materialized aggregate join view can introduce many lock conflicts and/or deadlocks that did not arise in the absence of this materialized view.

As an example of this effect, consider a scenario in which there are two base relations: the *lineitem* relation, and the *partsupp* relation, with the schemas *lineitem* (*orderkey*, *partkey*) (and possibly some other attributes), and *partsupp* (*partkey*, *suppkey*). Suppose that in transaction T_1 some customer buys items p_{11} and p_{12} in order o_1, which will cause the tuples (o_1, p_{11}) and (o_1, p_{12}) to be inserted into the *lineitem* relation. Also suppose that concurrently in transaction T_2 another customer buys items p_{21} and p_{22} in order o_2. This will cause the tuples (o_2, p_{21}) and (o_2, p_{22}) to be inserted into the *lineitem* relation. Suppose that parts p_{11} and p_{21} come from supplier s_1, while parts p_{12} and p_{22} come from supplier s_2. Then there are no lock conflicts nor is there any potential for deadlock between T_1 and T_2, since the tuples inserted by them are distinct.

Suppose now that we create a materialized aggregate join view *suppcount* to provide quick access to the number of parts ordered from each supplier, defined as follows:

```
create aggregate join view suppcount as
select p.suppkey, count(*)
```

from lineitem l, partsupp p

where l.partkey=p.partkey

group by p.suppkey;

Now both transactions T_1 and T_2 must update the materialized view *suppcount*. Since both T_1 and T_2 update the same pair of tuples in *suppcount* (the tuples for suppliers s_1 and s_2), there are now potential lock conflicts. To make things worse, suppose that T_1 and T_2 request their exclusive locks on *suppcount* in the following order:

(1) T_1 requests a lock for the tuple whose $suppkey=s_1$.

(2) T_2 requests a lock for the tuple whose $suppkey=s_2$.

(3) T_1 requests a lock for the tuple whose $suppkey=s_2$.

(4) T_2 requests a lock for the tuple whose $suppkey=s_1$.

Then a deadlock will occur.

The danger of this sort of deadlock is not necessarily remote. Suppose there are R suppliers, m concurrent transactions, and that each transaction represents a customer buying items randomly from r different suppliers. Then according to [GR93, page 428-429], if $mr<<R$, the probability that any particular transaction deadlocks is approximately $(m-1)(r-1)^4/(4R^2)$. (If we do not have $mr<<R$, then the probability of deadlock is essentially one. Hence, no matter whether $mr<<R$ or not, we can use a unified formula $min(1, (m-1)(r-1)^4/(4R^2))$ to roughly estimate the probability that any particular transaction deadlocks.) For reasonable values of R, m, and r, this probability of deadlock is unacceptably high. For example, if $R=3,000$, $m=8$, and $r=32$, the deadlock probability is approximately 18%. Merely doubling m to 16 raises this probability to 38%. In such a scenario large numbers of concurrent transactions will result in very high deadlock rates.

In view of this, one alternative is to simply avoid updating the materialized view within the transactions. Instead, we batch these updates to the materialized view and apply them later in separate transactions. This “works”; unfortunately, it requires that the system gives up on serializability and/or recency (it is possible to provide a theory of serializability in the presence of deferred updates if readers of the materialized view are allowed to read old versions of the view [KLM+97].) Giving up on serializability and/or recency for materialized views may ultimately turn out to be the best approach for any number of reasons; but before giving up altogether, it is worth investigating techniques that guarantee immediate update propagation with serializability semantics yet still give reasonable performance. Providing such guarantees is desirable in certain cases. (Such guarantees are required in the TPC-R benchmark [PF00], presumably as a reflection of some real world application demands.) In this chapter we explore techniques that can guarantee serializability without incurring high rates of deadlock and lock contention.

Our focus is materialized aggregate join views. In an extended relational algebra, a general instance of such a view can be expressed as $AJV=\gamma(\pi(\sigma(R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n)))$, where γ is the aggregate operator. SQL allows the aggregate operators *COUNT*, *SUM*, *AVG*, *MIN*, and *MAX*. However, because *MIN* and *MAX* cannot be maintained incrementally (the problem is deletes/updates — e.g., when the *MIN*/*MAX* value is deleted, we need to compute the new *MIN*/*MAX* value using all the values in the aggregate group [GKS01]), we restrict our attention to the three aggregate operators that make sense for materialized aggregates: *COUNT*, *SUM*, and *AVG*. Note that by letting $n=1$ in the definition of *AJV*, we also include aggregate views over single relations.

A useful observation is that for *COUNT*, *SUM*, and *AVG*, the updates to the materialized aggregate join views are associative and commutative, so it really does not matter in which order they are processed. In our running example, the state of *suppcount* after applying the updates of T_1 and T_2 is independent of the order in which they are applied. (Some care must be exercised to ensure that transactions that, unlike T_1 and T_2, are reading *suppcount* also see a consistent view of *suppcount*.)

This line of reasoning leads one to consider locking mechanisms that increase concurrency for commutative and associative operations.

Many special locking modes that support increased concurrency through the special treatment of “hot spot” aggregates in base relations [GK85, O86, Reu82] or by exploiting update semantics [BR92, RAA94] have been proposed. An early and particularly relevant example of locks that exploit update semantics was proposed by Korth [Kor83]. The basic idea is to identify classes of update transactions so that within each class, the updates are associative and commutative. For example, if a set of transactions update a record by adding various amounts to the same field in the record, they can be run in any order and the final state of the record will be the same, so they can be run concurrently. To ensure serializability, other transactions that read or write the record must conflict with these addition transactions. This insight is captured in Korth’s P locking protocol, in which addition transactions get P locks on the records they update through addition, while all other data accesses (including those by transactions not doing additive updates) are protected by standard S and X locks. P locks do not conflict with each other while they do conflict with S and X locks.

Borrowing this insight, we propose a V locking protocol (“V” for “View.”) In it, transactions that cause updates to materialized aggregate join views with associative and commutative aggregates (including *COUNT*, *SUM*, and *AVG*) get standard S and X locks on base relations but get V locks on the materialized view. V locks conflict with S and X locks but not with each other. At this level of discussion, V locks appear virtually identical to the (20+ year old!) P locks.

Unfortunately, there is a subtle difference between the problem solved by P locks and the materialized aggregate join view update problem. For P locks, the assumption is that updates are of two types: updates that modify existing tuples, which are handled by P locks; and updates that create new tuples or delete existing tuples, which are handled by X locks. At this level the same solution applies to updates of materialized aggregate join views. However, a transaction cannot know at the outset whether it will cause an update of an existing materialized view tuple, the insertion of a new tuple, or the deletion of an existing tuple. (Recall that the transaction inserts a tuple into a base relation and generates a new join result tuple, which only indirectly updates a materialized view tuple — the transaction does not know from the outset whether or not this new join result tuple will be aggregated into an existing materialized view tuple.) If we use X locks for the materialized view updates, we are back to our original problem of high lock conflict and deadlock rates. If we naively use our V locks for these updates, as we will show in Section 4.2, the semantics of the aggregate join view may be violated. In particular, it is possible that we could end up with what we call "split group duplicates" — multiple tuples in the aggregate join view for the same group. (Due to a similar reason, previous approaches for handling "hot spot" aggregates [GK85, O86, Reu82, BR92, RAA94] cannot be applied to materialized aggregate join views.)

To solve the split group duplicate problem, we augment V locks with a construct we call W locks. W locks are short-term locks. (The W lock sounds a lot like a latch, but it is not a latch; the split group duplicate problem arises even in the presence of latches. Furthermore, unlike latches, W locks must be considered in deadlock detection.) With W locks the semantics of materialized aggregate join views can be guaranteed — at any time, for any aggregate group, either zero or one tuple corresponding to this group exists in a materialized aggregate join view. Also, the probability of lock conflicts and deadlocks is greatly reduced, because W locks are short-term locks, and V locks do not conflict with each other or with W locks.

It is straightforward to implement V locks and W locks if the materialized view is stored without any indices or with hash indices. However, things become much more complex in the common case that there are B-tree indices over the materialized view. In this case, since the V lock is a form of a predicate lock, our first thought was to borrow from techniques that have been proposed for predicate locks. In particular, key-range locking (a limited form of predicate locking) on B-tree indices has been well-studied [Moh90a, Lom93]. However, we cannot simply use the techniques in [Moh90a, Lom93] to implement V and W key-range locks on B-tree indices. The reason is that V locks allow more concurrency than the exclusive locks considered in [Moh90a, Lom93], so during the period that a transaction T holds a V lock on an object, another transaction T' may delete this object by acquiring another V lock. To deal with this problem, we introduce a modified key-range locking strategy to implement V and W key-range locks on B-tree indices.

Other interesting properties of the V locking protocol exist because transactions getting V locks on materialized aggregate join views must get S and X locks on the base relations mentioned in their definition. The most interesting such property is that V locks can be used to support “direct propagate” updates to materialized views. Also, by considering the implications of the granularity of V locks and the interaction between base relation locks and accesses to the materialized view, we show that one can define a variant of the V locking protocol, the “no-lock” locking protocol, in which transactions do not set any long-term locks on the materialized view. Based on a similar reasoning, we show that the V locking protocol also applies to materialized non-aggregate join views.

The rest of the chapter is organized as follows. In Section 4.2, we explore the split group duplicate problem that arises with a naive use of V locks, and show how this problem can be avoided through the addition of W locks. In Section 4.3, we explore some thorny issues that arise when B-tree indices over the materialized views are considered. In Section 4.4, we explore the way V locks can be used to support both direct propagate updates and materialized non-aggregate join view maintenance. We also extend V locks to define a “no-lock” locking protocol. In Section 4.5, we give a correctness proof of the V+W locking protocol. In Section 4.6, we investigate the performance of the V locking protocol through a simulation study in a commercial RDBMS. We conclude in Section 4.7.

4.2 The Split Group Duplicate Problem

As mentioned in the introduction, we cannot simply use V locks on aggregate join views, even though the addition operation for the *COUNT*, *SUM*, and *AVG* aggregate operators in the view definitions is both commutative and associative. Recall that the problem is that for the V lock to work correctly, updates must be classified *a priori* into those that update a field in an existing tuple and those that create a new tuple or delete an existing tuple, which cannot be done in the view update scenario. In this section, we illustrate the split group duplicate problem that arises if we ignore this subtle difference between materialized view maintenance and the "traditional" associative/commutative update problems studied by Korth [Kor83] and others. First we illustrate the problem and its solution in the presence of hash indices or in the absence of indices on the materialized view. In Section 4.3, we consider the problem in the presence of B-tree indices (where its solution is considerably more complex.)

4.2.1 An Example of Split Groups

In this subsection, we explore an example of the split group duplicate problem in the case that the aggregate join view *AJV* is stored in a hash file implemented as described by Gray and Reuter [GR93]. (The case that the view is stored in a heap file is almost identical; just view the heap file as a hash file with one bucket.) Furthermore, suppose that we are using key-value locking. Suppose the schema of the aggregate join view *AJV* is (*a*, *sum(b)*), where attribute *a* is both the value locking attribute for the view and the hash key for the hash file. Suppose originally the aggregate join view *AJV* contains the tuple (20, 2) and several other tuples, but that there is no tuple whose attribute *a=1*.

Consider the following three transactions *T*, *T*′, and *T*″. Transaction *T* inserts a new tuple into a base relation *R* and this generates the join result tuple (1, 1), which needs to be integrated into *AJV*. Transaction *T*′ inserts another new tuple into the same base relation *R* and generates the join result tuple (1, 2). Transaction *T*″ deletes a third tuple from base relation *R*, which requires the tuple (20, 2) to be deleted from *AJV*. After executing these three transactions, the tuple (20, 2) should be deleted from *AJV* while the tuple (1, 3) should appear in *AJV*.

Now suppose that 20 and 1 have the same hash value so that the tuples (20, 2) and (1, 3) are stored in the same bucket *B* of the hash file. Also, suppose that initially there are four pages in bucket *B*: one bucket page P_1 and three overflow pages P_2, P_3, and P_4, as illustrated in Figure 4.1. Furthermore, let pages P_1, P_2, and P_3 be full while there are several open slots in page P_4.

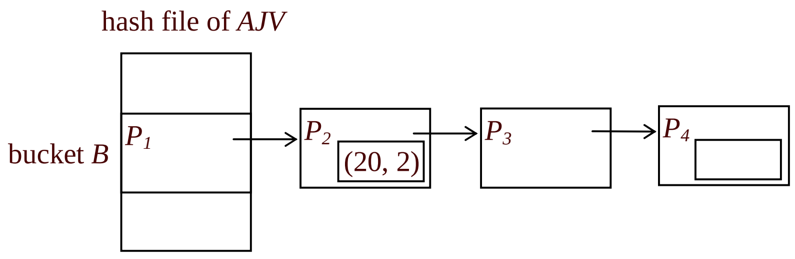


Figure 4.1 Hash file of the aggregate join view *AJV*.

To integrate a join result tuple t_1 into the aggregate join view *AJV*, a transaction *T* performs the following steps [GR93]:

1. Get an X value lock for $t_1.a$ on *AJV*. This value lock is held until transaction *T* commits/aborts.
2. Apply the hash function to $t_1.a$ to find the corresponding hash table bucket *B*.
3. Crab all the pages in bucket *B* to see whether a tuple t_2 whose attribute $a=t_1.a$ already exists. ("Crabbing" [GR93] means first getting an X semaphore on the next page, then releasing the X semaphore on the current page.)

4. If tuple t_2 exists in some page P in bucket B, stop the crabbing and integrate the join result tuple t_1 into tuple t_2. The X semaphore on page P is released only after the integration is finished.
5. If tuple t_2 does not exist, crab the pages in bucket B again to find a page P that has enough free space. Insert a new tuple into page P for the join result tuple t_1. The X semaphore on page P is released only after the insertion is finished.

Suppose now that we use V value locks instead of X value locks in this example and that the three transactions T, T', and T'' are executed in the following sequence:

1. First transaction T gets a V value lock for attribute $a=1$, applies the hash function to attribute $a=1$ to find the corresponding hash table bucket B, then crabs all the pages in bucket B to see whether a tuple t_2 whose attribute $a=1$ already exists in the hash file. After crabbing, it finds that no such tuple t_2 exists.
2. Next transaction T' gets a V value lock for attribute $a=1$, applies the hash function to attribute $a=1$ to find the corresponding hash table bucket B, and crabs all the pages in bucket B to see whether a tuple t_2 whose attribute $a=1$ already exists in the hash file. After crabbing, it finds that no such tuple t_2 exists.
3. Next, transaction T crabs the pages in bucket B again, finding that only page P_4 has enough free space. It then inserts a new tuple (1, 1) into page P_4 for the join result tuple (1, 1), commits, and releases the V value lock for attribute $a=1$.

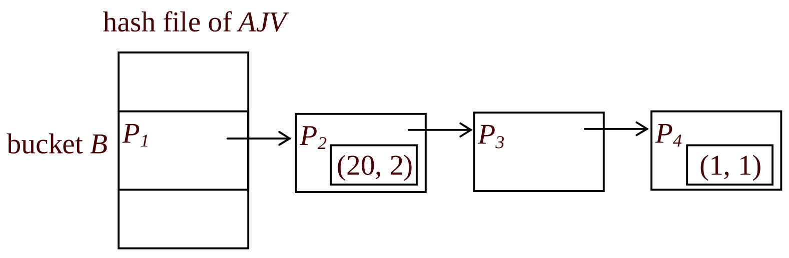


Figure 4.2 Hash file of the aggregate join view *AJV* – after inserting tuple (1, 1).

4. Then transaction T'' gets a V value lock for attribute $a=20$, finds that tuple (20, 2) is contained in page P_2, and deletes it (creating an open slot in page P_2). Then T'' commits, and releases the V value lock for attribute $a=20$.

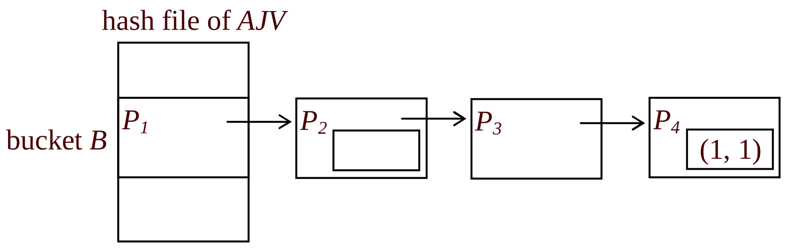


Figure 4.3 Hash file of the aggregate join view *AJV* – after deleting tuple (20, 2).

5. Finally, transaction T' crabs the pages in bucket B again, and finds that page P_2 has an open slot. It inserts a new tuple (1, 2) into page P_2 for the join result tuple (1, 2), commits, and releases the V value lock for attribute $a=1$.

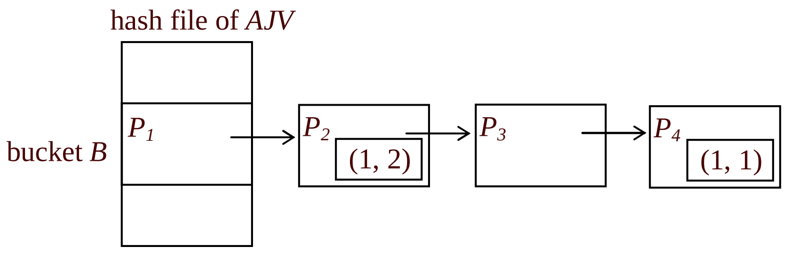


Figure 4.4 Hash file of the aggregate join view *AJV* – after inserting tuple (1, 2).

Now the aggregate join view *AJV* contains two tuples (1, 1) and (1, 2), whereas it should have only the single tuple (1, 3). This is why we call it the "split group duplicate" problem — the group for "1" has been split into two tuples.

One might think that during crabbing, holding an X semaphore on the entire bucket *B* could solve the split group duplicate problem. However, there may be multiple pages in the bucket *B* and some of them may not be in the buffer pool. Normally under all circumstances one tries to avoid performing I/O while holding a semaphore [GR93, page 849]. Hence, holding an X semaphore on the entire bucket for the duration of the operation could cause a substantial performance hit.

4.2.2 Preventing Split Groups with W Locks

4.2.2.1 The V+W Locking Protocol

To enable the use of high concurrency V locks while avoiding split group duplicates, we introduce a short-term lock mode, which we call the W lock mode, for aggregate join views. The W lock mode guarantees that for each aggregate group, at any time, at most one tuple corresponding to this group exists in the aggregate join view. With the addition of W locks we now have four kinds of elementary locks: S, X, V, and W.

The compatibilities among these locks are listed in Table 4.1, while the lock conversion lattice is shown in Figure 4.5 (see below). The W lock mode is only compatible with the V lock mode. A W lock can be either upgraded to an X lock or downgraded to a V lock. (In this respect the W lock is similar to the update mode lock [GR93], which can be either downgraded to an S lock or upgraded to an X lock.)

	V	S	X	W
V	yes	no	no	yes
S	no	yes	no	no
X	no	no	no	no
W	yes	no	no	no

Table 4.1 Compatibilities among the elementary locks.

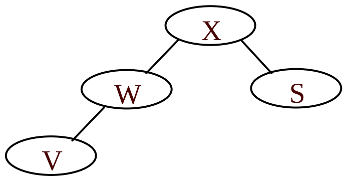


Figure 4.5 The lock conversion lattice of the elementary locks.

In the V+W locking protocol for materialized aggregate join views, S locks are used for reads, V and W locks are used for associative and commutative aggregate update writes, while X locks are used for transactions that do both reads and writes. These locks can be of any granularity, and, like traditional S and X locks, can be physical locks (e.g., tuple, page, or table locks) or value locks.

For fine granularity locks, there are multiple ways to define the corresponding coarser granularity intention locks as introduced in Gray et al. [GLP$^+$76]. In the following, we give one such definition, whose design criterion is to reduce the number of different kinds of intention locks as many as possible (e.g., we avoid introducing an SIW lock that can be downgraded to an SIV lock). Variations of this definition are straightforward.

We assume that W locks are only allowed at the finest granularity while V locks are allowed at all granularities. We define a coarse granularity IV lock corresponding to a fine granularity V lock. For a W lock at the finest granularity, we use IV (not IW) locks at coarser granularities. The IV lock is similar to the traditional IX lock except that it is compatible with the V lock. For a fine granularity X (S) lock, we use the traditional IX (IS) at coarser granularities. One can think that IX=IS+IV and X=S+V, as X locks are used for transactions that do both reads and writes. We introduce the SIV lock (S+IV) that is similar to the traditional SIX lock, i.e., the SIV lock is only compatible with the IS lock. Note that SIX=S+IX=S+(IS+IV)=(S+IS)+IV=S+IV=SIV, so we do not introduce the SIX lock, as it is the same as the SIV lock. Similarly, we introduce the VIS lock (V+IS) that is only compatible with the

IV lock. Note that VIX=V+IX=V+(IS+IV)=(V+IV)+IS=V+IS=VIS, so we do not introduce the VIX lock, as it is the same as the VIS lock.

The compatibilities among the coarse granularity locks are listed in Table 4.2, while the lock conversion lattice is shown in Figure 4.6. Since the behavior of intention locks is well understood, we do not discuss intention locks further in the rest of this chapter.

	V	S	X	IS	IV	IX	SIV	VIS
V	yes	no	no	no	yes	no	no	No
S	no	yes	no	yes	no	no	no	No
X	no	no	no	no	no	no	no	No
IS	no	yes	no	yes	yes	yes	yes	no
IV	yes	no	no	yes	yes	yes	no	yes
IX	no	no	no	yes	yes	yes	no	no
SIV	no	no	no	yes	no	no	no	no
VIS	no	no	no	no	yes	no	no	no

Table 4.2 Compatibilities among the coarse granularity locks.

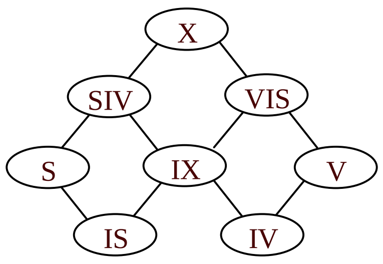


Figure 4.6 The lock conversion lattice of the coarse granularity locks.

4.2.2.2 Using W Locks

Transactions use W locks in the following way:

(1) To integrate a new join result tuple into an aggregate join view *AJV* (e.g., due to insertion into some base relation of *AJV*), we first put a short-term W lock on *AJV*. There are two special cases:

(a) If the same transaction has already put a V lock on *AJV*, this V lock is upgraded to the W lock.

(b) If the same transaction has already put an X lock on *AJV*, this W lock is unnecessary.

After integrating the new join result tuple into the aggregate join view *AJV*, we downgrade the short-term W lock to a long-term V lock that will be held until the transaction commits/aborts.

(2) To remove a join result tuple from the aggregate join view *AJV* (e.g., due to deletion from some base relation of *AJV*), we only need to put a V lock on *AJV*.

In this way, during aggregate join view maintenance, high concurrency is guaranteed by the fact that V locks are compatible with themselves. Note that when using V locks and W locks, multiple transactions may concurrently update the same tuple in the aggregate join view. Hence, logical undo is required on the aggregate join view *AJV* if the transaction updating *AJV* aborts.

The split group duplicate problem cannot occur if the system uses W locks. The reason is as follows. By enumerating all possible cases, we see that the split group duplicate problem will only occur under the following conditions: (1) two transactions integrate two new join result tuples into the aggregate join view *AJV* simultaneously, (2) these two join result tuples belong to the same aggregate group, and (3) no tuple corresponding to that aggregate group currently exists in the aggregate join view *AJV*. Using the short-term W lock, one transaction, say *T*, must do the update to the aggregate join view *AJV* first (by inserting a new tuple *t* with the corresponding group by attribute value into *AJV*). During the period that transaction *T* holds the short-term W lock, no other transaction can integrate another join result tuple that has the same group by attribute value as tuple *t* into the aggregate join view *AJV*. Then when a subsequent transaction *T*′ updates the view, it will see the existing tuple *t*. Hence, transaction *T*′ will aggregate its join result tuple that has the same group by attribute value as tuple *t* into tuple *t* (rather than inserting a new tuple into *AJV*).

As mentioned in the introduction, the W lock is similar in some respect to the latches that are used by DBMS to enforce serial updates to concurrently accessed data structures. However, there are some important differences. Unlike latches, W locks must be considered in deadlock detection, because although deadlocks are much less likely with W locks than with long-term X locks, they are still possible. Also, latches are orthogonal to the locking protocol in that they cannot be upgraded or downgraded to any locks (latches are either held or released.) Finally, and perhaps most importantly, the standard use of latches (short-term exclusion on updated data structures) will not prevent the split group duplicate problem efficiently.

We refer the reader to Section 4.5 for the correctness proof of the V+W locking protocol.

4.3 V and W Locks and B-Trees

In this section, we consider the particularly thorny problem of implementing V locks (with the required W locks) in the presence of B-tree indices. On B-tree indices, we use value locks to refer to key-range locks. To be consistent with the approach advocated by Mohan [Moh90a], we use next-key locking to implement key-range locking. We use “key” to refer to the indexed attribute of the B-tree index. We assume that the entry of the B-tree index is of the following format: (key value, row id list).

4.3.1 Split Groups and B-Trees

We begin by considering how split group duplicates can arise when a B-tree index is declared over the aggregate join view *AJV*. Suppose the schema of *AJV* is (*a*, *b*, *sum(c)*), and we build a B-tree index I_B on attribute *a*. Also, assume there is no tuple (1, 2, *X*) in *AJV*, for any *X*. Consider the following two transactions *T* and *T*′. Transaction *T* integrates a new join result tuple (1, 2, 3) into the aggregate join view *AJV* (by insertion into some base relation *R*). Transaction *T*′ integrates another new join result tuple (1, 2, 4) into the aggregate join view *AJV* (by insertion into the same base relation *R*). Using standard concurrency control without V locks, to integrate a join result tuple t_1 into the aggregate join view *AJV*, a transaction will execute something like the following operations:

(1) Get an X value lock for $t_1.a$ on the B-tree index I_B of *AJV*. This value lock is held until the transaction commits/aborts.

(2) Make a copy of the row id list in the entry for $t_1.a$ of the B-tree index I_B.

(3) For each row id in the row id list, fetch the corresponding tuple t_2. Check whether or not $t_2.a=t_1.a$ and $t_2.b=t_1.b$.

(4) If some tuple t_2 satisfies the condition $t_2.a=t_1.a$ and $t_2.b=t_1.b$, integrate tuple t_1 into tuple t_2 and stop.

(5) If no tuple t_2 satisfies the condition $t_2.a=t_1.a$ and $t_2.b=t_1.b$, insert a new tuple into *AJV* for tuple t_1. Also, insert the row id of this new tuple into the B-tree index I_B.

Suppose now we use V value locks instead of X value locks and the two transactions *T* and *T′* above are executed in the following sequence:

(1) Transaction *T* gets a V value lock for $a=1$ on the B-tree index I_B, searches the row id list in the entry for $a=1$, and finds that no tuple t_2 whose attributes $t_2.a=1$ and $t_2.b=2$ exists in *AJV*.

(2) Transaction *T′* gets a V value lock for $a=1$ on the B-tree index I_B, searches the row id list in the entry for $a=1$, and finds that no tuple t_2 whose attributes $t_2.a=1$ and $t_2.b=2$ exists in *AJV*.

(3) Transaction *T* inserts a new tuple t_1=(1, 2, 3) into *AJV*, and inserts the row id of tuple t_1 into the row id list in the entry for $a=1$ of the B-tree index I_B.

(4) Transaction *T′* inserts a new tuple t_3=(1, 2, 4) into *AJV*, and inserts the row id of tuple t_3 into the row id list in the entry for $a=1$ of the B-tree index I_B.

Now the aggregate join view *AJV* contains two tuples (1, 2, 3) and (1, 2, 4) instead of a single tuple (1, 2, 7); hence, we have the split group duplicate problem.

4.3.2 Implementing V Locking with B-trees

Implementing a high concurrency locking scheme in the presence of indices is difficult, especially if we consider issues of recoverability. Key-value locking as proposed by Mohan [Moh90a] was perhaps the first published description of the issues that arise and their solution. Unfortunately, we cannot directly use the techniques in [Moh90a] to implement V and W as value (key-range) locks.

To illustrate why, we use the following example. Suppose the schema of the aggregate join view *AJV* is (*a*, *sum(b)*), and a B-tree index is built on attribute *a* of the aggregate join view *AJV*. Suppose originally the aggregate join view *AJV* contains four tuples that correspond to *a=2*, *a=3*, *a=4*, and *a=5*. Consider the following three transactions *T*, *T′*, and *T″* that result in updates to the aggregate join view *AJV*. Transaction *T* deletes the tuple whose attribute *a=3* (by deletion from some base relation *R* of *AJV*). Transaction *T′* deletes the tuple whose attribute *a=4* (by deletion from the same base relation *R* of *AJV*). Transaction *T″* reads those tuples whose attribute *a* is between 2 and 5. Suppose we ignore the special properties of V locks and use the techniques in [Moh90a] to implement V and W value locks on the B-tree index. Then the three transactions *T*, *T′*, and *T″* could be executed in the following sequence:

(1) Transaction *T* puts a V lock for *a=3* and another V lock for *a=4* on the aggregate join view *AJV*.

	2	3	4	5
T		V	V	

(2) Transaction *T′* puts a V lock for *a=4* and another V lock for *a=5* on the aggregate join view *AJV*.

	2	3	4	5
T		V	V	
T′			V	V

(3) Transaction *T′* deletes the entry for *a=4* from the B-tree index. Transaction *T′* commits and releases the two V locks for *a=4* and *a=5*.

	2	3		5
T		V	V	

(4) Transaction *T* deletes the entry for *a=3* from the B-tree index.

	2			5
T		V	V	

(5) Before transaction *T* finishes execution, transaction *T″* finds the entries for *a=2* and *a=5* in the B-tree index. Transaction *T″* puts an S lock for *a=2* and another S lock for *a=5* on the aggregate join view *AJV*.

	2			5
T		V	V	
T″	S			S

In this way, transaction *T″* can start execution even before transaction *T* finishes execution. This is not correct (i.e., serializability can be violated), because there is a write-read conflict between transaction *T* and transaction *T″* (on the tuple whose attribute *a=3*). The main reason that this undesirable situation (transactions with write-read conflict can execute concurrently) occurs is due to the fact that V locks are compatible with themselves. Hence, during the period that a transaction holds a V lock on an object, another transaction may delete this object by acquiring another V lock.

To implement V and W value locks on B-tree indices correctly, we need to combine those techniques in [Moh90a, GR93] with the technique of logical deletion of keys [Moh90b, KMH97]. In Section 4.3.2.1, we describe the protocol for each of the basic B-tree operations in the presence of V locks. In Section 4.3.2.2, we explore the need for the techniques used in Section 4.3.2.1. We prove the correctness of the implementation method in Section 4.3.2.3.

4.3.2.1 Basic Operations for B-tree Indices

In our protocol, there are five operations of interest:

(1) **Fetch**: Fetch the row ids for a given key value v_1.

(2) **Fetch next**: Given the current key value v_1, find the next key value $v_2>v_1$ existing in the B-tree index, and fetch the row id(s) associated with key value v_2.

(3) **Put an X value lock on key value v_1.**

(4) **Put a V value lock on key value v_1.**

(5) **Put a W value lock on key value v_1.**

Unlike [Moh90a, GR93], we do not consider the operations of insert and delete. We show why this is by an example. Suppose a B-tree index is built on attribute *a* of an aggregate join view *AJV*. Assume we insert a tuple into some base relation of *AJV* and generate a new join result tuple *t*. The steps to integrate the join result tuple *t* into the aggregate join view *AJV* are as follows:

```
If the aggregate group of tuple t exists in AJV
        Update the aggregate group in AJV;
Else
        Insert a new aggregate group into AJV for tuple t;
```

Once again, we do not know whether we need to update an existing aggregate group in *AJV* or insert a new aggregate group into *AJV* until we read *AJV*. However, we do know that we need to acquire a W value lock on *t.a* before we can integrate tuple *t* into the aggregate join view *AJV*. Similarly, suppose we delete a tuple from some base relation of the aggregate join view *AJV*. We compute the corresponding join result tuples. For each such join result tuple *t*, we execute the following steps to remove tuple *t* from the aggregate join view *AJV*:

```
Find the aggregate group of tuple t in AJV;
Update the aggregate group in AJV;
```

```
        If all join result tuples have been removed from the aggregate group
            Delete the aggregate group from AJV;
```

In this case, we do not know whether we need to update an aggregate group in *AJV* or delete an aggregate group from *AJV* in advance. However, we do know that we need to acquire a V value lock on *t.a* before we can remove tuple *t* from the aggregate join view *AJV*.

The ARIES/KVL method described in [Moh90a] for implementing value locks on a B-tree index requires the insertion/deletion operation to be done immediately after a transaction gets appropriate locks. Also, in ARIES/KVL, the value lock implementation method is closely tied to the B-tree implementation method. This is because ARIES/KVL strives to take advantage of both IX locks and instant locks to increase concurrency. In the V+W locking mechanism, high concurrency has already been guaranteed by the fact that V locks are compatible with themselves.

We can exploit this advantage so that our method for implementing value locks for aggregate join views on B-tree indices is more general and flexible than the ARIES/KVL method. Specifically, in our method, after a transaction gets appropriate locks, we allow it to execute other operations before it executes the insertion/deletion/update/read operation. Also, our value lock implementation method is only loosely tied to the B-tree implementation method.

Our method for implementing value locks for aggregate join views on B-tree indices is as follows. Consider a transaction *T*.

Op1. Fetch: We first check whether some entry for value v_1 exists in the B-tree index. If such an entry exists, we put an S lock for value v_1 on the B-tree index. If no such entry exists, we find the smallest value v_2 in the B-tree index such that $v_2>v_1$. Then we put an S lock for value v_2 on the B-tree index.

Op2. Fetch next: We find the smallest value v_2 in the B-tree index such that $v_2>v_1$. Then we put an S lock for value v_2 on the B-tree index.

Op3. Put an X value lock on key value v_1: We first put an X lock for value v_1 on the B-tree index. Then we check whether some entry for value v_1 exists in the B-tree index. If no such entry exists, we find the smallest value v_2 in the B-tree index such that $v_2>v_1$. Then we put an X lock for value v_2 on the B-tree index.

Op4. Put a V value lock on key value v_1: We first check whether some entry for value v_1 exists in the B-tree index. If such an entry exists, we put a V lock for value v_1 on the B-tree index. If no entry for value v_1 exists, we find the smallest value v_2 in the B-tree index such that $v_2>v_1$. Then we put an X (not V) lock for value v_2 on the B-tree index.

Op5. Put a W value lock on key value v_1: We first put a W lock for value v_1 on the B-tree index. Then we check whether some entry for value v_1 exists in the B-tree index. If no entry for value v_1 exists, we do the following:

(a) Find the smallest value v_2 in the B-tree index such that $v_2>v_1$. Then we put a short-term W lock for value v_2 on the B-tree index. If the W lock for value v_2 on the B-tree index is acquired as an X lock, we upgrade the W lock for value v_1 on the B-tree index to an X lock. This situation may occur when transaction *T* already holds an S or X lock for value v_2 on the B-tree index.

(b) We insert into the B-tree index an entry for value v_1 with an empty row id list. Note: that at a later point transaction *T* will insert a row id into this row id list after transaction *T* inserts the corresponding tuple into the aggregate join view.

(c) We release the short-term W lock for value v_2 on the B-tree index.

Table 4.3 summarizes the locks acquired during different operations.

		current key v_1	next key v_2
fetch	v_1 exists	S	
	v_1 does not exist		S
fetch next			S
X value lock	v_1 exists	X	
	v_1 does not exist	X	X
V value lock	v_1 exists	V	
	v_1 does not exist		X
W value lock	v_1 exists	W	
	v_1 does not exist and the W lock on v_2 is acquired as a W lock	W	W
	v_1 does not exist and the W lock on v_2 is acquired as an X lock	X	X

Table 4.3 Summary of locking.

During the period that a transaction *T* holds a V (or W, or X) value lock for value v_1 on the B-tree index, if transaction *T* wants to delete the entry for value v_1, transaction *T* needs to do a logical deletion of keys [Moh90b, KMH97] instead of a physical deletion. That is, instead of removing the entry for value v_1 from the B-tree index, it is left there with a *delete_flag* set to 1. If the delete were to be rolled back, then the *delete_flag* is reset to 0. If another transaction inserts an entry for value v_1 into the B-tree index before the entry for value v_1 is garbage collected, the *delete_flag* of the entry for value v_1 is reset to 0. This is to avoid the potential write-read conflicts discussed at the beginning of Section 4.3.2.

The physical deletion operations are necessary, otherwise the B-tree index may grow unbounded. To leverage the overhead of the physical deletion operations, we perform them as garbage collection by other operations (of other transactions) that happen to pass through the affected nodes in the B-tree index [KMH97]. That is, a node reorganization operation checks all the entries in a leaf of the B-tree index and removes all such entries that have been marked deleted and currently have no locks on them. This can be implemented in the following way. We introduce a special short-term Z lock mode that is not compatible with any lock mode (including itself). No lock can be upgraded to a Z lock. A transaction *T* can get a Z lock on an object if no transaction (including transaction *T* itself) is currently holding any lock on this object. Also, during the period that transaction *T* holds a Z lock on an object, no transaction (including transaction *T* itself) can be granted another lock (including Z lock) on this object.

Note the Z lock mode is different from the X lock mode. For example, if transaction *T* itself is currently holding an S lock on an object, transaction *T* can still get an X lock on this object. That is, transaction *T* can get an X lock on an object if no other transaction is currently holding any lock on this object. For each entry with value *v* whose *delete_flag=1*, we request a conditional Z lock (conditional locks are discussed in [Moh90a]) for value *v*. If the conditional Z lock request is granted, we delete this entry from the leaf of the B-tree index, then we release the Z lock. If the conditional Z lock request is denied, we do not do anything with this entry. Then the physical deletion of this entry is left to other future operations.

We use the Z lock (instead of X lock) to prevent the following undesirable situation: a transaction that is currently using an entry (e.g., holding an S lock on the entry), where the entry is marked logically deleted, tries to physically delete this entry. Z locks can be implemented easily using the techniques in [GR93, Chapter 8] (by making small changes to the lock manager). Note the above method is different from the method described in [Moh90b] while both methods work. We choose the Z lock method to simplify our key-range locking protocol for aggregate join views on B-tree indices. As mentioned in [Moh90b], the log record for garbage collection is a redo-only log record.

In Op4 (put a V value lock on key value v_1), the situation that no entry for value v_1 exists in the B-tree index does not often occur. To illustrate this, consider an aggregate join view *AJV* that is defined on base relation *R* and several other base relations. Suppose a B-tree index I_B is built on attribute *d* of the aggregate join view *AJV*. If we insert a new tuple *t* into base relation *R* and generate several new join result tuples, we need to acquire appropriate W value locks on the B-tree index I_B before we can integrate these new join result tuples into the aggregate join view *AJV*. If we delete a tuple *t* from base relation *R*, to maintain the aggregate join view *AJV*, normally we need to first compute the corresponding join result tuples that are to be removed from the aggregate join view *AJV*. These join result tuples must have been integrated into the aggregate join view *AJV* before. Hence, when we acquire V value locks for their *d* attribute values, these *d* attribute values must exist in the B-tree index I_B.

However, there is an exception. Suppose attribute *d* of the aggregate join view *AJV* comes from base relation *R*. Consider the following scenario (see Section 4.4 below for details). There is only one tuple *t* in base relation *R* whose attribute $d=v$. However, there is no matching tuple in the other base relations of the aggregate join view *AJV* that can be joined with tuple *t*. Hence, there is no tuple in the aggregate join view *AJV* whose attribute $d=v$. Suppose transaction *T* executes the following SQL statement:

delete from *R* where *R.d*=*v*;

In this case, to maintain the aggregate join view *AJV*, there is no need for transaction *T* to compute the corresponding join result tuples that are to be removed from the aggregate join view *AJV*. Transaction *T* can execute the following "direct propagate" update operation:

delete from *AJV* where *AJV.d*=*v*;

Then when transaction *T* requests a V value lock for $d=v$ on the B-tree index I_B, transaction *T* will find that no entry for value *v* exists in the B-tree index I_B. We will return to direct propagate updates in Section 4.4.

4.3.2.2 Are These Techniques Necessary?

The preceding section is admittedly dense and intricate, so it is reasonable to ask if all this effort is really necessary. Unfortunately the answer appears to be yes — we use the following aggregate join view *AJV* to illustrate the rationale for the techniques introduced in the previous section. The schema of the aggregate join view *AJV* is (*a*, *sum(b)*). Suppose a B-tree index is built on attribute *a* of the aggregate join view *AJV*. We show that if any of the techniques from the previous section are omitted (and not replaced by other equivalent techniques), then we cannot guarantee serializability.

Technique 1. As mentioned above in Op4 (put a V value lock on key value v_1), we need to put an X lock (instead of a V lock) for value v_2 on the B-tree index. To illustrate why, we use the following example. Suppose originally the aggregate join view *AJV* contains only one tuple that corresponds to *a=4*. Consider the following three transactions *T*, *T′*, and *T″* on the aggregate join view *AJV*. Transaction *T* deletes the tuple whose attribute *a=2*. Transaction *T′* integrates two new join result tuples (2, 5) and (3, 6) into the aggregate join view *AJV*. Transaction *T″* reads those tuples whose attribute *a* is between 1 and 3. Suppose we put a V lock (instead of an X lock) for value v_2 on the B-tree index. Also, suppose the three transactions *T*, *T′*, and *T″* are executed in the following way:

(1) Transaction *T* finds the entry for *a=4* in the B-tree index. Transaction *T* puts a V lock for *a=4* on the aggregate join view *AJV*.

			4
T			V

(2) Transaction *T′* puts a W lock for *a=2* and another W lock for *a=4* on the aggregate join view *AJV*.

			4
T			V
T′	W		W

(3) Transaction T' inserts the tuple (2, 5) and an entry for $a=2$ into the aggregate join view *AJV* and the B-tree index, respectively. Transaction T' downgrades the two W locks for $a=2$ and $a=4$ on the aggregate join view *AJV* to V locks.

	2		4
T			V
T′	V		V

(4) Transaction T' puts a W lock for $a=3$ and another W lock for $a=4$ on the aggregate join view *AJV*.

	2		4
T			V
T′	V	W	W

(5) Transaction T' inserts the tuple (3, 6) and an entry for $a=3$ into the aggregate join view *AJV* and the B-tree index, respectively. Transaction T' downgrades the two W locks for $a=3$ and $a=4$ on the aggregate join view *AJV* to V locks.

	2	3	4
T			V
T′	V	V	V

(6) Transaction T' commits and releases the three V locks for $a=2$, $a=3$, and $a=4$.

	2	3	4
T			V

(7) Transaction *T* deletes the entry for $a=2$ from the B-tree index.

		3	4
T			V

(8) Before transaction *T* finishes execution, transaction T'' finds the entries for $a=3$ and $a=4$ in the B-tree index. Transaction T'' puts an S lock for $a=3$ on the aggregate join view *AJV*.

		3	4
T			V
T″		S	

In this way, transaction T'' can start execution even before transaction *T* finishes execution. This is not correct, because there is a write-read conflict between transaction *T* and transaction T'' (on the tuple whose attribute $a=2$).

Technique 2. As mentioned above in Op5 (put a W value lock on key value v_1), we need to put a W lock (instead of a V lock) for value v_2 on the B-tree index. To illustrate why, we use the following example. Suppose originally the aggregate join view *AJV* contains two tuples that correspond to *a=1* and *a=4*. Consider the following three transactions *T*, *T′*, and *T″* on the aggregate join view *AJV*. Transaction *T* integrates a new join result tuple (3, 5) into the aggregate join view *AJV*. Transaction *T′* integrates a new join result tuple (2, 6) into the aggregate join view *AJV*. Transaction *T″* reads those tuples whose attribute *a* is between 1 and 3. Suppose we put a V lock (instead of a W lock) for value v_2 on the B-tree index. Also, suppose the three transactions *T*, *T′*, and *T″* are executed in the following way:

(1) Transaction *T* puts a W lock for *a=3* and another V lock for *a=4* on the aggregate join view *AJV*.

	1			4
T			W	V

(2) Transaction *T′* finds the entries for *a=1* and *a=4* in the B-tree index. Transaction *T′* puts a W lock for *a=2* and another V lock for *a=4* on the aggregate join view *AJV*.

	1			4
T			W	V
T′		W		V

(3) Transaction *T* inserts the tuple (3, 5) and an entry for *a=3* into the aggregate join view *AJV* and the B-tree index, respectively.

	1		3	4
T			W	V
T′		W		V

(4) Transaction *T* commits and releases the W lock for *a=3* and the V lock for *a=4*.

	1		3	4
T′		W		V

(5) Before transaction *T′* inserts the entry for *a=2* into the B-tree index, transaction *T″* finds the entries for *a=1*, *a=3*, and *a=4* in the B-tree index. Transaction *T″* puts an S lock for *a=1* and another S lock for *a=3* on the aggregate join view *AJV*.

	1		3	4
T′		W		V
T″	S		S	

In this way, transaction *T″* can start execution even before transaction *T′* finishes execution. This is not correct, because there is a write-read conflict between transaction *T′* and transaction *T″* (on the tuple whose attribute *a=2*).

Technique 3. As mentioned above in Op5 (put a W value lock on key value v_1), if the W lock for value v_2 on the B-tree index is acquired as an X lock, we need to upgrade the W lock for value v_1 on the B-tree index to an X lock. To illustrate why, we use the following example. Suppose originally the aggregate join view *AJV* contains only one tuple that corresponds to *a=4*. Consider the following two transactions *T* and *T′* on the aggregate join view *AJV*. Transaction *T* first reads those tuples whose attribute *a* is between 1 and 4, then integrates a new join result tuple (3, 6) into the aggregate join view *AJV*. Transaction *T′* integrates a new join result tuple (2, 5) into the aggregate join view *AJV*. Suppose we do not upgrade the W lock for value v_1 on the B-tree index to an X lock. Also, suppose the two transactions *T* and *T′* are executed in the following way:

(1) Transaction *T* finds the entry for *a=4* in the B-tree index. Transaction *T* puts an S lock for *a=4* on the aggregate join view *AJV*. Transaction *T* reads the tuple in *AJV* whose attribute *a=4*.

			4
T			S

(2) Transaction *T* puts a W lock for *a=3* and another W lock for *a=4* on the aggregate join view *AJV*. Note the W lock for *a=4* is acquired as an X lock since transaction *T* has already put an S lock for *a=4* on the aggregate join view *AJV*.

			4
T		W	X

(3) Transaction *T* inserts the tuple (3, 6) and an entry for *a=3* into the aggregate join view *AJV* and the B-tree index, respectively. Then transaction *T* downgrades the W lock for *a=3* on the aggregate join view *AJV* to a V lock. Note transaction *T* does not downgrade the X lock for *a=4* on the aggregate join view *AJV* to a V lock.

		3	4
T		V	X

(4) Before transaction *T* finishes execution, transaction *T′* finds the entries for *a=3* and *a=4* in the B-tree index. Transaction *T′* puts a W lock for *a*=2 and another W lock for *a=3* on the aggregate join view *AJV*.

		3	4
T		V	X
T′	W	W	

In this way, transaction *T′* can start execution even before transaction *T* finishes execution. This is not correct, because there is a read-write conflict between transaction *T* and transaction *T′* (on the tuple whose attribute *a=2*).

Technique 4. As mentioned above in Op5 (put a W value lock on key value v_1), if no entry for value v_1 exists in the B-tree index, we need to insert an entry for value v_1 into the B-tree index. To illustrate why, we use the following example. Suppose originally the aggregate join view *AJV* contains two tuples that correspond to *a=1* and *a=5*. Consider the following three transactions *T*, *T′*, and *T″* on the aggregate join view *AJV*. Transaction *T* integrates two new join result tuples (4, 5) and (2, 6) into the aggregate join view *AJV*. Transaction *T′* integrates a new join result tuple (3, 7) into the aggregate join view *AJV*. Transaction *T″* reads those tuples whose attribute *a* is between 1 and 3. Suppose we do not insert an entry for value v_1 into the B-tree index. Also, suppose the three transactions *T*, *T′*, and *T″* are executed in the following way:

(1) Transaction *T* finds the entries for *a=1* and *a=5* in the B-tree index. For the new join result tuple (4, 5), transaction *T* puts a W lock for *a=4* and another W lock for *a=5* on the aggregate join view *AJV*.

	1				5
T				W	W

(2) Transaction *T* finds the entries for *a*=*1* and *a*=*5* in the B-tree index. For the new join result tuple (2, 6), transaction *T* puts a W lock for *a*=*2* and another W lock for *a*=*5* on the aggregate join view *AJV*.

	1				5
T		W		W	W

(3) Transaction *T* inserts the tuple (4, 6) and an entry for *a*=*4* into the aggregate join view *AJV* and the B-tree index, respectively. Then transaction *T* downgrades the W lock for *a*=*4* on the aggregate join view *AJV* to a V lock. Note transaction *T* still holds the W lock for *a*=*5* on the aggregate join view *AJV*, since transaction *T* has requested the W lock for *a*=*5* on the aggregate join view *AJV* twice.

	1			4	5
T		W		V	W

(4) Transaction *T′* finds the entries for *a*=*1*, *a*=*4*, and *a*=*5* in the B-tree index. Transaction *T′* puts a W lock for *a*=*3* and another W lock for *a*=*4* on the aggregate join view *AJV*.

	1			4	5
T		W		V	W
T′			W	W	

(5) Transaction *T′* inserts the tuple (3, 7) and an entry for *a*=*3* into the aggregate join view *AJV* and the B-tree index, respectively.

	1		3	4	5
T		W		V	W
T′			W	W	

(6) Transaction *T′* commits and releases the two W locks for *a*=*3* and *a*=*4*.

	1		3	4	5
T		W		V	W

(7) Before transaction *T* inserts the entry for *a*=*2* into the B-tree index, transaction *T″* finds the entries for *a*=*1*, *a*=*3*, *a*=*4*, and *a*=*5* in the B-tree index. Transaction *T″* puts an S lock for *a*=*1* and another S lock for *a*=*3* on the aggregate join view *AJV*.

	1		3	4	5
T		W		V	W
T″	S		S		

In this way, transaction *T″* can start execution even before transaction *T* finishes execution. This is not correct, because there is a write-read conflict between transaction *T* and transaction *T″* (on the tuple whose attribute *a*=*2*).

4.3.2.3 Correctness of the Key-range Locking Protocol

In this section, we prove the correctness (serializability) of our key-range locking strategy for aggregate join views on B-tree indices. Suppose a B-tree index I_B is built on attribute d of an aggregate join view *AJV*. To prove serializability, for any value v_1 (no matter whether or not an entry for value v_1 exists in the B-tree index, i.e., the phantom problem [GR93] is also considered), we only need to show that there is no read-write, write-read, or write-write conflict between two different transactions on those tuples of the aggregate join view *AJV* whose attribute d has value v_1 [BHG87, GR93]. As shown in [Kor83], write-write conflicts are avoided by the commutative and associative properties of the addition operation. Furthermore, the use of W locks guarantees that for each aggregate group, at any time at most one tuple corresponding to this group exists in the aggregate join view *AJV*. We enumerate all the possible cases to show that write-read and read-write conflicts do not exist. Since we use next-key locking, in the enumeration, we only need to focus on value v_1 and the smallest existing value v_2 in the B-tree index I_B such that $v_2>v_1$.

Consider the following two transactions T and T'. Transaction T updates (some of) the tuples in the aggregate join view *AJV* whose attribute d has value v_1. Transaction T' reads the tuples in the aggregate join view *AJV* whose attribute d has value v_1 (e.g., through a range query). Suppose v_2 is the smallest existing value in the B-tree index I_B such that $v_2>v_1$. Transaction T needs to get a V (or W, or X) value lock for $d=v_1$ on the B-tree index I_B of *AJV*. Transaction T' needs to get an S value lock for $d=v_1$ on the B-tree index I_B of *AJV*. There are four possible cases:

(1) Case 1: An entry E for value v_1 already exists in the B-tree index I_B. Also, transaction T' gets the S value lock for $d=v_1$ on the B-tree index I_B of *AJV* first.

To put an S value lock for $d=v_1$ on the B-tree index I_B, transaction T' needs to put an S lock for $d=v_1$ on *AJV*. During the period that transaction T' holds the S lock for $d=v_1$ on *AJV*, the entry E

for value v_1 always exists in the B-tree index I_B. Then during this period, transaction T cannot get the V (or W, or X) lock for $d=v_1$ on *AJV*. That is, transaction T cannot get the V (or W, or X) value lock for $d=v_1$ on the B-tree index I_B of *AJV*.

(2) Case 2: An entry E for value v_1 already exists in the B-tree index I_B. Also, transaction T gets a V (or W, or X) value lock for $d=v_1$ on the B-tree index I_B of *AJV* first.

To put a V (or W, or X) value lock for $d=v_1$ on the B-tree index I_B, transaction T needs to put a V (or W, or X) lock for $d=v_1$ on *AJV*. During the period that transaction T holds the V (or W, or X) lock for $d=v_1$ on *AJV*, the entry E for value v_1 always exists in the B-tree index I_B. Note during this period, if some transaction deletes the entry E for value v_1 from the B-tree index I_B, the entry E is only logically deleted. Only after transaction T releases the V (or W, or X) lock for $d=v_1$ on *AJV* may the entry E for value v_1 be physically deleted from the B-tree index I_B. Hence, during the period that transaction T holds the V (or W, or X) lock for $d=v_1$ on *AJV*, transaction T' cannot get the S lock for $d=v_1$ on *AJV*. That is, transaction T' cannot get the S value lock for $d=v_1$ on the B-tree index I_B of *AJV*.

(3) Case 3: No entry for value v_1 exists in the B-tree index I_B. Also, transaction T' gets the S value lock for $d=v_1$ on the B-tree index I_B of *AJV* first.

To put an S value lock for $d=v_1$ on the B-tree index I_B, transaction T' needs to put an S lock for $d=v_2$ on *AJV*. During the period that transaction T' holds the S lock for $d=v_2$ on *AJV*, no other transaction T'' can insert an entry for value v_3 into the B-tree index I_B such that $v_1 \leq v_3 < v_2$. This is because to do so, transaction T'' needs to get a W (or X) lock for $d=v_2$ on *AJV*. Then during the period that transaction T' holds the S lock for $d=v_2$ on *AJV*, transaction T cannot get the V (or W, or X) value lock for $d=v_1$ on the B-tree index I_B of *AJV*. This is because to do so, transaction T

needs to get an X (or W, or X) lock for $d=v_2$ on *AJV*. Note if transaction T' itself inserts an entry for value v_3 into the B-tree index I_B such that $v_1 \leq v_3 < v_2$, transaction T' will hold an X lock for $d=v_3$ on *AJV* (see how W and X value locks are implemented on the B-tree index in Section 4.3.2.1). Then transaction T still cannot get the V (or W, or X) value lock for $d=v_1$ on the B-tree index I_B of *AJV* before transaction T' finishes execution.

(4) Case 4: No entry for value v_1 exists in the B-tree index I_B. Also, transaction T gets the V (or W, or X) value lock for $d=v_1$ on the B-tree index I_B of *AJV* first.

In this case, there are three possible scenarios:

(a) Transaction T gets the V value lock for $d=v_1$ on the B-tree index I_B of *AJV* first. Hence, transaction T puts an X lock for $d=v_2$ on *AJV*. During the period that transaction T holds the X lock for $d=v_2$ on *AJV*, no other transaction T'' can insert an entry for value v_3 into the B-tree index I_B such that $v_1 \leq v_3 < v_2$. This is because to do so, transaction T'' needs to get a W (or X) lock for $d=v_2$ on *AJV*. Then during the period that transaction T holds the X lock for $d=v_2$ on *AJV*, transaction T' cannot get the S value lock for $d=v_1$ on the B-tree index I_B of *AJV*. This is because to do so, transaction T' needs to get an S lock for $d=v_2$ on *AJV*.

(b) Transaction T gets the W value lock for $d=v_1$ on the B-tree index I_B of *AJV* first. Hence, transaction T puts a W lock for $d=v_1$ and another W lock for $d=v_2$ on *AJV*. Also, transaction T inserts a new entry for value v_1 into the B-tree index I_B. Before transaction T inserts the new entry for value v_1 into the B-tree index I_B, transaction T holds the two W locks for $d=v_1$ and $d=v_2$ on *AJV*. During this period, no other transaction T'' can insert an entry for value v_3 into the B-tree index I_B such that $v_1 \leq v_3 < v_2$. This is because to do so, transaction T'' needs to get a W (or X) lock for $d=v_2$ on *AJV*. Then during this period, transaction T' cannot get the S value lock for $d=v_1$ on the B-tree index I_B of *AJV*. This is because to do so, transaction T' needs to get an S lock for $d=v_2$ on *AJV*. After transaction *T* inserts the new entry for value v_1 into the B-tree index I_B, transaction *T* will hold a V (or W) lock for $d=v_1$ on *AJV* until transaction *T* finishes execution. Then during this period, transaction *T′* still cannot get the S value lock for $d=v_1$ on the B-tree index I_B of *AJV*. This is because to do so, transaction *T′* needs to get an S lock for $d=v_1$ on *AJV*.

(c) Transaction *T* gets the X value lock for $d=v_1$ on the B-tree index I_B of *AJV* first. Hence, transaction *T* puts an X lock for $d=v_1$ and another X lock for $d=v_2$ on *AJV*. During the period that transaction *T* holds the two X locks for $d=v_1$ and $d=v_2$ on *AJV*, no other transaction *T″* can insert an entry for value v_3 into the B-tree index I_B such that $v_1 \le v_3 < v_2$. This is because to do so, transaction *T″* needs to get a W (or X) lock for $d=v_2$ on *AJV*. Then during the period that transaction *T* holds the two X locks for $d=v_1$ and $d=v_2$ on *AJV*, transaction *T′* cannot get the S value lock for $d=v_1$ on the B-tree index I_B of *AJV*. This is because to do so, depending on whether transaction *T* has inserted a new entry for value v_1 into the B-tree index I_B or not, transaction *T′* needs to get an S lock for either $d=v_1$ or $d=v_2$ on *AJV*.

In the above three scenarios, the situation that transaction *T* itself inserts an entry for value v_3 into the B-tree index I_B such that $v_1 \le v_3 < v_2$ can be discussed in a way similar to Case 3.

Hence, for any value v_1, there is no read-write or write-read conflict between two different transactions on those tuples of the aggregate join view *AJV* whose attribute *d* has value v_1. As discussed at the beginning of this section, write-write conflicts do not exist and thus our key-range locking protocol guarantees serializability.

4.4 Other Uses and Extensions of V Locks

In this section we briefly discuss three other interesting aspects of using V locks for materialized view maintenance. In Section 4.4.1, we discuss the possibility of supporting direct propagate updates. In Section 4.4.2, we show how observations about the appropriate granularity of V locks illustrate the possibility of a locking protocol for materialized views that supports serializability without requiring any long-term locks whatsoever on the views. In Section 4.4.3, we describe how to apply the V locking protocol to non-aggregate join views.

4.4.1 Direct Propagate Updates

In the preceding sections of this chapter, with one exception at the end of Section 4.3.2.1, we have assumed that materialized aggregate join views are maintained by first computing the join of the newly updated (inserted, deleted) tuples with the other base relations, then aggregating these join result tuples into the aggregate join view. In this section we will refer to this approach as the "indirect approach" to updating the materialized view. However, in certain situations, it is possible to propagate updates on base relations directly to the materialized view, without computing any join. As we know of at least one commercial system (Teradata) that supports such direct propagate updates, in this section we investigate how they can be handled in our framework.

Direct propagate updates are perhaps most useful in the case of (non-aggregate) join views, so we consider join views in the following discussion. (Technically, we do not need to mention the distinction between join views and aggregate join views, since non-aggregate join views are really included in our general class of views – recall that we are considering views $AJV=\gamma(\pi(\sigma(R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n)))$. If the aggregate operator γ in this formula has the effect of putting every tuple of the enclosed project-select-join in its own group, then what we have is really a non-aggregate join view.) However, the same discussion holds for direct propagate updates to aggregate join views.

Our focus in this chapter is not to explore the merits of direct propagate updates or when they apply; rather, it is to see how they can be accommodated by the V locking protocol. We begin with an example. Suppose we have two base relations, *A*(*a*, *b*, *c*) and *B*(*d*, *e*, *f*). Consider the following join view:

create join view *JV* as

select *A*.*a*, *A*.*b*, *B*.*e*, *B*.*f* from *A*, *B* where *A*.*c*=*B*.*d*;

Next consider a transaction *T* that executes the following SQL statement:

update *A* set *A*.*b*=2 where *A*.*a*=1;

To maintain the join view, transaction *T* only needs to execute the following operation (without performing a join with base relation *B*):

update *JV* set *JV*.*b*=2 where *JV*.*a*=1;

This is a “direct propagate” update, since transaction *T* does not compute a join to maintain the view. Similarly, suppose that a transaction *T*′ executes the following SQL statement:

update *B* set *B*.*e*=4 where *B*.*f*=3;

To maintain *JV*, transaction *T*′ can also do a direct propagate update with the following operation:

update *JV* set *JV*.*e*=4 where *JV*.*f*=3;

If these transactions naively use V locks on the materialized view, there is apparently a problem: since two V locks do not conflict, *T* and *T*′ can execute concurrently. This is not correct, since there is a write-write conflict between *T* and *T*′ on any tuple in *JV* with *a=1* and *f=3*. This could lead to a non-serializable schedule.

One way to prevent this would be to require all direct propagate updates to get X locks on the materialized view tuples that they update while indirect updates still use V locks. While this is correct, it is also possible to use V locks for the direct updates if we require that transactions that update base relations in materialized view definitions get X locks on the tuples in the base relations they update and S locks on the other base relations mentioned in the view definition. Note that these are exactly the locks the transactions would acquire if they were using indirect materialized view updates instead of direct propagate updates.

Informally, this approach with V locks works because updates to materialized views (even direct propagate updates) are not arbitrary; rather, they must be preceded by updates to base relations. So if two transactions using V locks would conflict in the join view on some tuple *t*, they must conflict on one or more of the base relations updated by the transactions, and locks at that level will resolve the conflict.

In our running example, *T* and *T*′ would conflict on base relation *A* (since *T* must get an X lock and *T*′ must get an S lock on the same tuples in *A*) and/or on base relation *B* (since *T* must get an S lock and *T*′ must get an X lock on the same tuples in *B*.) Note that these locks could be tuple-level, or table-level, or anything in between, depending on the specifics of the implementation. A formal complete correctness proof of this approach can be done easily by making minor changes to the proof in Section 4.5.

Unlike the situation for indirect updates to materialized aggregate join views, for direct propagate updates the V lock will not result in increased concurrency over X locks. Our point here is to show that we do not need special locking techniques to handle direct propagate updates: the transactions obtain locks as if they were doing updates indirectly (X locks on the base relations they update, S locks on the base relations with which they join, and V locks on the materialized view.) Then the transactions can use either update approach (direct or indirect) and still be guaranteed of serializability.

4.4.2 Granularity and the No-Lock Locking Protocol

Throughout the discussion in this chapter we have been purposely vague about the granularity of locking. This is because the V lock can be implemented at any granularity; the appropriate granularity is a question of efficiency, not of correctness. V locks have some interesting properties with respect to granularity and concurrency, which we explore in this section.

In general, finer granularity locking results in higher concurrency. This is not true of V locks if we consider only transactions that update the materialized views. The reason is that V locks do not conflict with one another, so that a single table-level V lock on a materialized view is the same, with respect to concurrency of update transactions, as many tuple-level V locks on the materialized view.

This is not to say that a single table-level V lock per materialized view is a good idea; indeed, a single table-level V lock will block all readers of the materialized view (since it looks like an X lock to any transaction other than an updater also getting a V lock.) Finer granularity V locks will let readers of the materialized view proceed concurrently with updaters (if, for example, they read tuples that are not being updated.) In a sense, a single V lock on the view merely signals "this materialized view is being updated;" read transactions "notice" this signal when they try to place S locks on the view.

This intuition can be generalized to produce a protocol for materialized views that requires no long-term locks at all on the materialized views. In this protocol, the function provided by the V lock on the materialized view (letting readers know that the view is being updated) is implemented by X locks on the base relations. The observation that limited locking is possible when data access patterns are constrained was exploited in a very different context (locking protocols for hierarchical database systems) in [SK80].

In the no-lock locking protocol, like the V locking protocol, updaters of the materialized view must get X locks on the base relations they update and S locks on other base relations mentioned in the view. To interact appropriately with updaters, readers of the materialized view are required to get S locks on all the base relations mentioned in the view. If the materialized view is being updated, there must be an X lock on one of the base relations involved, so the reader will block on this lock. Updaters of the materialized view need not get V locks on the materialized view (since only they would be obtaining locks on the view, and they do not conflict with each other), although they do require short-term W locks to avoid the split group duplicate problem.

It seems unlikely that in a practical situation this no-lock locking protocol would yield higher performance than the V locking protocol. The no-lock locking protocol benefits updaters (who do not have to get V locks) at the expense of readers (who have to get multiple S locks.) However, we present it here as an interesting application of how the semantics of materialized view updates can be exploited to reduce locking while still guaranteeing serializability.

4.4.3 Applying the V Locking Protocol to Non-aggregate Join Views

Besides aggregate join views, the V locking protocol also applies to (non-aggregate) join views of the form $JV=\pi(\sigma(R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n))$. In fact, for join views, only V locks are necessary. W locks are no longer needed unless we need to implement key-range locking for join views on B-tree indices (see below). This is due to the following reasons:

(1) As discussed in Section 4.4.1, updates to materialized views must be preceded by updates to base relations. So if two transactions using V locks would conflict in the join view on some tuple *t*, they must conflict on one or more of the base relations updated by the transactions, and locks at that level will resolve the conflict.

(2) The split group duplicate problem does not exist on join views.

We refer the reader to Section 4.5 for a formal complete correctness proof of this approach.

In a practical situation, for a join view *JV*, the V locking protocol is not likely to yield higher performance than the traditional X locking protocol, unless the join view *JV* contains a large number of duplicate tuples (e.g., due to projection). This is because a join view with a large number of duplicate tuples behaves much like an aggregate join view with a few tuples, as duplicate tuples are hard to differentiate [LYC+00]. This effect is clearer from the correctness proof in Section 4.5.2.

Implementing the V locking protocol for join views in the presence of B-tree indices is tricky. For example, suppose we do not use W locks on join views. That is, we only use S, X, and V value locks on join views. Suppose we implement S, X, and V value locks for join views on B-tree indices in the same way as described in Section 4.3.2.1. Also, suppose a B-tree index is built on attribute *a* of a join view *JV*. Then to insert a new join result tuple *t* into the join view *JV*, we need to first put a V value lock for *t.a* on the B-tree index. If no entry for *t.a* exists in the B-tree index, we need to find the smallest value v_2 in the B-tree index such that $v_2>t.a$ and put an X lock for value v_2 on the B-tree index. Unlike the W lock, the X lock for value v_2 on the B-tree index cannot be downgraded to a V lock. Hence, this X lock greatly reduces concurrency. However, we cannot replace the X lock for value v_2 on the B-tree index by a V lock.

To illustrate why, we use the following example. Suppose the schema of the join view *JV* is (*a*, *b*), and a B-tree index is built on attribute *a* of the join view *JV*. Suppose originally the join view *JV* contains two tuples (1, 7) and (4, 8). Consider the following three transactions *T*, *T′*, and *T″* on the join view *JV*. Transaction *T* inserts a new join result tuple (2, 5) into the join view *JV*. Transaction *T′* inserts a new join result tuple (3, 6) into the join view *JV*. Transaction *T″* reads those tuples whose attribute *a* is between 1 and 3. Suppose we replace the X lock for value v_2 on the B-tree index by a V lock. Also, suppose the three transactions *T*, *T′*, and *T″* are executed in the following way:

(1) Transaction *T* puts a V lock for *a=2* and another V lock for *a=4* on the join view *JV*.

	1			4
T		V		V

(2) Transaction *T*′ finds the entries for *a=1* and *a=4* in the B-tree index. Transaction *T*′ puts a V lock for *a=3* and another V lock for *a=4* on the join view *JV*.

	1			4
T		V		V
T′			V	V

(3) Transaction *T*′ inserts the tuple (3, 6) and an entry for *a=3* into the join view *JV* and the B-tree index, respectively.

	1		3	4
T		V		V
T′			V	V

(4) Transaction *T*′ commits and releases the two V locks for *a=3* and *a=4*.

	1		3	4
T		V		V

(5) Before transaction *T* inserts the entry for *a=2* into the B-tree index, transaction *T*″ finds the entries for *a=1*, *a=3*, and *a=4* in the B-tree index. Transaction *T*″ puts an S lock for *a=1* and another S lock for *a=3* on the join view *JV*.

	1		3	4
T		V		V
T″	S		S	

In this way, transaction *T*″ can start execution even before transaction *T* finishes execution. This is not correct (i.e., serializability can be violated), because there is a write-read conflict between transaction *T* and transaction *T*″ (on the tuple (2, 6)).

To implement value locks for join views on B-tree indices with high concurrency, we can utilize the W value lock mode and treat join views in the same way as aggregate join views. For join views, we still use four kinds of value locks: S, X, V, and W. For example, suppose a B-tree index is built on attribute *a* of a join view *JV*. As described in Section 4.2.2.2, to insert a new join result tuple *t* into the join view *JV*, we first put a W value lock for *t.a* on the B-tree index. After the new join result tuple *t* and its row id are inserted into the join view *JV* and the B-tree index, respectively, we downgrade the

W value lock for *t.a* to a V value lock. To delete a join result tuple *t* from the join view *JV*, we first put a V value lock for *t.a* on the B-tree index. For join views, all the four different kinds of value locks (S, X, V, and W) can be implemented on B-tree indices in the same way as described in Section 4.3.2.1. Or, the W value lock mode can be implemented on B-tree indices in a different way from what is described in Section 4.3.2.1. For example, consider the case that no entry for value v_1 exists in the B-tree index. After an entry for value v_1 with an empty row id list is inserted into the B-tree index, we can downgrade the W lock for value v_1 on the B-tree index to a V lock immediately. The correctness (serializability) of the implementation can be proved in a way similar to that described in Section 4.3.2.3. Note here, for join views, the W value lock mode is used for a different purpose from that for aggregate join views.

4.5 Correctness of the V+W Locking Protocol

In this section, we prove the correctness of the V(+W) locking protocol. The intuition for this proof relies on the fact that if two transactions updating the base relations of a join view *JV* have no lock conflict with each other on the base relations of *JV*, they must generate different join result tuples. Additionally, the addition operation for the *SUM*, *COUNT*, and *AVG* aggregate operators is both commutative and associative.

We begin by reviewing our assumptions. We assume that an aggregate join view *AJV* is maintained in the following way: first compute the join result tuple(s) resulting from the update(s) to the base relation(s) of *AJV*, then integrate these join result tuple(s) into *AJV*. During aggregate join view maintenance, we put appropriate locks on all the base relations of the aggregate join view (i.e., X locks on the base relations updated and S locks on the other base relations mentioned in the view definition). We use strict two-phase locking (except for W locks). We assume that the locking mechanism used by the database system on the base relations ensures serializability in the absence of aggregate join views. Unless otherwise specified, all the locks are long-term locks that are held until transaction commits. Transactions updating the aggregate join view obtain V and W locks as described in the V+W locking protocol. We make the same assumptions for non-aggregate join views.

We first prove serializability in Section 4.5.1 for the simple case where projection does not appear in the join view definition, i.e., $JV=\sigma(R_1 \bowtie \ldots \bowtie R_i \bowtie \ldots \bowtie R_n)$. In Section 4.5.2, we prove serializability for the general case where projection appears in the join view definition, i.e., $JV=\pi(\sigma(R_1 \bowtie \ldots \bowtie R_i \bowtie \ldots \bowtie R_n))$. In Section 4.5.3, we prove serializability for the case with aggregate join views $AJV=\gamma(\pi(\sigma(R_1 \bowtie \ldots \bowtie R_i \bowtie \ldots \bowtie R_n)))$, where γ is one of *COUNT*, *SUM*, or *AVG*. The proof in Sections 4.5.2 and 4.5.3 relies on the serializability result we get in Section 4.5.1.

4.5.1 Proof for Join Views without Projection

To show that the V locking protocol keeps the isolation property (serializability) of transactions, we only need to prove that for a join view $JV=\sigma(R_1 \bowtie \ldots \bowtie R_i \bowtie \ldots \bowtie R_n)$, the following assertions hold (the strict two-phase locking protocol guarantees these four assertions for the base relations) [BHG87, GR93]:

(1) Assertion 1: Transaction *T*'s writes to join view *JV* are neither read nor written by other transactions until transaction *T* completes.

(2) Assertion 2: Transaction *T* does not overwrite dirty data of other transactions in join view *JV*.

(3) Assertion 3: Transaction *T* does not read dirty data from other transactions in join view *JV*.

(4) Assertion 4: Other transactions do not write any data in join view *JV* that is read by transaction *T* before transaction *T* completes.

That is, we need to prove that no read-write, write-read, or write-write conflicts exist.

The proof for the absence of read-write or write-read conflicts is trivial, as V, W, and X locks are not compatible with S locks. In the following, we prove the absence of write-write conflicts. Consider the join result tuple $t_1 \bowtie \ldots \bowtie t_i \bowtie \ldots \bowtie t_n$ in the join view *JV* where tuple $t_i \in R_i$ $(1 \le i \le n)$. To update this join result tuple in the join view *JV*, transaction *T* has to update some tuple in some base relation. Suppose transaction *T* updates tuple t_i in base relation R_i for some $1 \le i \le n$. Then transaction *T* needs to use an X lock to protect tuple $t_i \in R_i$. Also, for join view maintenance, transaction *T* needs to use S locks to protect all the other tuples $t_j \in R_j$ $(1 \le j \le n,\ j \ne i)$. Then according to the two-phase locking protocol, before transaction *T* finishes execution, no other transaction can update any tuple $t_k \in R_k$ $(1 \le k \le n)$. That is, no other transaction can update the same join result tuple $t_1 \bowtie \ldots \bowtie t_i \bowtie \ldots \bowtie t_n$ in the join view *JV* until transaction *T* finishes execution. For a similar reason, transaction *T* does not overwrite dirty data of other transactions in the join view *JV*. ■

4.5.2 Proof for Join Views with Projection

Now we prove the correctness (serializability) of the V locking protocol for the general case where $JV = \pi(\sigma(R_1 \bowtie \ldots \bowtie R_i \bowtie \ldots \bowtie R_n))$. We assume that join view *JV* allows duplicate tuples. If no duplicate tuples are allowed in *JV*, we assume that each tuple in *JV* has a *dupcnt* attribute recording the number of copies of that tuple [LYC+00], otherwise *JV* cannot be incrementally maintained efficiently. For example, suppose we do not maintain the *dupcnt* attribute in *JV*. Suppose we delete a tuple from a base relation R_i $(1 \le i \le n)$ of *JV* and this tuple (when joined with other base relations) produces tuple *t* in *JV*. Then we cannot decide whether we should delete tuple *t* from *JV* or not, as there may be other tuples in base relation R_i that (when joined with other base relations) also produces tuple *t* in *JV*. If we maintain the *dupcnt* attribute in the join view *JV*, then *JV* becomes an aggregate join view. The proof for the aggregate join view case is shown in Section 4.5.3 below. Hence, in the following, we only consider join views that allow duplicate tuples.

For a join view *JV* with projection, multiple tuples in *JV* may have the same value due to projection. In this case, the V locking protocol allows multiple transactions to update the same tuple in the join view *JV* concurrently. Hence, the proof in Section 4.5.1 no longer works.

We use an example to illustrate the point. Suppose the schema of base relation *A* is (*a*, *c*), the schema of base relation *B* is (*d*, *e*). The join view *JV* is defined as follows:

create join view *JV* as

select *A.a*, *B.e* from *A*, *B* where *A.c*=*B.d*;

Suppose base relation *A*, base relation *B*, and the join view *JV* originally look as shown in Figure 4.7.

relation *A*

	a	*c*
t_{A1}	1	4
t_{A2}	1	5

relation *B*

	d	*e*
t_{B1}	4	1
t_{B2}	5	2

join view *JV*

	a	*e*
t_{JV1}	1	1
t_{JV2}	1	2

Figure 4.7 Original status of base relation *A*, base relation *B*, and join view *JV*.

Consider the following two transactions. Transaction T_1 updates tuple t_{B1} in base relation *B* from (4, 1) to (4, 2). To maintain the join view *JV*, we compute the old and new join result tuples (1, 4, 4, 1) and (1, 4, 4, 2). Then we update tuple t_{JV1} in the join view *JV* from (1, 1) to (1, 2).

relation *A*

	a	*c*
t_{A1}	1	4
t_{A2}	1	5

relation *B*

	d	*e*
t_{B1}	4	2
t_{B2}	5	2

join view *JV*

	a	*e*
t_{JV1}	1	2
t_{JV2}	1	2

Figure 4.8 Status of base relation *A*, base relation *B*, and join view *JV* – after updating tuple t_{B1}.

Now a second transaction T_2 updates tuple t_{B2} in base relation *B* from (5, 2) to (5, 3). To maintain the join view *JV*, we compute the old and new join result tuples (1, 5, 5, 2) and (1, 5, 5, 3). Then we need to update one tuple in the join view *JV* from (1, 2) to (1, 3). Since all the tuples in the join view *JV* have value (1, 2) at present, it makes no difference which tuple we select to update. Suppose we select tuple t_{JV1} in the join view *JV* for update.

relation *A*

	a	*c*
t_{A1}	1	4
t_{A2}	1	5

relation *B*

	d	*e*
t_{B1}	4	2
t_{B2}	5	3

join view *JV*

	a	*e*
t_{JV1}	1	3
t_{JV2}	1	2

Figure 4.9 Status of base relation *A*, base relation *B*, and join view *JV* – after updating tuple t_{B2}.

Note transactions T_1 and T_2 update the same tuple t_{JV1} in the join view *JV*. At this point, if we abort transaction T_1, we cannot change tuple t_{JV1} in the join view *JV* back to the value (1, 1), as the current value of tuple t_{JV1} is (1, 3) rather than (1, 2). However, we can pick up any other tuple (such as t_{JV2}) in the join view *JV* that has value (1, 2) and changes its value back to (1, 1). That is, our V locking protocol requires logical undo (instead of physical undo) on the join view if the transaction holding the V lock aborts.

In the following, we give an "indirect" proof of the correctness of the V locking protocol using the serializability result in Section 4.5.1. Our intuition is that although multiple tuples in the join view *JV* may have the same value due to projection, they originally come from different join result tuples before projection. Hence, we can show serializability by "going back" to the original join result tuples.

Consider an arbitrary database *DB* containing multiple base relations and join views. Suppose that there is another database *DB′* that is a "copy" of *DB*. The only difference between *DB* and *DB′* is that for each join view with projection $JV=\pi(\sigma(R_1\bowtie\ldots\bowtie R_i\bowtie\ldots\bowtie R_n))$ in *DB*, we replace it by a join view without projection $JV'=\sigma(R_1\bowtie\ldots\bowtie R_i\bowtie\ldots\bowtie R_n)$ in *DB'*. Hence, $JV=\pi(JV')$. Each tuple t in the join view *JV* corresponds to one tuple t' in JV' (by projection).

Consider multiple transactions T_1, T_2, …, and T_g. To prove serializability, we need to show that in *DB*, any allowed concurrent execution of these transactions is equivalent to some serial execution of these transactions. Suppose that multiple transactions T_1', T_2', …, and T_g' exist in *DB'*. Each transaction T_j' $(1\leq j\leq g)$ is a "copy" of transaction T_j with the following differences:

(1) Suppose in *DB*, transaction T_j reads tuples Δ of *JV*. In *DB'*, we let transaction T_j' read the tuples Δ' in JV' that correspond to Δ in *JV*.

(2) Suppose in *DB*, transaction T_j updates *JV* by Δ. According to the join view maintenance algorithm, transaction T_j needs to first compute the corresponding join result tuples Δ' that produce Δ, then integrate Δ' into *JV*. In *DB'*, we let transaction T_j' update JV' by Δ'. That is, we always keep $JV=\pi(JV')$.

Hence, except for the projection on the join views,

(1) For every j $(1\leq j\leq g)$, transactions T_j' and T_j read and write the "same" tuples.

(2) At any time, *DB'* is always a "copy" of *DB*.

For any allowed concurrent execution *CE* of transactions T_1, T_2, …, and T_g in *DB*, we consider the corresponding (and also allowed) concurrent execution *CE'* of transactions T_1', T_2', …, and T_g' in *DB'*. By the reasoning in Section 4.5.1, we know that in *DB'*, such an concurrent execution *CE'* of transactions T_1', T_2', …, and T_g' is equivalent to some serial execution of the same transactions. Suppose one such serial execution is transactions T'_{k_1}, T'_{k_2}, …, and T'_{k_g}, where $\{k_1, k_2, \ldots, k_g\}$ is a permutation of $\{1, 2, \ldots, g\}$. Then it is easy to see that in *DB*, the concurrent execution *CE* of transactions T_1, T_2, …, and T_g is equivalent to the serial execution of transactions T_{k_1}, T_{k_2}, …, and T_{k_g}.

■

4.5.3 Proof for Aggregate Join Views

We can also prove the correctness (serializability) of the V+W locking protocol for aggregate join views. Such a proof is similar to the proof in Section 4.5.2, so we only point out the differences between these two proofs and omit the details:

(1) For any aggregate join view $AJV=\gamma(\pi(\sigma(R_1\bowtie\ldots\bowtie R_i\bowtie\ldots\bowtie R_n)))$ in *DB*, we replace it by a join view $JV'=\sigma(R_1\bowtie\ldots\bowtie R_i\bowtie\ldots\bowtie R_n)$ in *DB'*. Each tuple in the aggregate join view *AJV* corresponds to one or multiple tuples in *JV'* (by projection and aggregation). At any time, we always keep $AJV=\gamma(\pi(JV'))$, utilizing the fact that the addition operation for the *SUM*, *COUNT*, and *AVG* aggregate operators is both commutative and associative.

(2) In the presence of updates that cause the insertion or deletion of tuples in the aggregate join view, the short-term W locks guarantee that the "race" conditions that can cause the split group duplicate problem cannot occur. For each aggregate group, at any time at most one tuple corresponding to this group exists in the aggregate join view *AJV*.

4.6 Performance of the V Locking Protocol

In this section, we investigate the performance of the V locking protocol through a simulation study in a commercial parallel RDBMS. We focus on the throughput of a targeted class of transactions (i.e., transactions that update a base relation of an aggregate join view). Our measurements were performed with the database client application and server running on an NCR WorldMark 4400 workstation with four 400MHz processors, 1GB main memory, six 8GB disks, and running the Microsoft Windows 2000 operating system. We allocated a processor and a disk for each data server, so there were at most four data servers on each workstation.

4.6.1 Benchmark Description

We used the two relations *lineitem* and *partsupp* and the aggregate join view *suppcount* that are mentioned in the introduction for the tests. The schemas of the *lineitem* and *partsupp* relations are listed as follows:

lineitem (<u>orderkey</u>, partkey, price, discount, tax, orderdate, comment)

partsupp (<u>partkey</u>, suppkey, supplycost, comment)

The underscore indicates the partitioning attributes. The aggregate join view *suppcount* is partitioned on the *suppkey* attribute. For each relation, we built an index on the partitioning attribute. In our tests, different *partsupp* tuples have different *partkey* values. There are *R* different *suppkey*s, each corresponding to the same number of tuples in the *partsupp* relation.

	number of tuples	total size
lineitem	8M	586MB
partsupp	0.25M	29MB

Table 4.4 Test data set.

We used the following kind of transaction for testing:

T: Insert *r* tuples that have a specific *orderkey* value into the *lineitem* relation. Each of these *r* tuples has a different and random *partkey* value and matches a *partsupp* tuple on the *partkey* attribute. Each of these *r* matched *partsupp* tuples has a different (and thus random) *suppkey* value.

We evaluated the performance of our V lock method and the traditional X lock method in the following way:

(1) We tested our largest available hardware configuration with four data server nodes. This is to prevent certain system resources (e.g., disk I/Os) from becoming a bottleneck too easily in the presence of high concurrency.

(2) We ran *x* *T*'s. Each of these *x* *T*'s has a different *orderkey* value. *x* is an arbitrarily large number. Its specific value does not matter, as we only focus on the throughput of the RDBMS.

(3) In the X lock method, if a transaction deadlocked and aborted, we automatically re-executed it until it committed.

(4) We used the tuple throughput (number of tuples inserted successfully per second) as the performance metric. It is easy to see that the transaction throughput = the tuple throughput / *r*. In the rest of Section 4.6, we use throughput to refer to the tuple throughput.

(5) We performed two tests:

    (a) **Concurrency test**: We fixed $R=3,000$. In both the V lock method and the X lock method, we tested four cases: $m=2$, $m=4$, $m=8$, and $m=16$, where *m* is the number of concurrent transactions. In each case, we let *r* vary from 1 to 64.

    (b) **Number of aggregate groups test**: We fixed $m=16$ and $r=32$. In both the V lock method and the X lock method, we let *R* vary from 1,500 to 6,000.

    Before we ran each test, we restarted the computer to ensure a cold buffer pool.

(6) We could not implement our V locking protocol in the database software, as we did not have access to the source code. Since the essence of the V locking protocol is that V locks do not conflict with each other, we used the following method to evaluate the performance of the V lock method. We created *m* copies of the aggregate join view *suppcount*. At any time, each of the *m* concurrent transactions dealt with a different copy of *suppcount*. Using this method, our testing

results of the V lock method would show slightly different performance from that of an actual implementation of the V locking protocol. This is because in an actual implementation of the V locking protocol, we would encounter the following issues:

(a) Short-term X page latch conflicts and W lock conflicts during concurrent updates to the aggregate join view *suppcount*.

(b) Hardware cache invalidation in an SMP environment during concurrent updates to the aggregate join view *suppcount*.

However, we believe that these issues are minor compared to the substantial performance improvements gained by the V lock method over the X lock method (see Sections 4.6.2 and 4.6.3 below for details). The general trend shown in our testing results should be close to that of an actual implementation of the V locking protocol.

4.6.2 Concurrency Test Results

We first discuss the deadlock probability and throughput testing results from the concurrency test in Sections 4.6.2.1 and 4.6.2.2, respectively.

4.6.2.1 Deadlock Probability

As mentioned in the introduction, for the X lock method, we can use the unified formula $min(1, (m-1)(r-1)^4/(4R^2))$ to roughly estimate the probability that any particular transaction deadlocks. We show the deadlock probability of the X lock method computed by the unified formula in Figure 4.10. (Note: all figures in Sections 4.6.2.1 and 4.6.2.2 use logarithmic scale for the x-axis.)

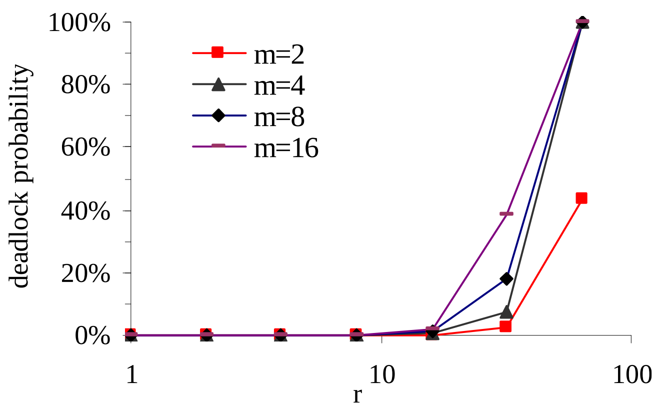


Figure 4.10 Predicted deadlock probability of the X lock method (concurrency test).

For the X lock method, the deadlock probability increases linearly with both *m* and the fourth power of *r*. When both *m* and *r* are small, this deadlock probability is small. However, when either *m* or *r* becomes large, this deadlock probability approaches 1 quickly. For example, consider the case with *m=16*. When *r=16*, this deadlock probability is only 2%. However, when *r=32*, this deadlock probability becomes 38%. The larger *r*, the smaller *m* is needed to make this deadlock probability become close to 1.

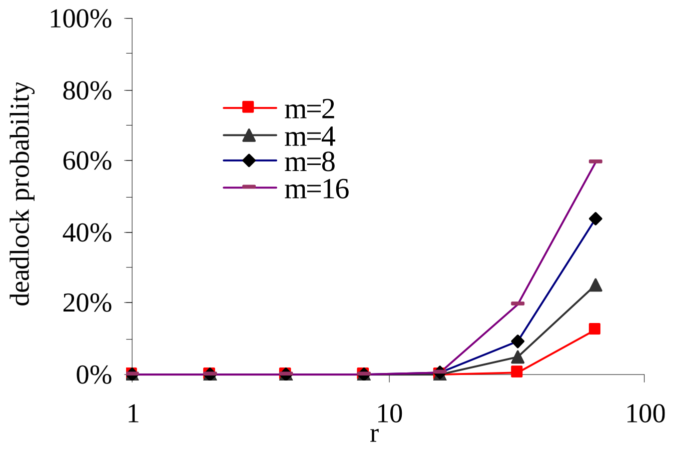


Figure 4.11 Measured deadlock probability of the X lock method (concurrency test).

We show the deadlock probability of the X lock method measured in our tests in Figure 4.11 (see above). Figures 4.10 and 4.11 roughly match. This indicates that our unified formula is fairly good for the purpose of giving a rough estimate of the deadlock probability of the X lock method.

For the X lock method, to see how deadlocks influence performance, we investigated the relationship between the throughput and the deadlock probability. By definition, when the deadlock probability becomes close to 1, almost every transaction will deadlock. Deadlock has the following negative influences on throughout:

(1) Deadlock detection/resolution is a time-consuming process. During this period, the deadlocked transactions cannot make any progress.

(2) The deadlocked transactions will be aborted and re-executed. This wastes system resources.

(3) Once deadlocks start to occur, they tend to occur repeatedly. This is because the deadlocked transactions will be aborted and re-executed. During re-execution, these transactions may deadlock again. That is, these transactions may loop in the circle of deadlock, abortion, and re-execution several times.

(4) Transactions that are deadlocked will not release the locks held by them until they are aborted. During this period, other transactions requesting these locks will be blocked. These other transactions will not release their locks for quite some time and may block other transactions.

Hence, once the system starts to deadlock, the deadlock problem tends to become worse and worse. Eventually, the throughput of the X lock method deteriorates significantly.

4.6.2.2 Throughput

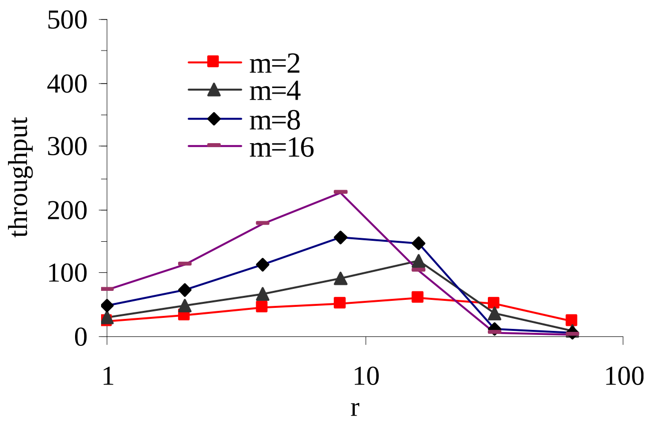


Figure 4.12 Throughput of the X lock method (concurrency test).

We show the throughput of the X lock method in Figure 4.12. For a given *m*, when *r* is small, the throughput of the X lock method keeps increasing with *r*. This is because executing a large transaction is much more efficient than executing a large number of small transactions. When *r* becomes large enough (e.g., *r=32*), the X lock method causes a large number of deadlocks. That is, the X lock method runs into a severe deadlock problem. The larger *m*, the smaller *r* is needed for the X lock method to run into the deadlock problem. Once the deadlock problem occurs, the throughput of the X lock method deteriorates significantly. Actually, it decreases as *r* increases. This is because the larger *r*, the more transactions are aborted and re-executed due to deadlock.

For a given *r*, before the deadlock problem occurs, the throughput of the X lock method increases with *m*. This is because the larger *m*, the higher concurrency in the RDBMS. However, when *r* is large enough (e.g., *r=32*) and the X lock method runs into the deadlock problem, due to the extreme overhead of repeated transaction abortion and re-execution, the throughput of the X lock method decreases as *m* increases.

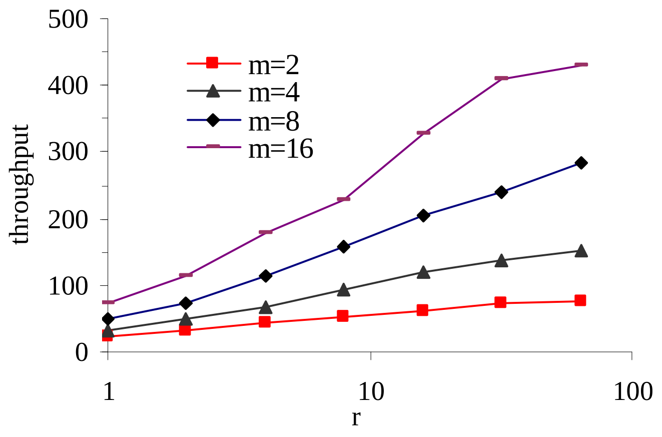


Figure 4.13 Throughput of the V lock method (concurrency test).

We show the throughput of the V lock method in Figure 4.13. The general trend of the throughput of the V lock method is similar to that of the X lock method (before the deadlock problem occurs). That is, the throughput of the V lock method increases with both *m* and *r*. However, the V lock method never deadlocks. For a given *m*, the throughput of the V lock method keeps increasing with *r* (until all system resources become fully utilized). Once the X lock method runs into the deadlock problem, the V lock method exhibits great performance advantages over the X lock method, as the throughput of the X lock method in this case deteriorates significantly.

We show the ratio of the throughput of the V lock method to that of the X lock method in Figure 4.14. (Note: Figure 4.14 uses logarithmic scale for both the x-axis and the y-axis.) Before the X lock method runs into the deadlock problem, the throughput of the V lock method is the same as that of the X lock method. However, when the X lock method runs into the deadlock problem, the throughput of the V lock method does not drop while the throughput of the X lock method is significantly worse. In this case, the ratio of the throughput of the V lock method to that of the X lock method is greater than 1. For example, when *r=32*, for any *m*, this ratio is at least 1.3. When *r=64*, for any *m*, this ratio is at least 3. In general, when the X lock method runs into the deadlock problem, this ratio increases with both *m* and *r*. This is because the larger *m* or *r*, the easier the transactions deadlock in the X lock method. The extreme overhead of repeated transaction abortion and re-execution exceeds the benefit of the higher concurrency (efficiency) brought by a larger *m* (*r*).

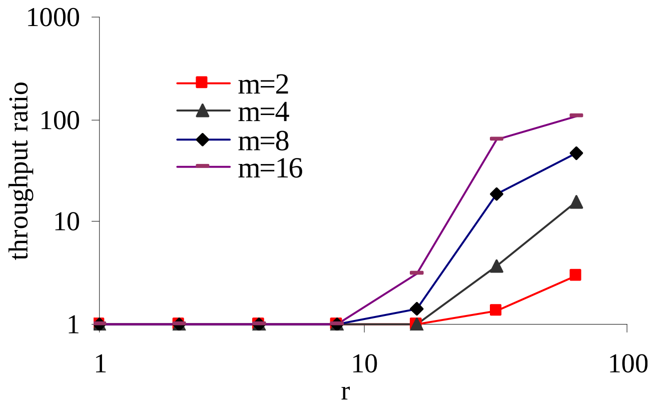


Figure 4.14 Throughput improvement gained by the V lock method (concurrency test).

4.6.3 Number of Aggregate Groups Test Results

In this section, we discuss the deadlock probability and throughput testing results from the number of aggregate groups test. Recall that in the number of aggregate groups test, we fixed *m=16*, *r=32*, and let *R* vary from 1,500 to 6,000. We show the deadlock probability of the X lock method computed by the unified formula and measured in our tests in Figure 4.15 (see below). The two curves in Figure 4.15 roughly match. This indicates that our unified formula roughly reflects the real world situation. For the X lock method, the deadlock probability increases quadratically as *R* decreases. That is, the smaller the number of distinct *suppkey*s *R*, the larger the deadlock probability.

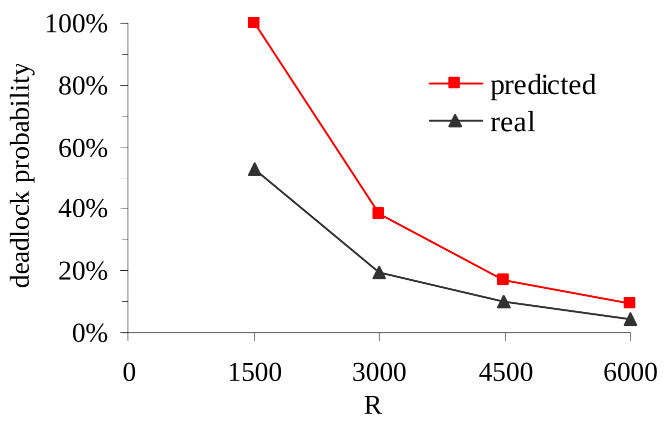


Figure 4.15 Deadlock probability of the X lock method (number of aggregate groups test).

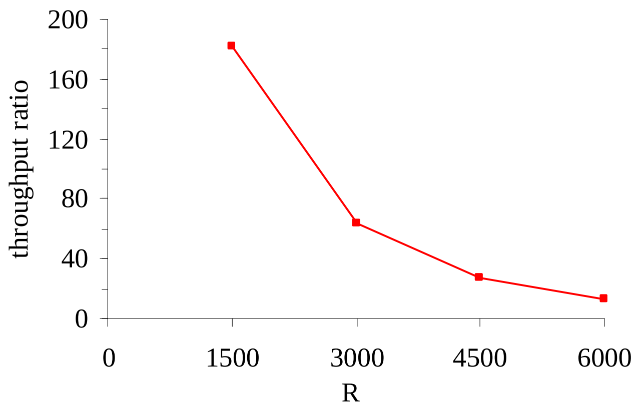


Figure 4.16 Throughput improvement gained by the V lock method (number of aggregate groups test).

We show the ratio of the throughput of the V lock method to that of the X lock method in Figure 4.16. In all our testing cases, the X lock method runs into the deadlock problem and the ratio is greater than 1. The smaller the number of distinct *suppkey*s *R*, the more severe the deadlock problem of the X lock method and the greater the ratio. That is, the smaller *R*, the greater performance advantages the V lock method exhibits over the X lock method.

We believe that in a real world workload, it would be common for our V locking protocol to exhibit significant performance advantages over the traditional X locking protocol. This is because in a real world workload, people may use different aggregate join views that have different *R* values. These *R* values may be either larger or smaller than the *R* values used in our testing. However, in practice, the number *m* of concurrent transactions would be much larger than the ones used in our testing. Hence, even if the *R* values are larger than the ones used in our testing, it would still be common for the X lock method to have a high deadlock probability.

4.7 Conclusion

The V locking protocol is designed to support concurrent, immediate updates of materialized aggregate join views without engendering the high lock conflict rates and high deadlock rates that could result if two-phase locking with S and X lock modes were used. This protocol borrows from the theory of concurrency control for associative and commutative updates, with the addition of a short-term W lock to deal with insertion anomalies that result from some special properties of materialized view updates. Perhaps surprisingly, due to the interaction between locks on base relations and locks on the materialized view, this locking protocol, designed for concurrent update of aggregates, also supports direct propagate updates and materialized non-aggregate join view maintenance.

It is an open question whether or not immediate updates with serializable semantics are a good idea in the context of materialized views. Certainly there are advantages to deferred updates, including potential efficiencies from the batching of updates and shorter path lengths for transactions that update base relations mentioned in materialized views. However, these efficiencies must be balanced against the semantic uncertainty and the “stale data” problems that may result when materialized views are not “in synch” with base data. The best answer to this question will only be found through a thorough exploration of how well both approaches (deferred and immediate) can be supported; it is our hope that the techniques in this chapter can contribute to the discussion in this regard.

Chapter 5: Conclusion

In a broad sense, this dissertation focused on understanding some fundamental aspects of building an operational data warehouse that can be used to support real-time decision-making about an enterprise's day-to-day operations. Among the large number of challenges in achieving this goal, we focused on two key issues: (1) providing better access to early query results while queries are running and (2) making the information stored in a data warehouse as fresh as possible.

For the first problem, we introduced a non-blocking parallel hash ripple join algorithm to support interactive queries in a parallel DBMS. Compared to previous work, our parallel hash ripple join algorithm (a) combines parallelism with sampling to speed convergence, and (b) maintains good performance in the presence of memory overflow. Results from a prototype implementation in a parallel DBMS showed that its rate of convergence scales with the number of processors, and that when allowed to run to completion, even in the presence of memory overflow, it is competitive with the traditional parallel hybrid hash join algorithm.

For the second problem, we proposed two techniques to improve the efficiency of immediate materialized view maintenance. We identified two challenges for immediate materialized view maintenance: resource usage and lock conflicts.

With respect to the first challenge, we showed that in parallel RDBMSs, simple single-node updates to base relations can give rise to expensive all-node operations for materialized view maintenance. We presented a comparison of three materialized join view maintenance methods in a parallel RDBMS, which we referred 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 showed that the method of choice depends on the environment, in particular, the update activity on base relations and the amount of available storage space.

With respect to the second challenge, we showed that immediate materialized view maintenance with transactional consistency, if enforced by generic concurrency control mechanisms, can result in low levels of concurrency and high rates of deadlock. While this problem is superficially amenable to well-known techniques such as fine-granularity locking and special lock modes for updates that are associative and commutative, we showed that these previous techniques do not fully solve the problem. We extended previous high concurrency locking techniques to apply to materialized view maintenance, and showed how this extension can be implemented even in the presence of indices on the materialized view.

There is substantial opportunity for future work on operational data warehousing. For example:

(1) To improve the visibility of the operational status of a DBMS, we can add progress indicators for SQL queries. Supporting progress indicators for the entire breadth of the SQL standard is a significant undertaking. For example, some of the open problems are: (a) how to continuously refine the estimated costs of UDFs and spatial queries, (b) how to provide precise estimates for nested queries with correlated sub-queries at a reasonable overhead, and (c) how to support progress indicators for SQL queries in a parallel DBMS.

(2) It would be interesting to build a workload management tool that makes use of progress indicators to provide feedback on the state of the DBMS. For example, suppose that for some reason, the DBA needs to speed up the execution of a certain query. The DBA might choose to block the execution of several queries to allow more resources to be allocated to the certain query. The workload management tool can help the DBA choose which queries to block based on the state of the queries including information such as memory usage, remaining query execution time, etc.

(3) It would be interesting to use progress indicators to facilitate automatic database administration. For example, the user may embed triggers in a progress indicator. The firing condition of such a trigger can be: “send an email to the user if after a whole day’s execution, the query finishes less

than 10% of the work.” This trigger function can be achieved by keeping track of the history of the progress indicator.

We intend to pursue these issues in future work.

Bibliography

[BDT83] D. Bitton, D.J. DeWitt, and C. Turbyfill. Benchmarking Database Systems: A Systematic Approach. VLDB 1983: 8-19.

[BHG87] P.A. Bernstein, V. Hadzilacos, and N. Goodman. Concurrency Control and Recovery in Database Systems. Addison-Wesley publishers, 1987.

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

[BR92] B.R. Badrinath, K. Ramamritham. Semantics-Based Concurrency Control: Beyond Commutativity. TODS 17(1): 163-199, 1992.

[CM96] D. Choy, C. Mohan. Locking Protocols for Two-Tier Indexing of Partitioned Data. Proc. International Workshop on Advanced Transaction Models and Architectures, 1996.

[Coc77] W.G. Cochran. Sampling Techniques. John Wiley and Sons, Inc., New York, 3rd edition, 1977.

[GK85] D. Gawlick, D. Kinkade. Varieties of Concurrency Control in IMS/VS Fast Path. Database Engineering Bulletin 8(2): 3-10, 1985.

[GKS01] J. Gehrke, F. Korn, and D. Srivastava. On Computing Correlated Aggregates over Continual Data Streams. SIGMOD Conf. 2001: 13-24.

[GLP$^+$76] J. Gray, R.A. Lorie, and G.R. Putzolu et al. Granularity of Locks and Degrees of Consistency in a Shared Data Base. IFIP Working Conference on Modeling in Data Base Management Systems 1976: 365-394.

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

[GR93] J. Gray, A. Reuter. Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers, 1993.

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

[Haa97] P.J. Haas. Large-Sample and Deterministic Confidence Intervals for Online Aggregation. Proc. Ninth Intl. Conf. Scientific and Statistical Database Management, 1997, 51-62.

[Haa00] P.J. Haas. Hoeffding inequalities for online aggregation. Proc. Computing Sci. Statist.: 31st Symp. on the Interface, 74-78. Interface Foundation of North America, 2000.

[HFC$^+$00] J.M. Hellerstein, M. Franklin, and S. Chandrasekaran et al. Adaptive Query Processing: Technology in Evolution. IEEE Data Engineering Bulletin, June 2000.

[HH98] P.J. Haas and J.M. Hellerstein. Join algorithms for online aggregation. IBM Research Report RJ 10126, IBM Almaden Research Center, San Jose, CA, 1998.

[HH99] P.J. Haas, J.M. Hellerstein. Ripple Joins for Online Aggregation. SIGMOD Conf. 1999: 287-298.

[HHW97] J.M. Hellerstein, P.J. Haas, and H. Wang. Online Aggregation. SIGMOD Conf. 1997: 171-182.

[HNS$^+$96] P.J. Haas, J.F. Naughton, and S.Seshadri et al. Selectivity and cost estimation for joins based on random sampling. J. Comput. System Sci., 52:550-569, 1996.

[IFF$^+$99] Z.G. Ives, D. Florescu, and M. Friedman et al. An Adaptive Query Execution System for Data Integration. SIGMOD Conf. 1999: 299-310.

[KLM+97] A. Kawaguchi, D.F. Lieuwen, and I.S. Mumick et al. Concurrency Control Theory for Deferred Materialized Views. ICDT 1997: 306-320.

[KMH97] M. Kornacker, C. Mohan, and J.M. Hellerstein. Concurrency and Recovery in Generalized Search Trees. SIGMOD Conf. 1997: 62-72.

[Knu98] D.E. Knuth. The Art of Computer Programming, Vol 2. Addison Wesley, 3rd edition, 1998.

[Kor83] H.F. Korth. Locking Primitives in a Database System. JACM 30(1): 55-79, 1983.

[LYC+00] W. Labio, J. Yang, and Y. Cui et al. Performance Issues in Incremental Warehouse Maintenance. VLDB 2000: 461-472.

[Lom93] D.B. Lomet. Key Range Locking Strategies for Improved Concurrency. VLDB 1993: 655-664.

[Moh90a] C. Mohan. ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes. VLDB 1990: 392-405.

[Moh90b] C. Mohan. Commit_LSN: A Novel and Simple Method for Reducing Locking and Latching in Transaction Processing Systems. VLDB 1990: 406-418.

[O86] P.E. O'Neil. The Escrow Transactional Method. TODS 11(4): 405-430, 1986.

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

[PF00] M. Poess, C. Floyd. New TPC Benchmarks for Decision Support and Web Commerce. SIGMOD Record 29(4): 64-71, 2000.

[QGM+96] D. Quass, A. Gupta, and I.S. Mumick et al. 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.

[RAA94] R.F. Resende, D. Agrawal, and A.E. Abbadi. Semantic Locking in Object-Oriented Database Systems. OOPSLA 1994: 388-402.

[Reu82] A. Reuter. Concurrency on High-trafic Data Elements. PODS 1982: 83-92.

[SD89] D.A. Schneider, D.J. DeWitt. A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment. SIGMOD Conf. 1989: 110-121.

[SK80] A. Silberschatz, Z.M. Kedem. Consistency in Hierarchical Database Systems. JACM 27(1): 72-80, 1980.

[SN95] A. Shatdal, J.F. Naughton. Adaptive Parallel Aggregation Algorithms. SIGMOD Conf. 1995: 104-114.

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

[UF99] T. Urhan, M. Franklin. XJoin: Getting Fast Answers from Slow and Bursty Networks. Technical Report. CS-TR-3994, UMIACS-TR-99-13. February, 1999.

[WA91] A.N. Wilschut, P.M. Apers. Dataflow Query Execution in a Parallel Main-Memory Environment. PDIS 1991: 68-77.

[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.