Processing Aggregates in Parallel Database Systems

Ambuj Shatdal
Jeffrey F. Naughton

Technical Report #1233

June 1994

Processing Aggregates in Parallel Database Systems[*]

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

Computer Sciences Technical Report # 1233

June, 1994

Abstract

Aggregates are rife in real life SQL queries. However, in the parallel query processing literature aggregate processing has received surprisingly little attention; furthermore, the way current parallel database systems do aggregate processing is far from optimal in many scenarios. We describe two hashing based algorithms for parallel evaluation of aggregates. A performance analysis via an analytical model and an implementation on the Intel Paragon multi-computer shows that each works well for some aggregation selectivities but poorly for the remaining. Fortunately, where one does poorly the other does well and vice-versa. Thus, the two together cover all possible selectivities. We show how, using sampling, an optimizer can decide which of the two algorithms to use for a particular query. Finally, we investigate the impact of data skew on the performance of these algorithms.

1 Introduction

SQL queries in the real world are replete with aggregate 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. The deceptive simplicity of aggregate processing is possibly the reason for this negligence. However, we find that though it looks straightforward, most parallel database systems (hereafter called PDBMSs) implement aggregate processing algorithms that are far from optimal. In this paper we study aggregate processing on shared nothing parallel database systems, and propose two different schemes for aggregate processing on PDBMSs.

The standard parallel algorithm for aggregation 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.

[*]This research was supported by NSF grants IRI-9113736 and IRI-9157357.

Briefly, the first approach we propose parallelizes the second phase of the traditional approach. The second approach 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. The algorithms we propose are simple enough that we hesitate to call them "new" algorithms, as we suspect that they may have already been thought of and perhaps even implemented. However, to our knowledge neither a description of these algorithms nor an analysis of their performance has appeared in the literature.

While these approaches are simple, their performance behavior is not obvious. The analytical models and the implementation on the Intel Paragon parallel super computer [Int93] show that the two proposed schemes are complementary in terms of performance. We find that whereas first approach works well when the number of result tuples is small, the second approach works better when the `GROUP BY` is not very selective. We show that it is relatively easy for the optimizer to decide which scheme is the best in a given scenario. This can be decided either on the basis of some known statistic about the relation, or by an efficient sampling based strategy. Finally, we study how these schemes perform in presence of data skew. We first characterize the data skew problem in aggregate processing and investigate the impact of skew on the performance of these algorithms.

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 is somewhat similar to the proposed two phase approach in that it uses local aggregation. We study its performance and show that it fares better than the traditional approach but worse than the proposed two phase approach. The second algorithm of Bitton et al. uses broadcast of the tuples and lets each node process the tuples belonging to a subset of groups. This is too 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 some issues in dealing with bucket overflow when using hash bases aggregation in a memory constrained environment.

The rest of the paper is organized as follows. Section 2 introduces aggregation, the previous approaches to aggregate processing and describes the two proposed approaches. An analytical evaluation of the different algorithms is presented in Section 3. In Section 4 we describe the implementation of the algorithms on the Intel Paragon and show their performance results. Section 5 shows how we can decide which algorithm to use given a particular scenario. In Section 6 we discuss the effect of data skew on aggregate processing. Section 7 offers our conclusions.

2 The Algorithms

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[1]. Thus the number of result tuples depends on the selectivity of the `GROUP BY` attributes. We define `GROUP BY` selectivity as $\frac{|Result|}{|Relation|}$. We find that it does indeed vary quite a lot. For example, in the TPC-D benchmark we found `GROUP BY` selectivities of 0.25, 0.00167 and 1e-5. The `HAVING` clause, when properly constructed (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. The following simple query will serve as the running example using relation R(tid,cardNo,amount).

select cardNo, avg(amount) **from** R
group by cardNo;

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 detailed below, 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.

We also present analytical cost models of the four approaches. 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. Fortunately, as we shall see in Section 3, in this study the results are robust under even fairly significant perturbations of the constants in the cost model, so even an approximate model is sufficient. The simplifying assumptions in the model include no overlap between CPU, I/O and message passing, sufficient network bandwidth, and that all nodes work completely in parallel thus allowing us to study the performance of just one node. With few exceptions (noted later in this paper) 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 and network speed were chosen to reflect the characteristics of the current generation of

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

Symbol	Description	Values
N	number of processors	32
Mips	MIPS of the processor	15
R	size of relation	400 MB
$\|R\|$	number of tuples in R	4 Million
$\|R_i\|$	number of tuples in R on node i	$\|R\|/N$
P	page size	4 KB
IO	effective time to read a page	3.5ms
p	projectivity of the aggregation	25%
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 add a tuple to current aggr value	300/Mips
t_m	time to merge two aggr values in sorted streams	150/Mips
t_s	time to compare and swap two keys	500/Mips
t_v	time to move a tuple	400/Mips
S	selectivity of the `GROUP BY`	$\frac{1}{\|R\|}$ to 0.5
S_l	phase 1 selectivity in Two Phase	$max(S * N, 1)$
S_g	phase 2 selectivity in Two Phase	$max(\frac{1}{N}, S)$
t_d	time to compute destination	10/Mips
m_p	message protocol cost per page	1000/Mips
m_l	message latency for one page	1.3 ms

Table 1: Parameters for the Analytical Models

commercially available multiprocessors (e.g. the Intel Paragon). The I/O rate was as observed on the Maxtor disk on the Paragon. The software parameters are based on instruction counts taken from the Gamma prototype and are similar to those in previous studies e.g. [BCL93]. In the following we assume that aggregation on a node is done by hashing.

In the remainder of this section we discuss previously proposed approaches to parallel aggregation and two new approaches that we have not seen discussed in the literature.

2.1 Traditional Approach

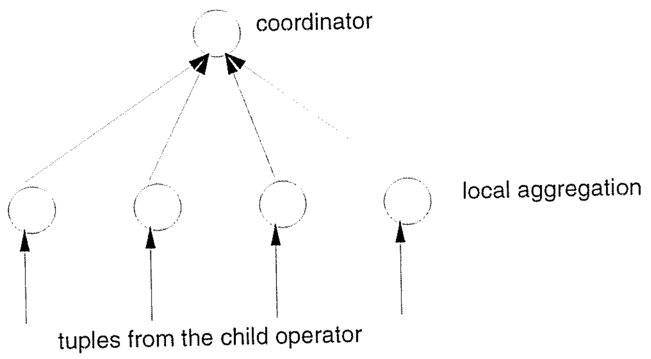


Figure 1: Traditional Scheme

The traditional approach, e.g. the one implemented in Gamma [DGS+90] and some commercial PDBMSs, is for each node to do aggregation on the partition of the relation on the node. These result in local (group, aggregate-value) tuples. These are sent to a central coordinator which merges the local results into the overall (global) aggregate value (Figure 1). In our example, it will result in each node computing local sum, and count for each group resulting in tuples like (cardNo = 1234, sum = 100, count = 2) being generated on each node. Assuming the second node (in a 2 node system) produces (cardNo = 1234, sum = 300, count = 3), the coordinator computes the final value to be (cardNo = 1234, avg = (100 + 300)/(2 + 3) = 80).

As we will show, this approach works well if the number of resulting tuples is very small. But as soon as the selectivity of the GROUP BY becomes moderate, the central coordinator node starts becoming a bottleneck. In TPC-D queries, for example, there is a GROUP BY with selectivity as low as 0.25 (i.e. on average, 4 tuples form a group). Hence, in general using single node global aggregation forms a serial bottleneck at that node.

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

- scan cost (IO): $(R_i/P) * IO$
- select cost, getting tuple out of data page: $|R_i| * (t_r + t_w)$
- local aggregation involving reading, hashing and computing the cumulative value: $|R_i| * (t_r + t_h + t_a)$, or if using sorting then sorting and scanning/aggregating the cumulative value: $|R_i| * \log |R_i| * t_s + |R_i| * (t_v + t_a)$

- generating result tuples: $|R_i| * S_l * t_w$
- message cost for sending result to coordinator: $(R_i * S_l/P) * (m_p + m_l)$

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: $|G| * (t_r + t_a)$ or using merging of sorted streams: $|G| * t_m$
- generating final result: $|G| * S_g * t_r$
- I/O cost for storing result: $(G * S_g/P) * IO$

2.2 Approach of Bitton et al. : Hierarchical Merging

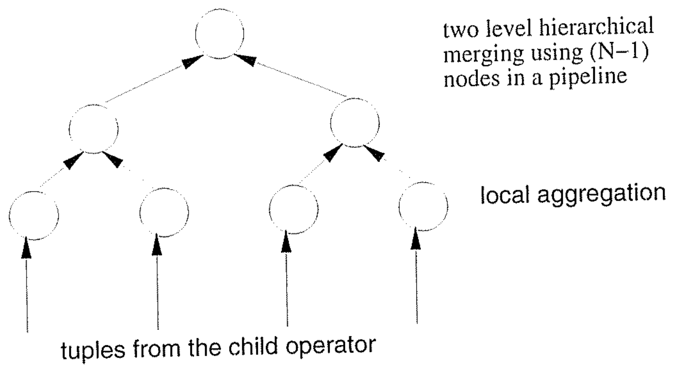

two level hierarchical merging using (N−1) nodes in a pipeline
local aggregation
tuples from the child operator

Figure 2: Local Aggregation with Hierarchical Merging

The first algorithm proposed in [BBDW83] first makes each node do aggregation on locally resident data like the traditional approach. However, instead of one node doing the merging of these local aggregate results into final ones, it utilizes a (pipelined) binary merging scheme, thus off-loading some of the work from the final node (see Figure 2). However, the final node still has to generate all the final aggregate values, as it has to do the final merge.

This approach also works well if the number of resulting tuples is very small. It even handles the moderate selectivity ranges well because of the hierarchical merging. However, when the number of tuples become sufficiently large, its performance declines as the final merging phase becomes the bottleneck.

The central merging phase of the traditional approach is now replaced by a pipelined hierarchical merging. This necessitates addition of a sorting step in the hashed based aggregation which consumes $|R_i * S_l| \log(|R_i| * S_l) * t_s + |R_i * S_l| * t_v$ amount of CPU. Otherwise, the cost of the first phase remains identical. Assuming ideal pipelining, the cost is determined the bottleneck node in the pipeline. Thus the cost will be determined by the maximum of the costs on each node which are computed as follows, where n is the number of tuples arriving to that node and f is the fraction reduction (up to at best 0.5) in the number of tuples after the merge.

- receiving tuples from previous operator in the pipeline: $(n/P) * m_p$
- merging the two arriving aggregate streams: $n * t_m$
- generating result tuples: $n * f * t_w$
- message cost for sending result to next operator: $(n * f/P) * (m_p + m_l)$
- storing result by the last operator: $p * R * S * IO$

We propose two possible alternatives for GROUP BY aggregate evaluation using hash based redistribution. The first one extends the traditional scheme by parallelizing the second phase. The second approach redistributes the data first by hashing on the GROUP BY attributes and then computes the aggregates locally thus avoiding the second phase necessary in the previous approaches.

2.3 Two Phase Parallel Aggregation

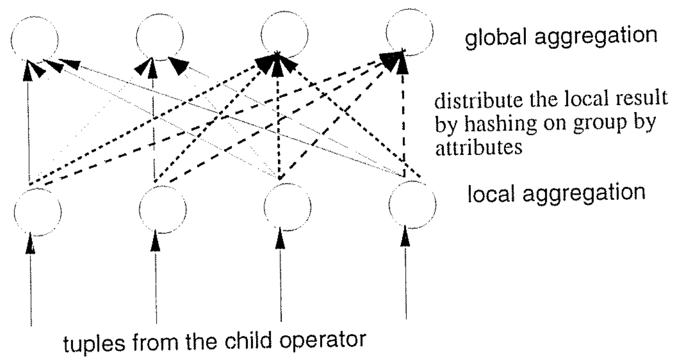

global aggregation
distribute the local result by hashing on group by attributes
local aggregation
tuples from the child operator

Figure 3: Two Phase Aggregation Scheme

In the first phase, the tuples after being read from disk are passed to the aggregate operator on the same node. This local "message passing" is more efficient than redistribution. Each aggregate operator then computes the aggregate on the set of tuples its scan operator generates.

In the second phase, the local aggregate results are partitioned on the GROUP BY attributes and sent to one of the "global aggregation" operators that are running on each of the nodes. The global aggregation operator merges these individual local results into the final aggregate values for each group. In our example query, each node will send tuples of the form (cardNo, sum, count) to the global aggregation operator. The global aggregation operator will compute the final result by summing up the tuples belonging to the same group received from all the nodes and dividing by the total count. This operation for different groups is done in parallel by redistributing these tuples on 'cardNo' attribute, so as to avoid the bottleneck in the traditional approach.

The main performance difference from the traditional approach is that the second phase is now parallel. The first phase remains the same except that now the destination for the tuples containing local aggregation information has to be computed adding $|R_i| * S_l * t_d$ to the cost. The second phase is parallelized becoming:

- 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: $|G_i| * (t_r + t_a)$ or using merging of sorted streams: $|G_i| * t_m$
- generating result tuples: $|G_i| * S_g * t_r$
- storing result to local disk: $(G_i * S_g/P) * IO$

In practice, however, there will be an additional but negligible overhead of sending control messages to all the nodes and receiving control messages after the termination of the operators.

2.4 Parallel Aggregation by Redistribution

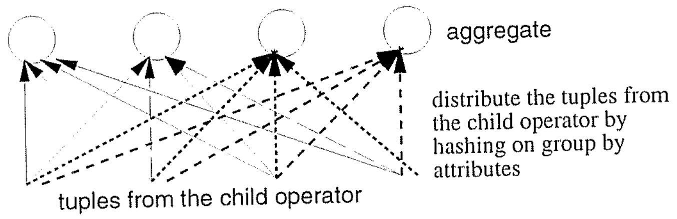

aggregate
distribute the tuples from the child operator by hashing on group by attributes
tuples from the child operator

Figure 4: Repartitioning Scheme

The second algorithm is motivated by the fast message passing in today's multi-computers. The tuples after being read are redistributed by hashing on the `GROUP BY` attributes (just like in a join operation where tuples are redistributed on the join attribute) as in Figure 4. Each node now does a `GROUP BY` as in a single node system and the result produced is final for that group because tuples belonging to a group are on only one node. In our example, all the tuples with cardNo = 1234 will be on one node and the result of the average will be final. In this method, therefore, we avoid a second phase in aggregation at the cost of redistributing the entire relation (which is not too expensive in current day multi-computers). We know of one commercial PDBMS that uses this approach for vector aggregates. Unfortunately, this strategy does not work efficiently as given when the number of groups is comparable or less than the number of nodes available because then not all the nodes will be exploited. We expect this strategy to be efficient when the number of groups is more than the number of processors available.

The cost model for repartitioning approach 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 and receive: $R_i/P * (m_p + m_l + m_p)$
- aggregate by reading and computing the cumulative sum: $|R_i| * (t_r + t_a)$ or by sorting and scanning/aggregating the cumulative value: $|R_i| * \log |R_i| * t_s + |R_i| * (t_v + t_a)$
- generating result tuples: $|R_i| * S * t_r$
- storing result to local disk: $(p * R_i * S_g/P) * IO$

However, one must observe that if the number of groups is less than the number of available processors, then all processors can not be exploited by the basic scheme. That is $R_i = R * max(S, \frac{1}{N})$. Implemented as is, it will show poor performance when number of groups is small, i.e. $S > \frac{1}{N}$.

3 Analytical Results

We studied performance of the four approaches under varying assumptions. The main performance characteristics are evident from Figure 5. It shows the performance characteristics of the

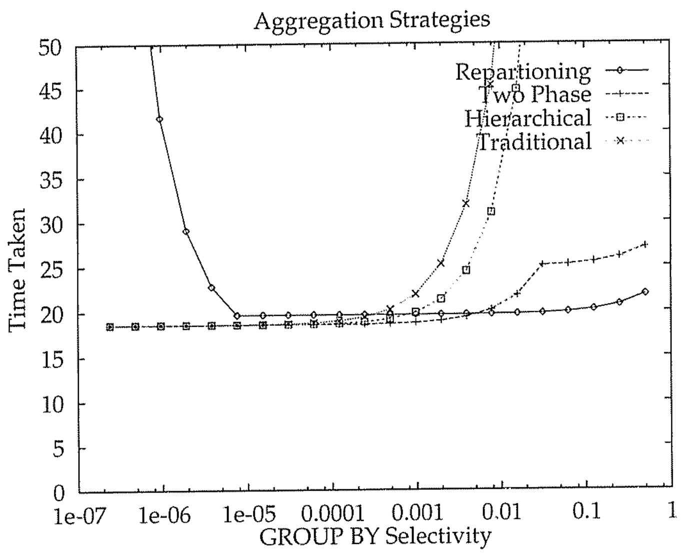


Figure 5: Relative Performance of the Approaches

algorithms for a standard configuration: 32 nodes each with 25 MIPS CPU, sufficient memory, 1 disk and an Intel Paragon-like network. The relation size was 4 million tuples. Two main observations are common to both.

1. The two phase approach can easily replace the previous approaches because it is never worse than them and the previous approaches suffer at higher selectivities.

2. The two phase approach does much better than repartitioning till number of groups is less than the number of processors. Then both algorithms perform similarly but finally the repartitioning approach is significantly better.

The reason why two phase scheme does better at low GROUP BY selectivities is that the overhead of repartitioning the whole relation is avoided and since very few tuples are generated the second phase does not affect the overall performance. As mentioned earlier, the repartitioning approach does not exploit all nodes when number of groups is less (or comparable) than the number of nodes. However for high selectivities, the repartitioning scheme does better because it does not duplicate the aggregation work, which the two phase scheme is forced to do for high selectivities thus nullifying the advantage of avoiding the redistribution by the merging of local aggregate values in the second phase.

The above results hold even when the parameters are changed. Figures 6, 7, 8, 9 show graphs with 4 times faster CPU, 4 times faster network, 4 times faster I/O and 4 times less memory respectively.

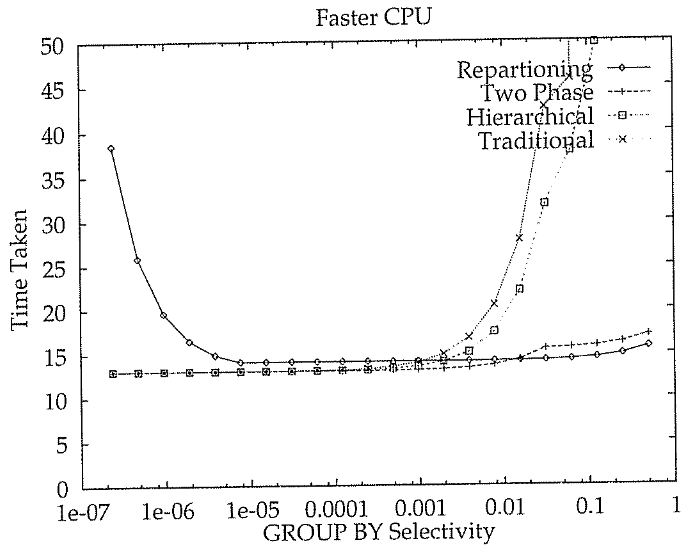


Figure 6: Performance with 4 times faster CPU

Most of the differences are expected. The faster CPU results in faster response time for both the algorithms; the faster network improves the relative performance of the repartitioning approach because it uses the network more than the others; the faster I/O improves the overall response time of all algorithms. However, we find that if we constrain memory a little such that not all of the data structure for the *group by* information (e.g. a hash table) can be kept in memory, then we find that the all except the repartitioning approach suffer significantly because of memory thrashing. Memory thrashing is modeled simply by assuming that a reference will miss and suffer a page fault with the probability that the group it refers to is not in memory.

This points to an interesting observation regarding the behavior of the algorithms. In the hash based aggregation a hash table entry is maintained per group. Since tuples belonging to

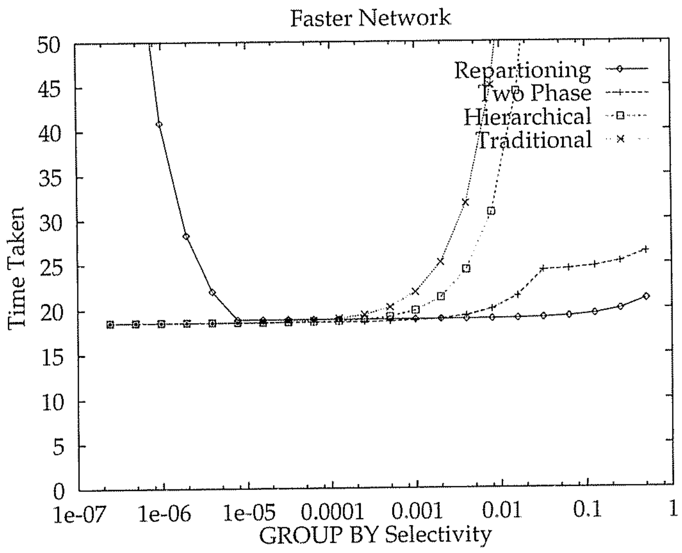


Figure 7: Performance with 4 times faster Network

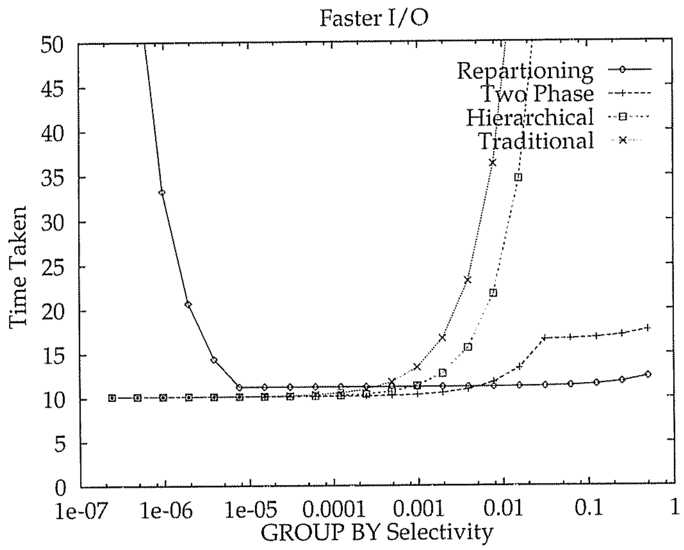


Figure 8: Performance using 4 disks per node in parallel

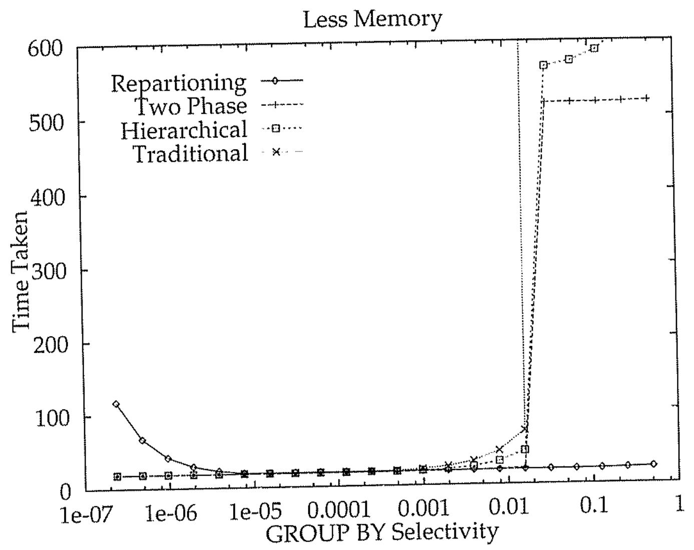
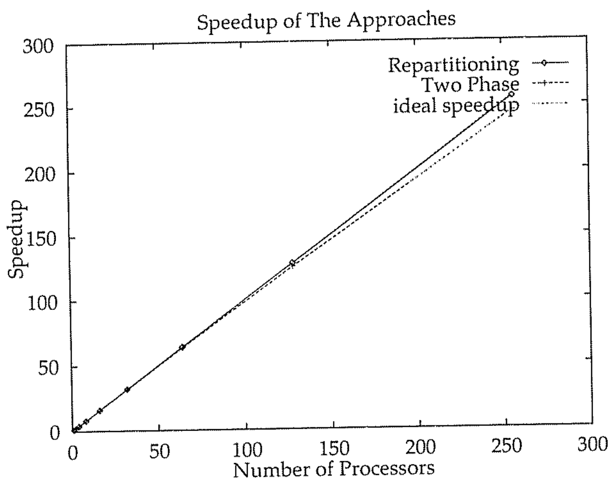


Figure 9: Performance in a Constrained Memory Configuration

a group are initially randomly distributes across the nodes, each node is likely to have tuples belonging to a particular group. Hence there will be a hash table entry for that group on all the nodes containing tuples belonging to the group. Thus the hash table is, in a way, replicated across the nodes of the system. In contrast, the repartitioning algorithm brings all tuples belonging to a group to a particular node and hence there is only one hash table entry of that group in the entire system. Hence the total memory requirement of the redistribution approach is significantly smaller. This smaller memory requirement results lets it handle larger number of groups without thrashing or using alternative (e.g. multi-bucket aggregation) approaches.



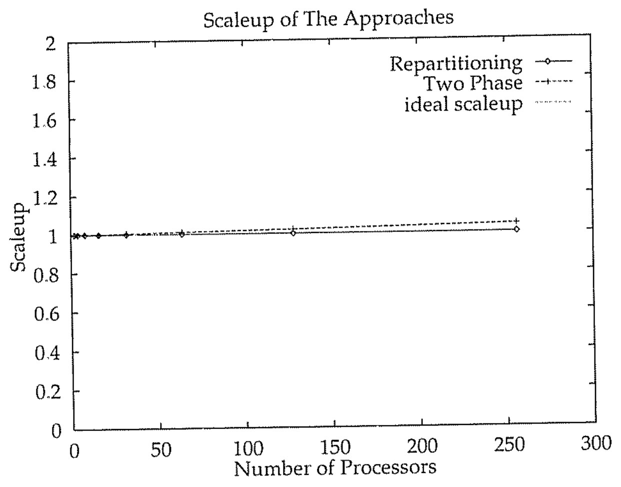
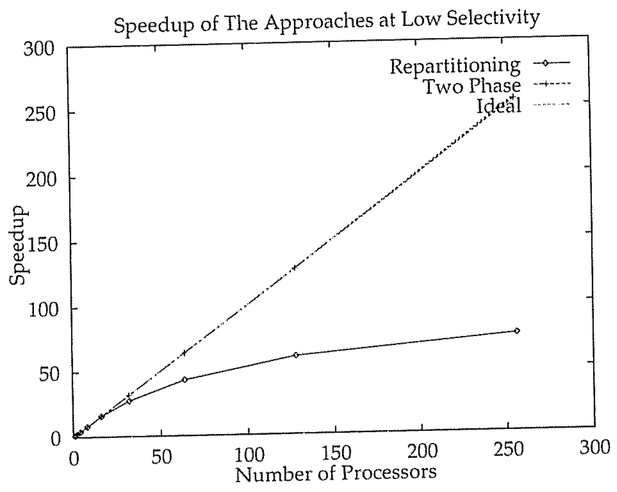


Figure 10: Speedup and Scaleup Characteristics of the Approaches at $\frac{1}{2^{11}}$ Selectivity

Figure 10 shows the speedup and scaleup for a selectivity of $\frac{1}{2^{11}}$ which lies in the "middle" range of the selectivity where both the algorithms are expected to perform well. We find that the speedup and scaleup characteristics of both the approaches are very close to ideal, the repartitioning approach being a little better. This is expected in the middle selectivity range because both the algorithms (modulo data skew) are fully parallelizable with little overhead. The two phase algorithm suffers the overhead of the second phase (which grows with the selectivity) whereas the overhead in the repartitioning scheme is the redistribution of the relation. As a system grows larger, the repartitioning algorithm will show better performance as evident from the speedup results.



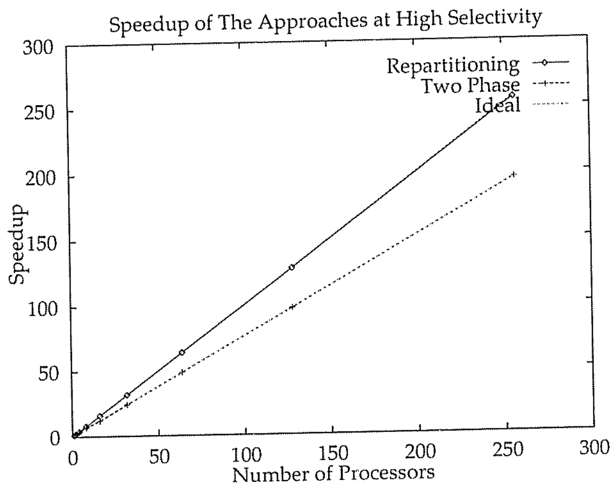


Figure 11: Speedup of the Approaches at low, $\frac{1}{2^{18}}$, and high, $\frac{1}{16}$ Selectivities

However, as evinced by Figure 11, the speedup characteristics of the algorithms at the low and high `GROUP BY` selectivities are quite different. At low selectivity, the two phase algorithm clearly wins as the repartitioning approach is unable to exploit all the processors. For high selectivities the repartitioning approach shows a significantly better speedup than the two phase. This further supports our claim that we need both the approaches in a PDBMS.

4 Implementation Results

In order to further investigate the performance of the two proposed approaches, we implemented them on the Intel Paragon parallel super computer. The Intel Paragon has a shared-nothing architecture with 64 nodes each having a i860 CPU, 16 MB memory and a Maxtor MXT-1240S disk. The nodes are connected by a high bandwidth, low latency interconnection network. We implemented the algorithms on top of the OSF/1 file system using the Paragon message passing library.

Since the using multiple processes per node was not recommended for performance reason, and we did not have a thread package, we decided to do our own mini thread management among the related processes. 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. As we will see, the performance numbers therefore match those of a fast CPU with a constrained memory in the analytical model.

We used 32 nodes of the system in our experiments. The 4 Million 100 byte tuples were partitioned in a round-robin fashion. Thus each node had 12.5 MB of relation. We decided to "block" the messages into 4 KB pages because sending large messages is more efficient than sending several smaller messages.

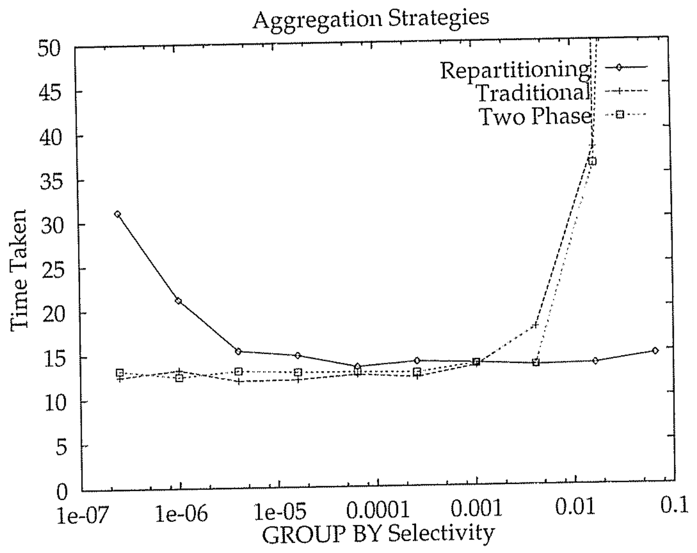
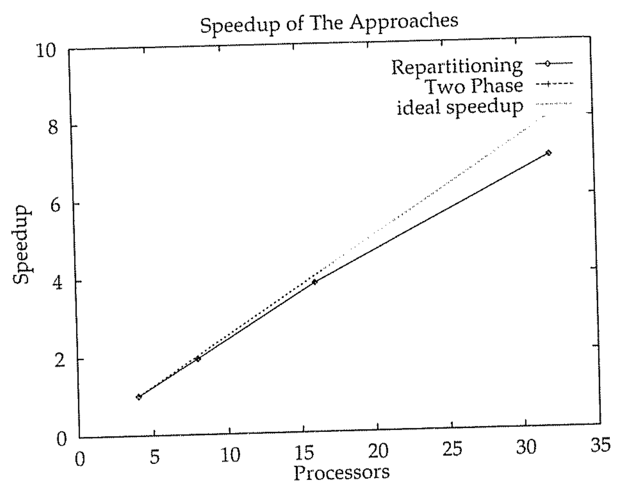


Figure 12: Relative Performance of the Approaches

The algorithms performed almost as expected from the analytical model. Figure 12 shows the performance of the traditional, two phase and the repartitioning approach.

As mentioned earlier, the low CPU cost of our implementation makes the performance numbers comparable to the fast CPU case in the analytical model. However there are some differences which must be noted. First, in the low selectivity range, the repartitioning approach does not do as bad as predicted by the analytical model even when only one processor is being used for aggregation. We investigated this and found that the reason was that the CPU utilization of the node being used is significantly higher than the average utilization of the CPU when all processors are being used. This shows that one disk per node is not sufficient to drive the processors at their full capacity. Second, in the high selectivity case, the memory thrashing results in a significantly poorer performance of the two phase and the traditional approaches. This is predicted, but not accurately, by the analytical model using constrained memory. The repartitioning approach requiring significantly less memory is not affected.

Figure 13 shows the speedup and scaleup for a selectivity of $\frac{1}{2^{10}}$ which lies in the "middle" range of the selectivity where both the algorithms are expected to perform well. We find that even in practice, the speedup and scaleup characteristics of both the approaches are close to ideal.



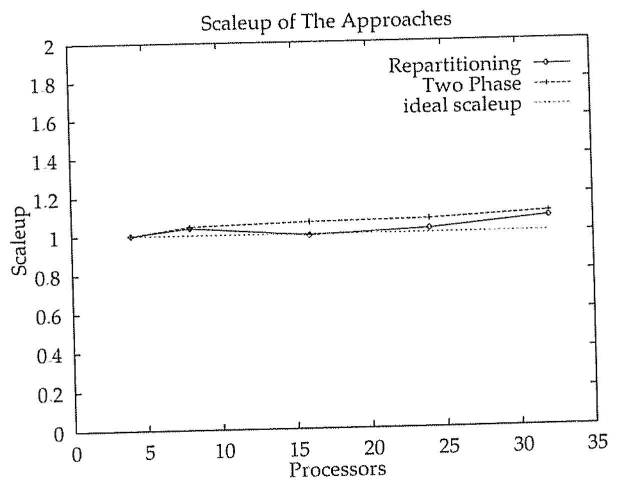


Figure 13: Speedup and Scaleup Characteristics of the Approaches at $\frac{1}{2^{10}}$ Selectivity

5 Selecting the Appropriate Approach

From the discussion above it is clear that the two proposed algorithms work well in their respective domains. The two phase approach works well when the number of groups is small and the repartitioning works well when the number of groups is large. In the middle ranges both algorithms show comparable performance. We also saw that this general observation does not change even with significant changes in system parameters.

Quantifying it a little, we find that two phase scheme is much better till GROUP BY selectivity of about $\frac{N}{|R|}$ because in this range the repartitioning scheme can not exploit all the processors. Between approximately $\frac{N}{|R|}$ and $\frac{1}{10*N}$ both algorithms have comparable performance though the two phase is slightly better. For selectivities above $\frac{1}{10*N}$ the repartitioning approach is significantly better. Here the two phase approach is not able to do enough reduction in the number of tuples through local aggregation in the first phase as the selectivity is very small. Therefore the second phase has to do a lot of work in global aggregation thus significantly increasing the overhead.

Evidently, the performance of these algorithms depends critically on the GROUP BY selectivity of the aggregate. In many cases (for example, if the aggregate is on an indexed column of a stored table) this selectivity may be available from the statistics stored in the system catalogs. However, in many cases such a statistic may not be available (for example, if the aggregate is on a column in a relation that is an intermediate result in a query evaluation plan). We propose a sampling based scheme for determining the selectivity in such cases.

The general problem of accurately estimating the number of groups (and hence the GROUP BY selectivity) is similar to the projection estimation problem. It is fairly complex and has received a lot of attention in the statistics literature [BF93].

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 a lot of leeway in the middle range where both the algorithms perform well. This does not require an accurate estimate of the number of groups (especially when they are large) making the problem significantly simpler than the general estimation problem. Our 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). 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
```

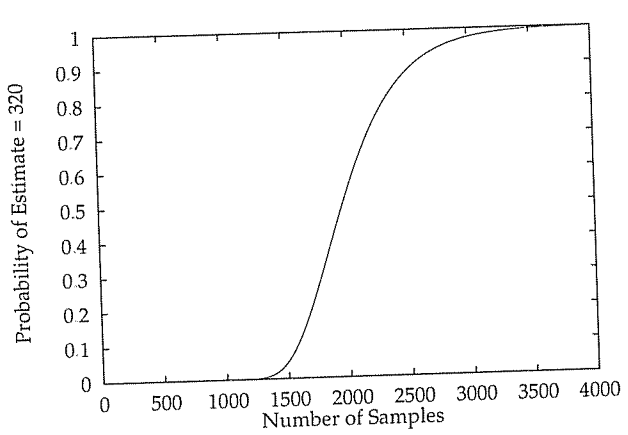
It can be shown that the number of samples required is fairly small. For example, for a crossover threshold of 320 (assuming 32 processor and 10 times as many groups) this is approximately 2563. This is likely to be less than 1% of any reasonably sized relation for small crossover thresholds.

The probability that the we find G groups in a sample (with replacement) of size S when there are exactly G groups in a uniformly distributed relation is given by the following recurrence equation.

$$P(g,s) = P(g,s-1) * \frac{g}{G} + P(g-1,s-1) * (1 - \frac{g-1}{G})$$

where $P(1,1) = 1$ and $P(g, s < g) = 0$. Solving for $P(G,S)$ gives the desired probability. This equation can be solved quite efficiently using dynamic programming. Note that the probability does not depend on the size of the relation. For any fixed value of $P(G,S)$, the number S grows approximately as $G * \log G$ [ER61]. This ensures that the number of samples required is not very large even if the crossover threshold is significant.

What is an appropriate value of $P(G,S)$ and hence the number of samples? The graph on the left in Figure 14 illustrates the shape of a typical probability curve, here for $P(320,S)$. It is evident that we must choose the probability in the upward convex part of the curve so as to ensure that we are stable in terms of estimates i.e. the estimate will not vary widely with addition or deletion of a few samples. In other words, the probability of underestimating the number of groups (e.g. estimating 320 groups when there are 322 groups) will fall rapidly as exemplified by the right graph. In our case we want to minimize the probability that we will underestimate the number of groups, since we will never overestimate. estimation. In practice the error would be how often we estimate the number of groups as "small" when it is not so, e.g. how often we estimate the number of groups to be less than or equal to, say 320, when in fact there are more groups. In practice, the cost of making a minor mistake is small and Figure 14 shows that we are not likely to make any major mistakes in estimation. Choosing $P(\textit{crossover threshold}, S) \geq 0.9$ implies that 90% of the time we will guess the number of groups exactly, and Figure 14 suggests that even in the 10% of the time we underestimate, it will not be by a significant amount.



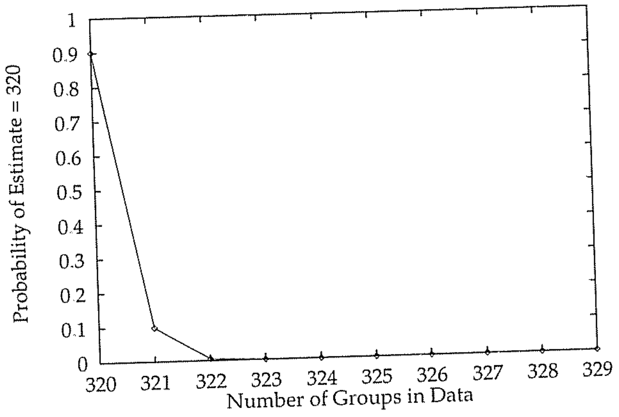


Figure 14: Accuracy in Estimating G

All the performance measures reported so far assume that the relation is drawn from a uniform distribution and is evenly declustered across the participating nodes. In the next section we challenge that assumption and see how data skew affects performance of the two approaches.

6 The Effect of Data Skew

The two main forms of data skew that can occur in aggregate processing are the following. First, instead of about equal number of tuples forming each group, the number of tuples forming a group may vary widely. That is, under uniformity assumption a GROUP BY selectivity of 0.01 implies that each group is formed by about 100 tuples, but in a data skew scenario it is possible to envision that a few groups, say, are formed with 5 tuples but others are formed with 1000 tuples averaging to a selectivity of 0.01. Upon repartitioning, such unequal distribution of group sizes will result in different processors getting different number of tuples. We call this skew the *selectivity skew*. In the second form, the child operator of the aggregate operator on each node produces a different number of tuples. We call this the *placement skew*. In our case, this implies that initially different nodes have different number of tuples.

These two kinds of skew will manifest themselves differently depending on the algorithm. Selectivity skew alone will not affect the performance of the two phase approach as far as the tuples belonging to a group are uniformly generated by different scan operators. This ensures that each node will get equal number of tuples albeit the number of tuples belonging to a group will vary. However, when the selectivity is high, the selectivity skew results in some groups having large number of tuples which can be aggregated in phase one of the algorithm thus reducing work for phase two. Hence, the two phase approach can potentially have better performance for high skew, high selectivity case. In the repartitioning approach, selectivity skew will result in some nodes getting more tuples to process than the others because all tuples belonging to a group will be on one node and hence its performance will be adversely affected.

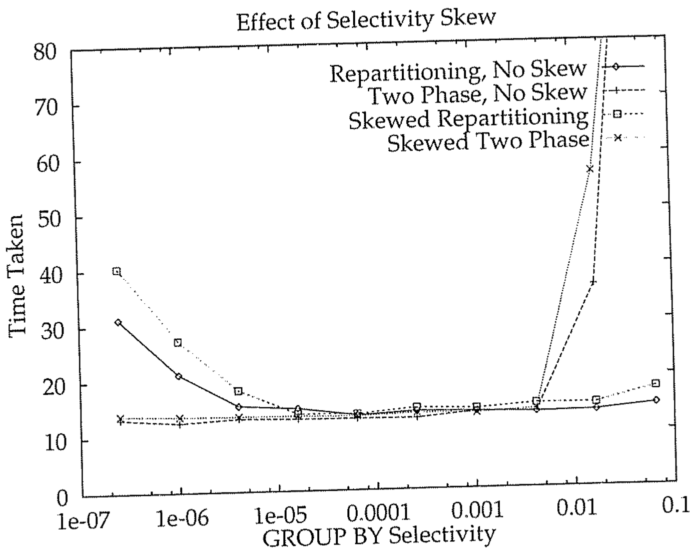


Figure 15: Effect of Selectivity Skew on Performance of the Approaches

Figure 15 shows the performance of the algorithms obtained from the implementation, the selectivity skew being modeled using a Zipf distribution with parameter $\theta = 0.1$ which results in the worst loaded node getting about two times as many tuples as the no-skew case. It shows that the two phase approach is not affected significantly whereas the repartitioning approach does a little poorer in all cases where all processors are utilized (i.e. number of groups $\geq$ number of processors). The I/O bottleneck of single disk, however, minimizes this difference because the CPUs wait for the disks which finish at about the same time as all nodes still have same amount of I/O requirements.

Next, we turn to placement skew, which we modeled by generating the database so that one node had twice as many tuples as the other nodes. Placement skew will affect the CPU performance of the two phase approach as all the tuples generated must be processed locally thus increasing the amount of work for the skewed node. The second phase of the algorithm should not be affected by skew as the number of tuples arriving there will be a function of the number of groups and the number of nodes and not the number of tuples in a group or initial placement.

Additionally, in our case, placement skew determines the amount of disk I/O a node has to perform. Hence, both algorithms will be equally hit in I/O performance. This is brought out by the implementation results in figure 16. The placement skew is modeled by having twice as many tuples on the skewed node as the no-skew case implying the worst case I/O cost is twice as much.

In summary, selectivity skew affects the repartitioning algorithm more than the two phase algorithm, while placement skew affects the two phase algorithm more than the repartitioning algorithm. However, over the ranges of skews that we tested, the impact of skew was not significant enough to change the relative performance of the algorithms in any important way.

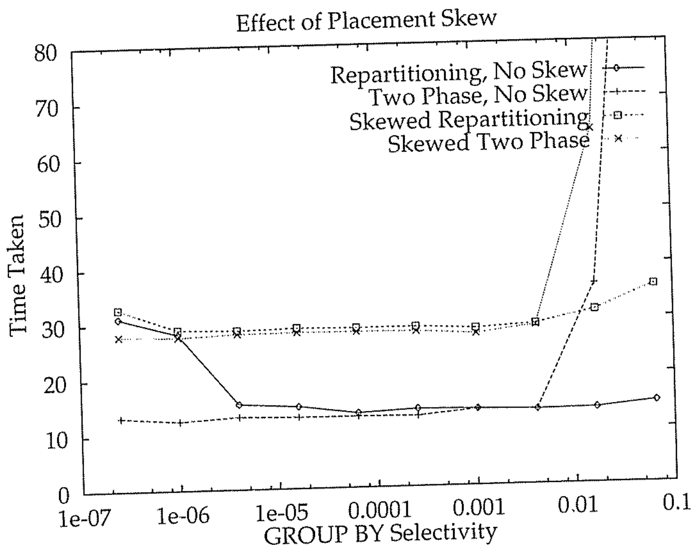


Figure 16: Effect of Placement Skew on Performance of the Approaches

7 Conclusions

We have shown that aggregate processing, though apparently simple, has not so obvious trade-offs. We show that each of the two approaches proposed do well in a limited domain decided by the selectivity of the GROUP BY attributes and that it is easy for an optimizer to use sampling to decide which algorithm to use. We also showed that the estimate of selectivity need not be very accurate in order to get good performance, but if we do not use the appropriate algorithm towards the fringes of the selectivity poor performance will result. We also showed what problems can be caused by data skew in aggregate processing and how it affects the relative performance of the two approaches.

References

[BBDW83] Dina Bitton, Haran Boral, David J. DeWitt, and W. Kevin Wilkinson. Parallel algorithms for the execution of relational database operations. *ACM Transactions on Database Systems*, 8(3):324–353, September 1983.

[BCL93] Kurt P. Brown, Michael J. Carey, and Miron Livny. Managing Memory to Meet Multiclass Workload Response Time Goals. In *Proc. of 19th VLDB Conf.*, pages 328–341, 1993.

[BF93] J. Bunge and M. Fitzpatrick. Estimating the Number of Species: A Review. *Journal of the American Statistical Association*, 88(421):364–373, March 1993.

[DGS+90] D. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H.-I Hsiao, and R. Rasmussen. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

[Eps79] Robert Epstein. Techniques for Processing of Aggregates in Relational Database Systems. Memorandum UCB/ERL M79/8, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, February 1979.

[ER61] Paul Erdös and Alfred Rényi. On a Classical Problem of Probability Theory. *MTA Mat. Kut. Int. Közl*, 6A:215–220, 1961. Also in Selected Papers of Alfred Rényi, volume 2, pages 617–621, *Akademiai Kiado, Budapest.*

[Gra93] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[Int93] Intel Corporation. *Paragon^{TM} OSF/1 USer's Guide*, February 1993.

[SM82] Stanley Y. W. Su and Krishna P. Mikkilineni. Parallel Algorithms and Their Implentation in MICRONET. In *Proc. of 8th VLDB Conf.*, pages 310–324, 1982.

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