A Scalable Hash Ripple Join Algorithm

Gang Luo[1] Curt J. Ellmann[2] Peter J. Haas[3] Jeffrey F. Naughton[1]
University of Wisconsin-Madison[1] NCR Advance Development Lab[2] IBM Almaden Research Center[3]
gangluo@cs.wisc.edu curt.ellmann@ncr.com peterh@almaden.ibm.com naughton@cs.wisc.edu

ABSTRACT

Recently, Haas and Hellerstein proposed the hash ripple join algorithm in the context of online aggregation. Although the algorithm 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. Furthermore, if memory overflows (for example, because the user allows the algorithm to run to completion for an exact answer), the algorithm degenerates to block ripple join and performance suffers. In this paper, we build on the work of Haas and Hellerstein and propose a new algorithm that (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 show 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.

1. Introduction

Online aggregation was proposed by Hellerstein et al. [9] 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 1. The precision of the running estimate improves as more and more input tuples are processed. In [8], 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.

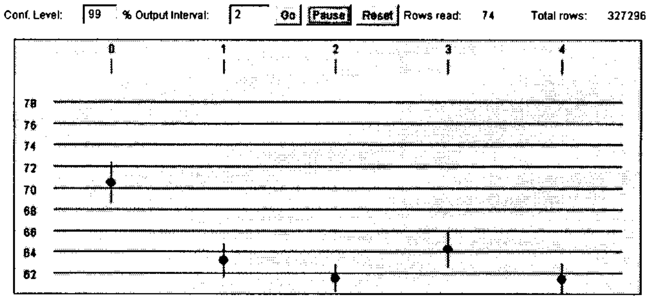

Conf. Level: 99 % Output Interval: 2
Rows read: 74 Total rows: 327296

Figure 1. An online aggregation interface for a query of the form: select avg(temperature) from t group by site.

In this paper 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 [14].

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 very straightforward way to a multiprocessor hash join. Our general approach is to use stratified sampling techniques that are similar in spirit to [2]. 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

ACM SIGMOD 2002, June 4-6, Madison, Wisconsin, USA.

"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 [2], 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 paper, 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 [8] 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 [8] will degenerate to the much slower block ripple join. One desirable property of the algorithm in [8] is that it continues to process tuples from the input relations in a random and independent way throughout. That is, upon memory overflow, [8] maintains independence and randomness at the expense of performance.

In this paper 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. Related Work

Figure 2 illustrates the original hash ripple join algorithm [8].

The original two-table hash ripple join 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 [8] 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.

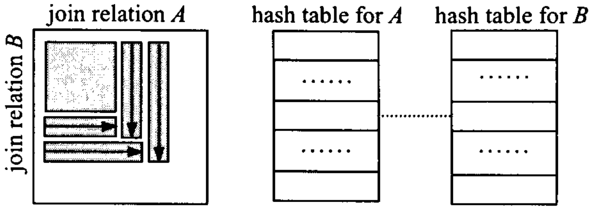


Figure 2. The original hash ripple join algorithm.

Ripple join algorithms for online aggregation are similar in some ways to the XJoin algorithm [17], which dynamically adjusts the algorithm's behavior in accordance with changes in the run time environment. XJoin was proposed for adaptive query processing [6], 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, [11] 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.

3. Parallel Hash Ripple Join Algorithm

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 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 [14] 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.

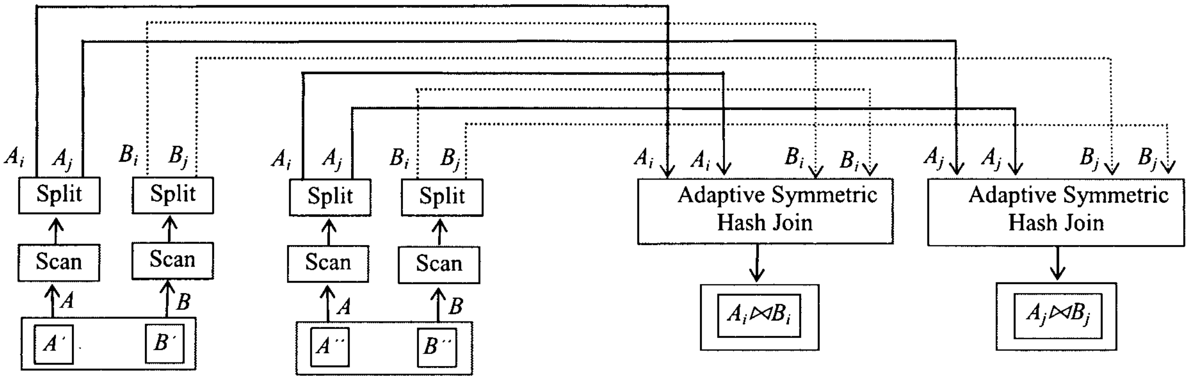


Figure 3. Dataflow network of operators for the parallel hash ripple join algorithm.

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 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 paper simplify when the tuples of A and/or B are already at the appropriate join nodes and need not be redistributed. We omit the details for brevity.)

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 [9], 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" [9].

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 [3], 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.

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 4 shows the hash tables at a specific node:

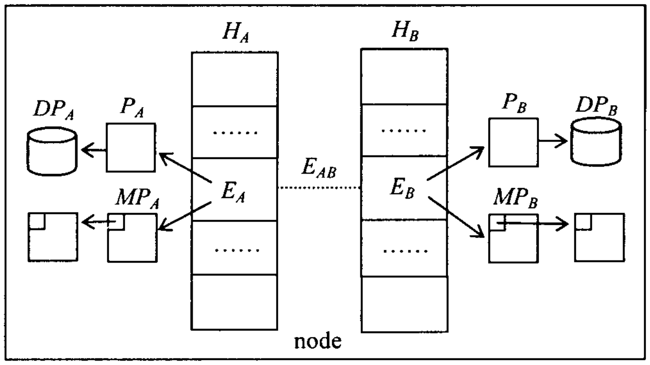


Figure 4. Hash tables at a node.

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

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.

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 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}.

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 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 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.

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 [12] to rearrange the hash table bucket pairs prior to selection.

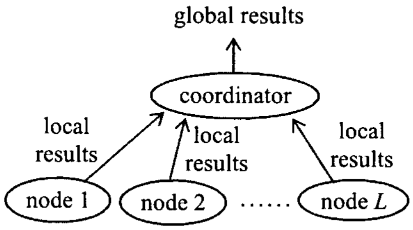
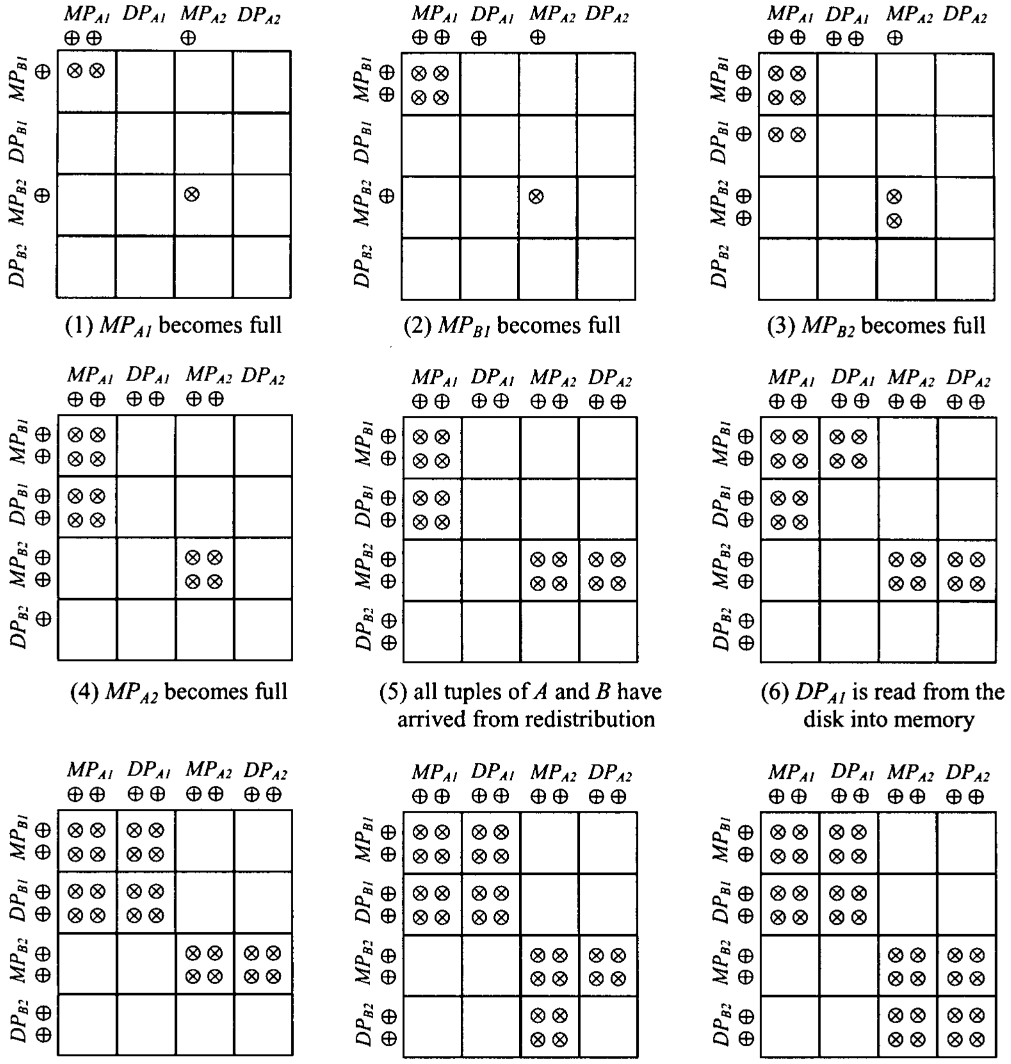


Figure 6. Computing global results from local results.



(1) MP_{A1} becomes full

(2) MP_{B1} becomes full

(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 7. Work done in different phases of the running example.

3.3 Supporting Online Aggregation

Ripple joins [8] were proposed to support online aggregation [9]. To support online aggregation in a parallel environment, we use the strategy of the Centralized Two Phase aggregation algorithm [15], as shown in Figure 6. 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*.

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 ⊕ denote the tuples arrived, and ⊗ denote the join work done. Then the progress of the join may look as shown in Figure 7.

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||)≤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.

3.6 Analytical Performance Model

To gain insight into the algorithms' behavior upon memory overflow, we use a simple analytical model. Table 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.

Table 1. System parameters used in the analytical model.

‖*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

We differentiate between the time to run to completion for the original hash ripple join algorithm [8] and our parallel hash ripple join algorithm in three cases:

(1) If ||*A*||+||*B*||≤*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||)≤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:

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

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:

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

(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:

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

Setting the system parameters as shown in Table 2, we present in Figure 8 the resulting performance of the parallel hash ripple join algorithm (PHRJ) and the original hash ripple join algorithm (OHRJ).

Table 2. System parameter settings.

‖*A*‖	2000
‖*B*‖	2000
S	40
join	400 microseconds
IO	30 milliseconds
BLOCK	60
page size	8000 bytes

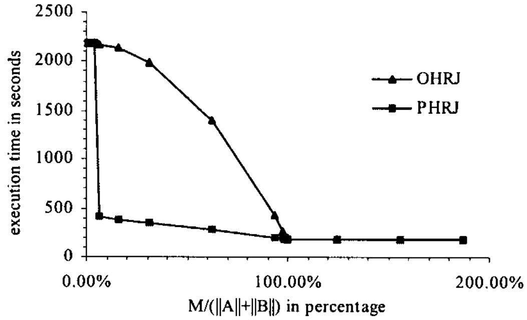


Figure 8. Execution time of join algorithms.

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 [8]. Specifically, the algorithm permits running estimates and associated confidence intervals ("error bars") to be displayed to the user; see, for example, Figure 1. In this section we give an overview of 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 3.3. Classical stratified sampling techniques [2] 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 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 [7, 8] 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.

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 [4, 5, 7] 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.

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 $1 \leq j \leq L_i$ and $1 \leq k \leq M_i$, 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.

4.2.1 Running Estimator

For fixed j, k, observe that, as in [8],

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

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 $1\leq j\leq L_i$ and $1\leq k\leq M_i$, 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

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

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.

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 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 $1\leq j\leq L_i$ and $1\leq k\leq M_i$. Using arguments as in [10], the results in [7, 8] 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 [8, 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 $\sigma^2=\mathrm{Var}[\hat{\mu}]=\sum_i w_i^2\,\mathrm{Var}[\hat{\mu}_i]$. Now fix i and observe that $\mathrm{Var}[\hat{\mu}]=\sum_{j,k,j',k'} w_{ijk}w_{ij'k'}\mathrm{Cov}[\hat{\mu}_{ijk},\hat{\mu}_{ij'k'}]$. Each term $\mathrm{Cov}[\hat{\mu}_{ijk},\hat{\mu}_{ij'k'}]$ 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

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

Note that in the case $j=j'$ and $k=k'$, the formula for $\mathrm{Cov}[\hat{\mu}_{ijk},\hat{\mu}_{ij'k'}]$ is essentially the same as that given in [8]. 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 [10] show that $\hat{\sigma}^2$ is indeed strongly consistent for σ^2. A development along the lines of [7] leads to efficient and stable numerical procedures for computing the various estimates described above.

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 [8], 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 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 [8], 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 $\sigma_{sc} = \mathrm{Cov}[\hat{\mu}_s, \hat{\mu}_c]$. Both σ_s^2 and σ_c^2 can be estimated as described in Section 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 $P(C \text{ and } D) \ge 1 - P(\bar{C}) - P(\bar{D})$ 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 [5] 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

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

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.

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 Intel x86 Family 6 Model 5 Stepping 3 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.

The relations used for the benchmarks were based on the standard Wisconsin Benchmark relations [1]. 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.

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 [14].

select online *avg (A.unique1)*
from *A*, *B*
where *A.unique2 = B.unique2*;

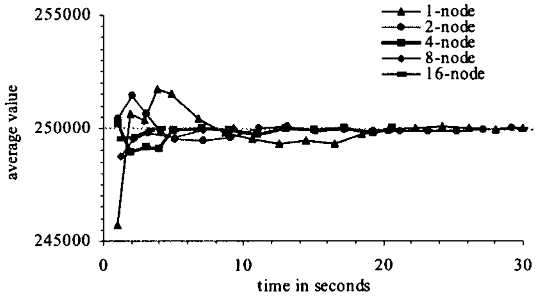


Figure 9. Average value computed over time (high join selectivity).

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 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 10 shows how the number of join result tuples generated increases over time.

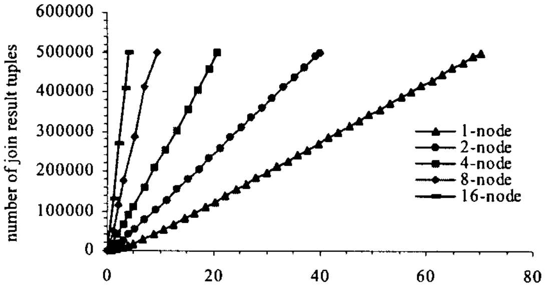


Figure 10. Join result tuples generated over time (high join selectivity).

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.

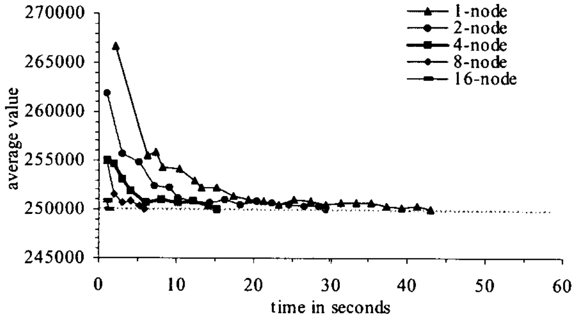


Figure 11. Average value computed over time (low join selectivity).

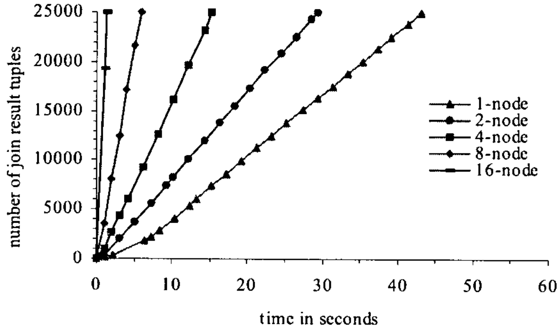


Figure 12. Join result tuples generated over time (low join selectivity).

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 11 and Figure 12, respectively. Note that now, in contrast to the results in Figure 9, the multiple-node configurations converge noticeably quicker than the single node configurations.

Table 3 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.

Table 3. Time (seconds) that memory overflow occurs.

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

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 4 shows the execution to completion time of the PHRJ algorithm and the traditional PHHJ algorithm [14] in the five configurations.

Table 4. Execution to completion time (seconds) of the two parallel join algorithms (speedup).

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

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.

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 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. The scaleup characteristics of the PHRJ algorithm match those of the traditional PHHJ algorithm.

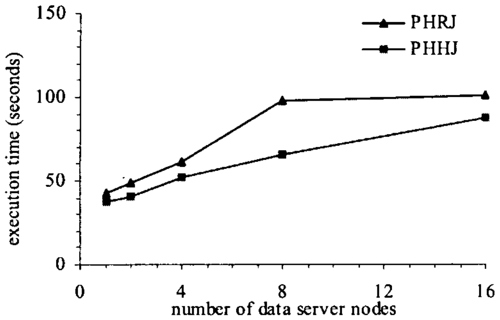


Figure 13. Execution to completion time of the two parallel join algorithms (scale-up).

6. Summary and Conclusions

The two primary tools that permit online exploration of massive data sets are sampling and parallel processing. In this paper 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 [8] 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 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 [8] 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.

Acknowledgements

We would like to thank Josef Burger, Joe Hellerstein, Kevin O'Connor, and Vijayshankar Raman for useful discussions. This work was supported by the NCR Corporation and also by NSF grants CDA-9623632 and ITR 0086002.

References

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

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

[3] G. Graefe. Query Evaluation Techniques for Large Databases. ACM Comput. Surveys, 25(2):73-170, June 1993.

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

[5] 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.

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

[7] 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.

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

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

[10] 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.

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

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

[13] F. Olken. Random Sampling from Databases. Ph.D. dissertation, UC Berkeley, April 1993. Available as Tech. Report LBL-32883, Lawrence Berkeley Laboratories.

[14] 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.

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

[16] K.L. Tan, C.H. Goh, and B.C. Ooi. Online Feedback for Nested Aggregate Queries with Multi-Threading. VLDB 1999: 18-29.

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