Adaptive Parallel Aggregation Algorithms*

Ambuj Shatdal Jeffrey F. Naughton
{shatdal,naughton}@cs.wisc.edu
Computer Sciences Department
University of Wisconsin-Madison

Abstract

Aggregation and duplicate removal are common in SQL queries. However, in the parallel query processing literature, aggregate processing has received surprisingly little attention; furthermore, for each of the traditional parallel aggregation algorithms, there is a range of grouping selectivities where the algorithm performs poorly. In this work, we propose new algorithms that dynamically adapt, at query evaluation time, in response to observed grouping selectivities. Performance analysis via analytical modeling and an implementation on a workstation-cluster shows that the proposed algorithms are able to perform well for all grouping selectivities. Finally, we study the effect of data skew and show that for certain data sets the proposed algorithms can even outperform the best of traditional approaches.

1 Introduction

SQL queries are replete with aggregate and duplicate elimination operations. One measure of the perceived importance of aggregation is that in the proposed TPC-D benchmark [TPC94] 15 out of 17 queries contain aggregate operations. Yet we find that aggregate processing is an issue almost totally ignored by the researchers in the parallel database community. Although it looks straightforward, most parallel database systems (hereafter called PDBMSs) implement aggregate processing algorithms that do not work well for the entire range of grouping selectivities[1]. In this paper we propose three different algorithms for aggregate processing on shared nothing PDBMSs that attempt to overcome this limitation.

The traditional algorithms of parallel aggregation and duplicate removal[2] are as follows. The standard parallel algorithm for aggregation, henceforth called Centralized Two Phase, is for each node in the multiprocessor to first do aggregation on its local partition of the relation. Then these partial results are sent to a centralized coordinator node, which merges these partial results to produce the final result. The second approach, henceforth called the Two Phase, is to parallelize the second phase of the Centralized Two Phase algorithm. The last approach, henceforth called Repartitioning, is to first redistribute the relation on the GROUP BY attributes and then do aggregation on each of the nodes producing the final result in parallel. We show later that the algorithms only well work for certain grouping selectivities. Both Two Phase algorithms work well only when the number of result tuples is small, while the Repartitioning algorithm works well only when the number of groups is large. We propose hybrid algorithms that overcome these shortcomings and work well independent of the number of groups being computed. All the algorithms change (decide) the method of computation depending on the workload, the number of groups in the relation, and are, therefore, able to adapt to different work loads. An interesting aspect of the non-sampling based adaptive algorithms is that each processor involved in the computation adapts based on what it observes, independently of what all the other processors are doing. This aspect of the algorithms allows them to proceed adaptively without any global synchronization.

As mentioned, there has been little work reported in literature on aggregate processing. Epstein [Eps79] discusses some algorithms for computing scalar aggregates and aggregate functions on a uniprocessor. Bitton et al. [BBDW83] discuss two sorting based algorithms for aggregate processing on a shared disk cache architecture. The first algorithm is somewhat similar to the Two Phase approach mentioned above in that it uses local aggregation. The second algorithm of Bitton et

*This research was supported by the NSF grant IRI-9157357

[1] GROUP BY or grouping selectivity is the ratio of the result and the input relation cardinalities.

[2] Duplicate elimination is algorithmically similar to aggregation except that the number of "groups" is relatively much larger in most cases.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association of Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
SIGMOD ' 95,San Jose , CA USA
© 1995 ACM 0-89791-731-6/95/0005..$3.50

al. uses broadcast of the tuples and lets each node process the tuples belonging to a subset of groups. This is impractical on today's multiprocessor interconnects, which do not efficiently support broadcasting. Su et al. [SM82] discuss an implementation of the traditional approach. Graefe [Gra93] discusses one optimization for dealing with bucket overflow for the Two Phase algorithm.

The rest of the paper is organized as follows. Section 2 introduces aggregation, the traditional approaches to aggregate processing and summarizes their performance characteristics. Section 3 presents the three new algorithms for parallel aggregation and duplicate elimination. An analytical evaluation of the different algorithms is presented in Section 4. In Section 5 we describe an implementation of the algorithms on a workstation-cluster and show their performance results. In Section 6 we discuss the effect of data skew on aggregate processing algorithms. Section 7 offers our conclusions.

2 The Background

An SQL aggregate function is a function that operates on groups of tuples. Its basic form is:

select [group by attributes] aggregates **from** {relations}
[**where** {predicates}]
group by {attributes}
having {predicates}

We note that in practice the aggregate operation is often accompanied by the `GROUP BY` operation[3]. Thus the number of result tuples depends on the selectivity of the `GROUP BY` attributes. We find that this selectivity does indeed vary quite a lot resulting in widely varying result sizes. For example, in the TPC-D benchmark we found the result size for aggregation varying from 2 tuples to as large as 0.28 million and 1.4 million tuples for a 100 GB database. In particular, in duplicate elimination the result sizes can be comparable to the input sizes. Hence a truly effective algorithm must cover the entire range of grouping selectivity: i.e. scalar aggregation to the result sizes which could be almost identical to the input sizes.

A properly constructed `HAVING` clause, i.e. one that can't be converted to a `WHERE` clause, is evaluated after the processing of the `GROUP BY` clause and it does not directly affect the performance of the aggregation algorithms we are trying to study. Hence we will assume that the query does not have a `HAVING` clause. In the remainder of the paper we will assume that aggregation is always accompanied with `GROUP BY` and that scalar aggregation can be considered as a special case where number of groups is 1.

[3]In the TPC-D benchmark 13 out of 15 queries with aggregates have `GROUP BY`.

Further, we assume a Gamma [DGS+90] like architecture where each relational operation is represented by operators. The data "flows" through the operators in a pipelined fashion as far as possible. For example, a join of two base relations is implemented as two select operators followed by a join operator. Aggregation can be implemented by one or two operators, as needed, which are fed by some child operator, e.g. a select or a join, and the result is sent to some parent operator, e.g. a store. In our study we assume that the child operator is a scan/select and the parent operator is a store. However, with the exception of the sampling based algorithm, all of our algorithms extend naturally to the case where the child and parent are other operators.

We present analytical cost models of the basic approaches and the proposed algorithms. The cost models developed below are quite simple and as such they should not be interpreted to predict exact running times of the algorithms. The intention is that although the models will not be able to predict the actual running times, they will be good enough to predict the relative performance of the algorithms under varying circumstances. The simplifying assumptions in the model include no overlap between CPU, I/O and message passing, and that all nodes work completely in parallel thus allowing us to study the performance of just one node. Even this simple model generates results that are qualitatively in agreement with measurements from our implementation.

We assume that the aggregation is being performed directly on a base relation stored on disks as in the example query. The parameters of the study are listed in Table 1 unless otherwise specified. These parameters are similar to those in previous studies e.g. [BCL93]. The CPU speed is chosen to reflect the characteristics of the current generation of commercially available microprocessors. The I/O rate was as observed on the SUN disk on the SUN SparcServer20/51. We model both high speed, high bandwidth network as in commercial multiprocessors like IBM SP-2 and slow speed, limited bandwidth network like the Ethernet, to match our implementation platform. The high bandwidth network is modeled only by the latency to send a message i.e. it has unlimited bandwidth. The limited bandwidth is modeled as a sequential resource where sending a fixed amount of data will take a fixed amount of time independent of the number of processors involved. The software parameters are based on instruction counts taken from the Gamma PDBMS. The projectivity, p, is the fraction of tuple relevant to aggregate computation. The grouping selectivity, S, is the ratio of result size to the input size and it varies from the result size of one (scalar aggregate) to the result size as large as half the input relation, as possible in duplicate elimination. In the following we assume that

Sym.	Description	Values
N	number of processors	32
mips	MIPS of the processor	40
R	size of relation	800 MB
$\|R\|$	number of tuples in R	8 Million
$\|R_i\|$	number of R tuples on node i	$\|R\|/N$
P	page size	4 KB
IO	time to read a page (seq. IO)	1.15ms
rIO	time to read a random page	15.0ms
p	projectivity of aggregation	16%
t_r	time to read a tuple	300/mips
t_w	time to write a tuple	100/mips
t_h	time to compute hash value	400/mips
t_a	time to process a tuple	300/mips
S	GROUP BY selectivity	$\frac{1}{\|R\|}$ to 0.5
S_l	phase 1 selectivity in 2-Phase	max(S*N,1)
S_g	phase 2 selectivity in 2-Phase	max($\frac{1}{N}$,S)
t_d	time to compute destination	10/mips
m_p	message protocol cost/page	1000/mips
m_l	time to send a page	2.0 ms
M	default max. hash table size	10K entries

Table 1: Parameters for the Analytical Models

aggregation on a node is done by hashing.

The underlying uniprocessor hash based aggregation works roughly as follows.

1. The tuples of the relation are read and a hash table is built by hashing on the GROUP BY attributes of the tuple. The first tuple hashing to a new value adds an entry to the hash table and the subsequent matches update the cumulative result as appropriate. It is easy to see that the memory requirement for the hash table is proportional to the number of distinct group values seen.

2. If the entire hash table is unable to fit in the allocated memory, the tuples are hash partitioned into multiple (as many as necessary to ensure no future memory overflow) buckets, and all but the first bucket are spooled to disk.

3. The overflow buckets are processed one by one as in step 1 above.

In the following sections we briefly describe the three traditional approaches to parallel aggregation. The first being the Centralized Two Phase algorithm (C-2P) [DGS+90].

2.1 Centralized Two Phase Algorithm

The most simple and common approach for parallel aggregation is for each node to do aggregation on the locally generated tuples in phase one and then merge these local aggregate values at a central coordinator in phase two. Intuitively, this will work well when the sequential merging step is small i.e. when the number of groups is small.

The analytical cost model of the algorithm is as follows. In the first phase each node processes the tuples residing locally.

- scan cost (IO): $(R_i/P) * IO$
- select cost, getting tuple off data page: $|R_i|*(t_r+t_w)$
- local aggregation involving reading, hashing and computing the cumulative value: $|R_i| * (t_r + t_h + t_a)$
- the tuples not processed in first pass need an extra read/write: $(1 - M/S_l) * p * R_i/P * 2 * IO$
- generating result tuples: $|R_i| * S_l * t_w$
- message cost for sending result to coordinator: $(p * R_i * S_l/P) * (m_p + m_l)$

In the second phase these local values are merged by the coordinator. The number of tuples arriving at the coordinator are: $|G| = \sum |R_i| * S_l = |R| * S_l$ and $G = p * R * S_l$.

- receiving tuples from local aggregation operators: $(G/P) * m_p$
- computing the final aggregate value for each group involves reading and computing the cumulative values: $|G| * (t_r + t_a)$
- the tuples not processed in first pass need an extra read/write: $(1 - M/S_g) * G/P * 2 * IO$
- generating final result: $|G| * S_g * t_w$
- I/O cost for storing result: $(G * S_g/P) * IO$

The sequential bottleneck can be overcome by parallelizing the merging phase resulting in the Two Phase algorithm (2P) [Gra93].

2.2 Two Phase Aggregation

The algorithm is similar to above, except that the merging phase is parallelized by hash-partitioning on the GROUP BY attribute. Each node aggregates the locally generated tuples. The local aggregate values are then hash-partitioned and the nodes merge these local aggregate values in parallel. This is expected to be efficient even when the number of groups is not too small as the merging step will not be a bottleneck. The analytical cost model of the second phase of the algorithm becomes:

- receiving tuples from local aggregation operators: $(G_i/P)*m_p$ where $G_i = p*R_i*S_l$ and $|G_i| = |R_i|*S_l$.

- computing the final aggregate value for each group arriving to this node: $|G_i| * (t_r + t_a)$
- generating result tuples: $|G_i| * S_g * t_w$
- the tuples not processed in first pass need an extra read/write: $(1 - M/S_l) * G_i/P * 2 * IO$
- storing result to local disk: $(G_i * S_g/P) * IO$

However, even now there can be two problems when the number of groups is relatively large. First, there is duplication of aggregation work in the local and the merging phases which becomes pronounced as the number of groups increase. This is best understood by an example. Assume a 4 processor system. If a group is formed out of 8 tuples (2 on each processor), then each processor will do 2 aggregate operations in first phase, and then 4 aggregate operations to merge the local aggregate values adding to 12 operations for 8 tuples. Now assume that number of groups is twice, i.e. only 4 tuples (1 per node) form a group. Then each node will do one aggregate operation per tuple and 4 aggregate operations are required to merge in the second phase, adding to 8 aggregation operation for 4 tuples. Clearly, total number of operations now is more (2 operations/tuple vs. 1.5 earlier). Second, since a group value is being accumulated on potentially all the nodes the overall memory requirement can be large as the amount of memory required is proportional to number of group values seen. The third approach trades off network traffic in order to solve these two problems resulting in the following algorithm.

2.3 Repartitioning Algorithm

The Repartitioning algorithm (Rep) first partitions the data on the `GROUP BY` attributes and then aggregates the partitions in parallel. It eliminates duplication of work as each value is processed for aggregation just once. It also reduces the memory requirement as each group value is stored in one place only. The cost model of the algorithm is as follows.

- scan cost (IO): $(R_i/P) * IO$
- select cost involving reading, writing, hashing and finding the destination for the tuple: $|R_i| * (t_r + t_w + t_h + t_d)$
- repartitioning send/receive: $p * R_i/P * (m_p + m_l + m_p)$
- aggregate by reading and computing the cumulative sum: $|R_i| * (t_r + t_a)$
- the tuples not processed in first pass need an extra read/write: $(1 - M/S) * p * R_i/P * 2 * IO$
- generating result tuples: $|R_i| * S * t_r$
- storing result to local disk: $(p * R_i * S_g/P) * IO$

However, if the number of groups is less than the number of processors then, even in the best case, not all processors can be utilized. That is $R_i = R * max(S, \frac{1}{N})$ in the best case. Another potential problem is that the network cost can overshadow the benefit gained. This is not likely to be a problem with current-day high-speed, high-bandwidth networks but as we shall see a limited bandwidth network like 10Mbit/sec Ethernet can severely affect its performance.

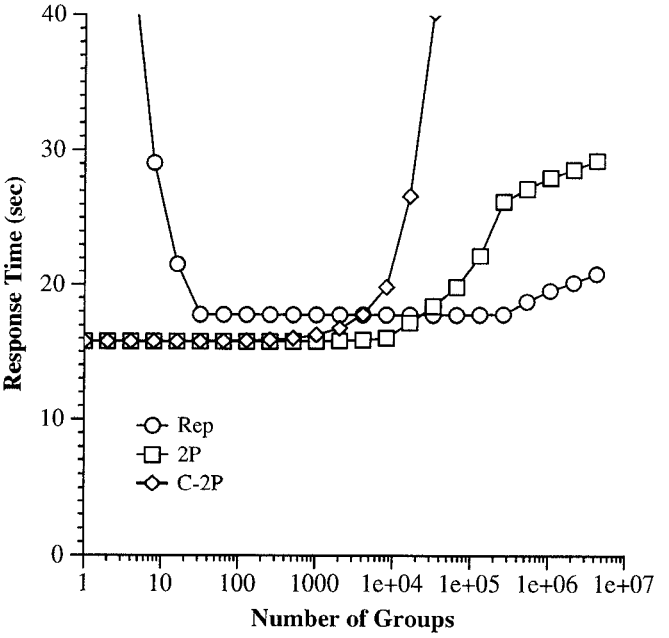

Figure 1: The Performance of Traditional Algorithms

The main performance characteristics of the the Centralized Two Phase, the Two Phase and the Repartitioning algorithms are summarized in Figure 1 for the 32 processor, one disk/node, configuration. It shows that, as expected, the Two Phase algorithms do well when the number of groups is small but the Repartitioning wins when the number of groups is large. It also shows that the network cost of repartitioning in the Repartitioning algorithms is not a serious problem if the interconnection has sufficient bandwidth (as in the IBM SP-2) but that wasted processors result in sub-optimal performance when number of groups is small. Finally, it is evident that each one of the algorithms performs poorly for some range of grouping selectivities and, therefore, none of these algorithms by themself will suffice for entire range of grouping selectivities. To further motivate the need for including the Repartitioning algorithm, Figure 2 shows the performance of the algorithms with no I/O costs as would be the case when aggregation or duplicate elimination is performed in a pipeline of operators.

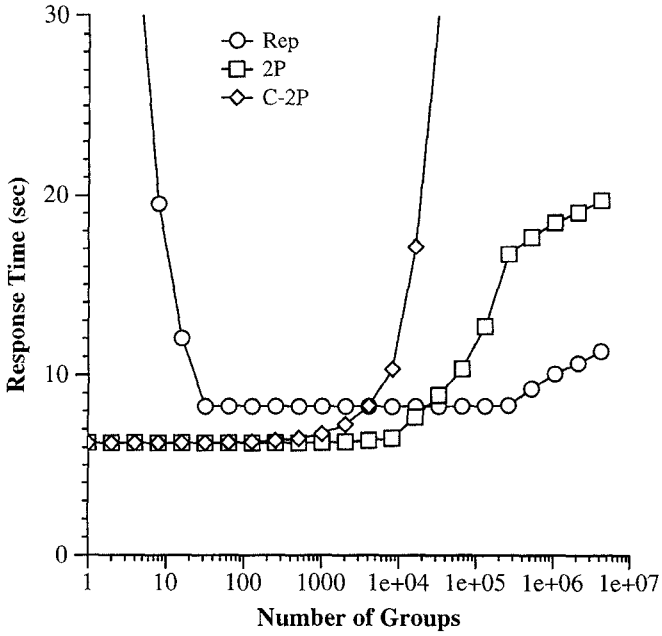

Figure 2: The Performance in an Operator Pipeline

3 The Algorithms

The natural question which arises from the previous discussion is: Given that the Two Phase and the Repartitioning algorithms work well for different grouping selectivities, can we somehow merge the two to get the best of both? Fortunately, the answer is yes and the following sections present three approaches to it. The first algorithm, Sampling (Samp) uses sampling to decide whether to use the Two Phase or the Repartitioning algorithm.

3.1 Sampling Based Approach

If we know the number of groups in the relation being aggregated then we can use the Two Phase algorithm if the number of groups is small or the Repartitioning algorithm if the number of groups is large. Unfortunately, the number of groups is not usually known beforehand. Sampling has been used effectively in past to estimate DBMS parameters [Ses92]. The general problem of accurately estimating the number of groups is similar to the projection estimation problem. However, in our case we only need to decide efficiently whether the number of groups in the relation is small or not because we have some leeway in the middle range where both the algorithms are expected to perform well. This does not require an accurate estimate of the number of groups, especially when it is large, making the problem significantly simpler than the general estimation problem which is fairly complex [BF93]. The basic scheme is as follows. First the optimizer will decide what is an appropriate switching point of the algorithms depending on the system characteristics. A reasonable number of groups for switching may be say, 10 times the number of processors available (a small number likely to lie in the middle range where both algorithms perform well). Call this the *crossover threshold*. Then, it can use the following algorithm to decide which scheme to use for aggregation.

```
sample the relation
find the number of groups in the sample
if (number of groups found < crossover threshold)
    use Two Phase
else
    use Repartitioning
```

The sampling can be implemented by letting each node randomly sample relation pages on its local disk. Page-oriented random sampling has been shown to be quite effective if there is no correlation between tuples in a page [Ses92]. Then the tuples in the sampled pages are aggregated using, possibly, the Centralized Two Phase algorithm. The number of groups obtained from the sample provides a lower bound on the number of groups in the relation. The trade-off here is between choosing a small crossover threshold and therefore a small overhead in sampling and decision making cost at the expense of using the Repartitioning algorithm when the number of groups is not too large. This trade-off is explored later.

It can be shown that the number of samples required is fairly small (about 10 times the crossover threshold) [ER61]. For example, for a crossover threshold of 320 (assuming 32 processor and 10 times as many groups) this is approximately 2563. This is likely to be less than 1% of any reasonably sized relation for small crossover thresholds.

The cost of this algorithm depends upon which algorithm is actually chosen. The cost of sampling and estimation is approximately as follows.

- let s be the sample size per node in bytes and $|s|$ be the number of tuples in the sample
- scan cost (IO): $(s/P) * rIO$
- select cost, getting tuple off data page: $|s| * (t_r + t_w)$
- local aggregation involving reading, hashing and computing the cumulative value: $|s| * (t_r + t_h + t_a)$
- generating result tuples: $|s| * S_l * t_w$
- message cost for sending result to coordinator: $(p * s * S_l/P) * (m_p + m_l)$

In the second phase these local values are merged by the coordinator. The number of tuples arriving at the coordinator are: $|G| = \sum |s| * S_l = |s| * N * S_l$ and $G = p * s * S_l$.

- receiving tuples from local aggregation operators: $(G/P) * m_p$

- computing the number of groups reading the tuples: $|G| * t_r$

However, since the sample is much smaller than the relation itself, the sampling cost is not significant.

3.2 Adaptive Two Phase Algorithm

The main idea in the Adaptive Two Phase (A-2P) algorithm is to start with the Two Phase algorithm under the common case assumption that the number of groups is small. However, if the algorithm detects that the number of groups is large it switches to the Repartitioning algorithm. The switching point can be determined as follows. The performance of the Two Phase algorithm worsens rapidly when it has to do intermediate I/O because of memory overflow. The crossover occurs when the intermediate I/O compensates for the additional network cost incurred in the Repartitioning algorithm. For high speed, high bandwidth networks the I/O cost will dominate. Hence, the number of groups at which the in-memory hash table in the local aggregation phase of the Two Phase algorithm is full and intermediate I/O is required seems to be a good switching point. (This is also the case from an implementation point of view as we avoid overflow processing in the first phase.) As mentioned earlier, the Repartitioning algorithm has better memory utilization and hence will much less likely require intermediate I/O.

The switching is independently decided by a node and is achieved as follows. A node upon detecting memory full stops processing its local tuples. It first partitions and sends the so far accumulated local results to the (global) aggregation phase thus freeing the memory. Then it proceeds to read and partition the remaining tuples and sending them to nodes they hash to.

The second (global aggregation) phase now can receive two kinds of "tuples". It can receive locally aggregated values and it can receive "raw" (perhaps projected) tuples which partition to the node. Both kinds of tuples can be merged into the same hash table: the aggregated values will have to be processed as in the global aggregation phase (e.g. for SQL average, the sum and the count will have to be added to the currently accumulated value), and the raw tuples will be processed as usual (e.g. for SQL average, the value will be added to the sum and the count will be incremented by one). The hash table will have the final aggregated values for all group values hashing to that node.

Here we must mention that [Gra93] points out another optimization to the Two Phase algorithm. It suggests that in the local aggregation phase, if the hash table is full then the locally generated tuples are hash partitioned and forwarded to the local aggregation phase. Hopefully, there might already be an entry there for that group which will save on I/O costs. If not, then the overflow processing is done at the destination node. As is evident, this optimization will improve the performance of the Two Phase algorithm, provided I/O is slower than network, because it may save some I/O. However, optimized Two Phase is inferior to the Adaptive Two Phase algorithm because:

1. There is a chance that there may be no group entry for a forwarded tuple on the destination node i.e. no saving on I/O and incurring an additional network cost.

2. All tuples still go through the local and global phases (despite the repartitioning) resulting in duplication of work.

3. Unlike the Adaptive Two Phase algorithm, which frees the memory used by the local aggregation phase as soon as the memory overflow is detected, this optimization continues to use it as it maintains the local hash table until all tuples are exhausted.

The Adaptive Two Phase algorithm, in practice, can have complex behavior as at any given point in time one set of processors in the computation may be executing the Two Phase algorithms while other are executing the Repartitioning algorithm. In the simple scenario, where all nodes execute the same algorithm, the approximate cost model of the Adaptive Two Phase algorithm is as follows. The first M/S_l tuples are processed like the Two Phase algorithm and the remaining tuples, if any, are processed like the Repartitioning algorithm. The cost model can be derived from this to be as follows:

- let $|P_i| = \min(\frac{M}{S_l}, |R_i|)$
- $TwoPhase(P_i)$: cost of processing P_i tuples as in the Two Phase algorithm
- $Repart(R_i - P_i + S_l * P_i)$ where the selectivity of the Repartitioning algorithm is replaced by $\frac{S*|R|}{|R|-|P_i|+|P_i|*S_l}$, if $|R_i| > |P_i|$ as it sees both the original tuples and partially aggregated values.

3.3 Adaptive Repartitioning Algorithm

Another way to combine the Two Phase and the Repartitioning algorithm is to start with the Repartitioning algorithm and switch to Two Phase if the number of groups is not large enough to justify using the Repartitioning algorithm. The Adaptive Repartitioning (A-Rep) algorithm is useful when the optimizer believes that the expected number of groups is large enough to use the Repartitioning algorithm. We expect that the algorithm will outperform the Adaptive Two Phase when the number of groups is very large as the first segment of tuples (the ones processed before switching) will not go through the additional phase of the Adaptive Two Phase algorithm.

However, in order to overcome the optimizer estimation errors, we must have a way switching to the Two Phase (actually Adaptive Two Phase) approach. The Adaptive Repartitioning algorithm does exactly that. It starts off with the Repartitioning approach. However, if it detects that the number of groups is very small after a certain number of tuples have been seen, it switches to the above mentioned Adaptive Two Phase algorithm. (Other methods for detecting potentially under-utilized processors can be used too.)

This switching is achieved as follows. When a node detects that it has seen too few groups given the number of seen tuples, it sends an "end-of-phase" message to all the nodes. Other nodes, upon receiving this message, follow suit by switching to the Adaptive Two Phase algorithm and sending their own "end-of-phase" message. All processes now do the Adaptive Two Phase algorithm where the global aggregation phase now uses the hash table left by the repartitioning phase.

An approximate and simplified cost model can be outlined as follows. Assume that first *initSeg* tuples seen by the first Repartitioning phase are used to judge if it is alright to continue with the repartitioning or not. Therefore first $(initSeg * N)$ tuples are processed with the cost incurred in the repartitioning algorithm cost. The remaining tuples are processed either with the cost of repartitioning algorithm or with the cost of hybrid algorithm depending on the decision. The actual cost of switching itself is negligible as the end-of-phase message is piggy-backed on the tuples being forwarded. The cost model is detailed below (assuming it correctly decides if the switching takes place of not).

- if $(S * |R_i| > threshold)$ then cost is same as that of the Repartitioning algorithm. Otherwise...
- *Repart*(*initSeg*): *initSeg* tuples are processed as in the Repartitioning algorithm
- $AdaptiveTwoPhase(|R_i| - initSeg + initSeg * S)$ tuples are processed as in Adaptive Two Phase algorithm with the S_g of the second phase is replaced by $\frac{S_g*(|R_i|*S_l)}{N*S_g*(|R_i|*S_l)+(S_l*initSeg)}$ as it can see tuples generated in the first repartitioning phase.

4 Analytical Results

We studied the performance of the three algorithms, the Two Phase and the Repartitioning algorithm using the analytical models described before. The main performance characteristics of the algorithms are evident from Figure 3. It shows the performance characteristics for the standard configuration with a high-bandwidth, high-speed network. The main observation is that all three algorithms are able to use the correct approach according to the workload demands. The Sampling algorithm has a constant (sampling) overhead for deciding

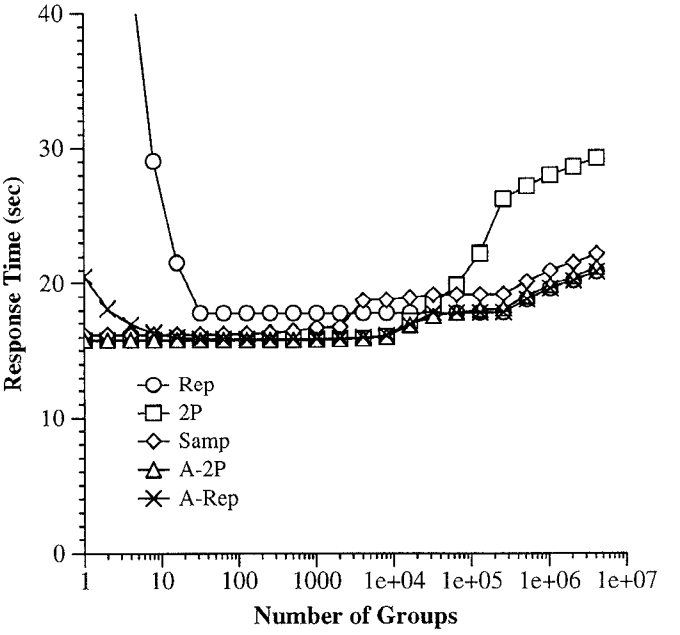

Figure 3: Relative Performance of the Approaches

which algorithm to use. The Adaptive Two Phase algorithm is able to track the appropriate algorithm with minimal overhead. This indicates that switching at the memory overflow point is a reasonable decision. Finally the Adaptive Repartitioning algorithms has no overhead for its common case but does suffer a little when the groups are too few (as in the beginning not all processors are used).

These characteristics are brought out again in Figure 4 which shows the performance of the algorithms on an eight processor, limited bandwidth network configuration for a 2 million tuple relation, as in our implementation. It shows that network cost becomes dominant for the Repartitioning approach and hence it makes sense to use the Repartitioning algorithm only when the number of groups is at least larger than the memory available. In other words, both Sampling and Adaptive Repartitioning algorithms must be careful to avoid doing Repartitioning if it is not essential. The Adaptive Two Phase algorithm switches from Two Phase to Repartitioning depending on memory usage. Hence it uses the Repartitioning approach only when it would be forced to do intermediate I/O otherwise. Hence it does not suffer as much due to the low bandwidth network. However, we believe that in practice most PDBMSs will have high bandwidth interconnects so that Figure 3 more accurately predicts the algorithms' performance than Figure 4.

We also show the expected scaleup of the algorithms under the low and high grouping selectivity in Figures 5 and 6 respectively. We are focusing on the extremes of the selectivity range, where it is most important to choose the correct algorithm.

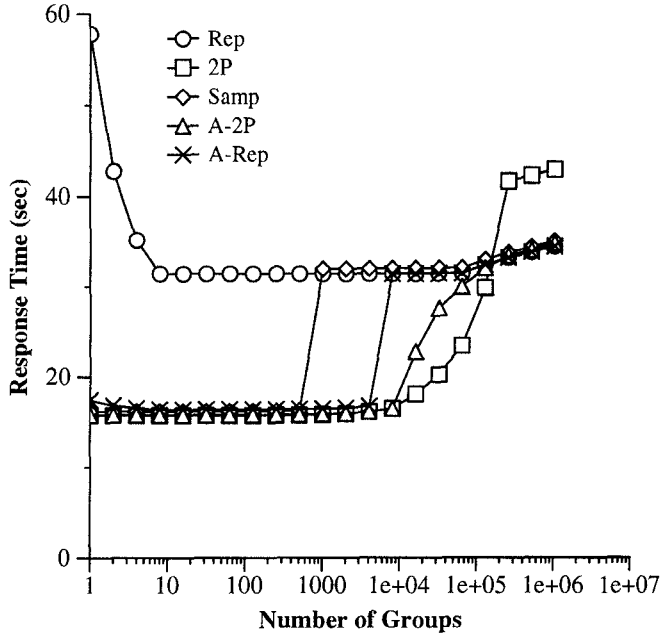

Figure 4: Performance on a Low-Bandwidth Network

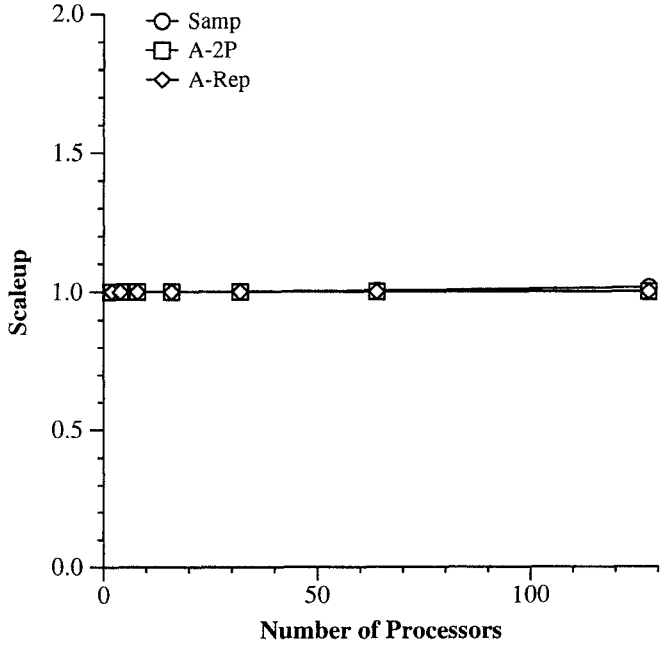

Figure 5: Scaleup of Algorithms: selectivity = 2.0e-6

The sampling overhead in the Sampling algorithm, as modeled and implemented, is proportional to the number of processors (i.e. it is a constant per processor). This is because the crossover threshold is proportional to the number of processors being $100 * N$ in our case. This overhead causes the Sampling algorithm to have small suboptimal scaleup in all cases as evident from Figures 5 and 6.

Both adaptive algorithms (Adaptive Two Phase and Adaptive Repartitioning) show almost ideal scaleup in the low selectivity case (Figure 5). Adaptive Two Phase works well here because it sticks with the Two Phase algorithm, which has good performance in this range; Adaptive Repartitioning works well because it quickly switches from Repartitioning to Two Phase when it discovers that there are few groups. In the high selectivity case (Figure 6), the performance of these algorithms is also good, because in this case Adaptive Two Phase switches to Repartitioning while Adaptive Repartitioning sticks with Repartitioning, which is the correct algorithm for the high selectivity case.

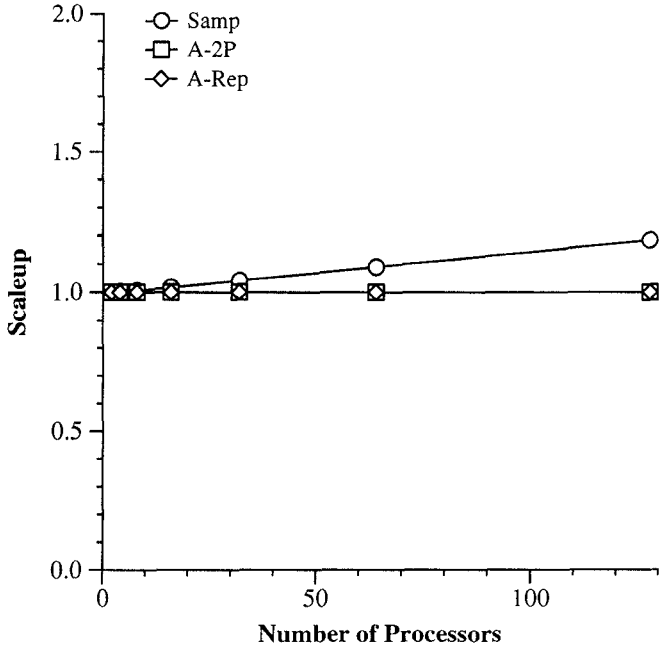

Figure 6: Scaleup of Algorithms: selectivity = 0.25

Finally, we studied the trade-off between the sample size and the performance of the Sampling algorithm. Figure 7 clearly indicates the trade-off between sample size (and hence sampling cost) and the penalty of using the incorrect algorithm for a 32 processor configuration. Increasing sample size lets one determine the number of groups more accurately increasing the crossover threshold at the increased sampling cost. In summary, for fast networks, one could use a small sample size as both Two Phase and Repartitioning approaches work efficiently and the sampling overhead can exceed the

difference. However, if the network is slow (or is of low bandwidth), one could afford to increase the sample size to ensure that one doesn't use the Repartitioning algorithms when the number of groups in the relation is not large.

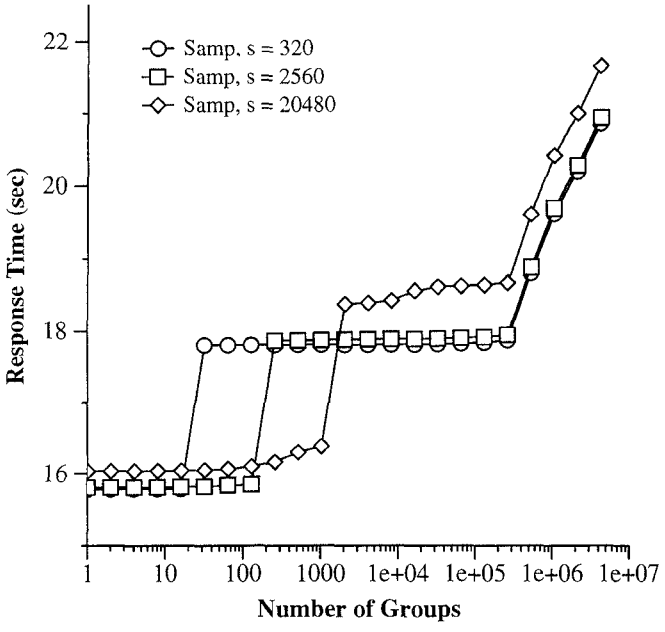

Figure 7: the sample size, performance trade-off

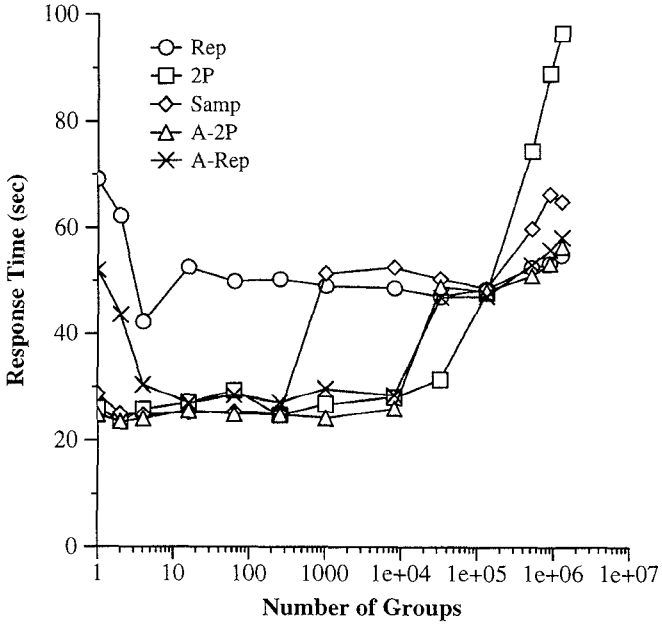

Figure 8: Relative Performance of the Approaches

5 Implementation Results

In order to validate the general conclusions from the analytical model, and to show that the new algorithms are straightforward to implement, we implemented the algorithms on a cluster of eight SUN SparcServer 20/51 workstations connected by a 10 Mbit/second Ethernet having one disk per node. We implemented the algorithms on top of the UNIX file system using the PVM parallel library [Oak93]. Our implementation had no concurrency control and did not use slotted pages. Hence the algorithms are significantly more CPU efficient than what would be found in a complete database system.

The 2 Million 100 byte tuples were partitioned in a round-robin fashion. Thus each node had 25 MB of relation. For efficiency reasons, we decided to "block" the messages into 2 KB pages.

The algorithms performed almost as expected from the analytical model. Figure 8 shows the performance of the Repartitioning, Two Phase, Sampling, Adaptive Two Phase and the Adaptive Repartitioning algorithms. As mentioned earlier, the low bandwidth Ethernet interconnect in our implementation makes the performance numbers comparable to the low bandwidth case in the analytical model.

We find that both Adaptive Two Phase and Adaptive Repartitioning algorithms do well and are able to change the method of computation on demand. The switching on memory full criterion for Adaptive Two Phase algorithm seems to perform well as it able to switch gracefully from Two Phase to Repartitioning approaches.

6 The Effect of Data Skew

All the performance measures reported so far assume that the relation is drawn from a uniform distribution. In this section we challenge that assumption and see how data skew affects the performance of the algorithms. As both the input relation size and the output (number of groups) size affects the performance of the algorithms, there can be two kinds of data skew in aggregate processing. These relate to whether the input or the output is non-uniform.

1. The number of groups/node is same but number of tuples/node is different. We call this *input skew.*

2. The number of tuples/node is same but number of groups/node is different. We call this *output skew.*

This can be contrasted with data skew in the join algorithms, where placement skew is analogous to input skew and join product skew is analogous to output skew[WDJ91]. However, since hash partitioning is not necessary for aggregate processing, the effect of data skew is significantly different.

The data skew, in essence, results in one or more nodes having to do more work than the under uniform distribution case. These skewed nodes then determine the overall performance of the algorithm. The expected effects of the different types of skew are as follows.

6.1 Input Data Skew

As mentioned, input data skew arises because of the variability in the input size across different nodes. Input skew mainly affects the input I/O cost. The rest of the processing is largely unaffected in the three algorithms. This will also be the case if the tuples are pipelined.

There are two cases of input skew:

1. When the number of groups is small, all algorithms (except Repartitioning) will eventually use the Two Phase approach. The skewed node will have to process more tuples (and do more I/O) severely affecting its performance. Repartitioning will be a little less affected as it only has to redistribute the excess tuples, but the I/O cost is likely to dominate.

2. When there is a large number of groups, all except Two Phase will use the Repartitioning approach (the Adaptive Repartitioning might switch to Two Phase before returning to Repartitioning). The skewed node will have to do more I/O and redistribute more tuples than others. The Two Phase algorithm will be affected even more as the skewed node has to process larger number of tuples locally.

6.2 Output Data Skew

Output skew provides a more interesting scenario. The skewed nodes have tuples with a larger number of distinct group values than the rest of the nodes. There are two cases for the skewed nodes.

1. When the number of groups is small, all algorithms (except Repartitioning) use the Two Phase approach. Since the total number of tuples is the same and there is no overflow processing, there is expected to be minimal performance degradation.

2. When the number of groups is large, the Adaptive Two Phase algorithm will change the skewed nodes to the Repartitioning approach, improving their performance. The other nodes will still do Two Phase. Since the nodes with several groups do repartitioning, the amount of intermediate I/O will be significantly reduced improving the overall performance of the algorithm.

   The Adaptive Repartitioning algorithm will first change to Adaptive Two Phase as it detects nodes with too few groups in them. Adaptive Two Phase will proceed as mentioned, resulting in similar performance.

   The Two Phase algorithm will suffer as the skewed node will have to do more I/O (than the uniformly distributed case). This is because the large number of groups on the skewed node will result in larger memory overflow and more intermediate I/O will be required.

   Both the Sampling (which will choose Repartitioning) and Repartitioning algorithms will suffer more than Adaptive Two Phase but less than the Two Phase algorithm. Repartitioning will make all nodes do an equal amount of work because all the group values will be more uniformly distributed. However, the algorithms will be unable to save on messages which is possible in the Adaptive Two Phase approach.

This indicates that the Adaptive Two Phase (and Adaptive Repartitioning) algorithm can outperform all of the static traditional approaches in this scenario. In other words, the new algorithms can be better than the best of the Two Phase and Repartitioning algorithms because they allows nodes to make independent decisions about which algorithms to run.

We evaluated the performance of the algorithms under output skew. The skew is modeled by assigning four out of the eight nodes all but four of the groups in the skewed relation. That is, four nodes have only one group value each, and the rest of the tuples are distributed among the remaining nodes. Figure 9 shows that both adaptive algorithms, Adaptive Two Phase and Adaptive Repartitioning, are able to do better than the traditional approaches[4]. Again, this is because of their ability to choose the correct approach for each node individually, something not possible in the traditional algorithms.

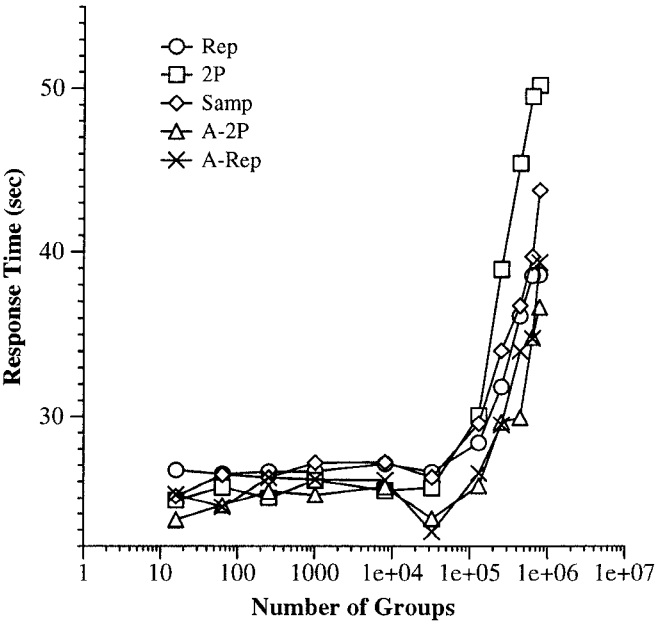

Figure 9: Performance under Output Skew

[4]Note that the Y-axis begin at 20 in order to zoom into the differences between the algorithms.

7 Conclusions

We show that each of the new algorithms are able to cope efficiently with unknown and varying grouping selectivities. Hence we are able to overcome the shortcomings of the traditional approaches, which do not work well for all the grouping selectivities.

If the system is to support only one algorithm, then the Adaptive Two Phase algorithm seems to be the best choice because in all cases it performs almost as well as the best of all other algorithms. However, if the system is to support multiple algorithms then the Adaptive Repartitioning could be supported as well to support efficient computation when the number of groups is very large (as expected to be the case in duplicate elimination). These two algorithms also show better performance under output data skew than the Two Phase and the Repartitioning algorithms.

References

[BBDW83] D. Bitton, H. Boral, et al. Parallel algorithms for the execution of relational database operations. *ACM Trans. on Database Systems*, 8(3), Sep. 1983.

[BCL93] K. P. Brown, M. J. Carey, and M. Livny. Managing Memory to Meet Multiclass Workload Response Time Goals. In *Proc. of 19th VLDB Conf.*, 1993.

[BF93] J. Bunge and M. Fitzpatrick. Estimating the Number of Species: A Review. *Journal of the American Statistical Association*, 88(421), March 1993.

[DGS+90] D. DeWitt, S. Ghandeharizadeh, et al. The Gamma database machine project. *IEEE Trans. on Knowledge and Data Engineering*, 2(1), March 1990.

[Eps79] R. Epstein. Techniques for Processing of Aggregates in Relational Database Systems. Memo UCB/ERL M79/8, E.R.L., College of Eng., Univ. of Calif., Berkeley, Feb. 1979.

[ER61] P. Erdös and A. Rényi. On a Classical Problem of Probability Theory. *MTA Mat. Kut. Int. Közl*, 6A, 1961. Also in Selected Papers of A. Rényi, v. 2, *Akademiai Kiado, Budapest.*

[Gra93] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2), June 1993.

[Oak93] Oak Ridge National Lab. *PVM 3 User's Guide and Reference Manual*, May 1993.

[Ses92] S. Seshadri. *Probabilistic Methods in Query Processing.* PhD thesis, Univ. of Wisconsin-Madison, Computer Sciences Department, 1992.

[SM82] S. Y. W. Su and K. P. Mikkilineni. Parallel Algorithms and Their Implentation in MICRONET. In *Proc. of 8th VLDB Conf.*, 1982.

[TPC94] TPC. TPC Benchmark^{TM} D (Decision Support). Working draft 6.5, Transaction Processing Performance Council, Feb. 1994.

[WDJ91] C. B. Walton, A. G. Dale, and R. M. Jenevein. A taxonomy and performance model of data skew effects in parallel joins. In *Proc. of the 17th VLDB Conf.*, 1991.