Batch Scheduling in Parallel Database Systems

Manish Mehta
Valery Soloviev
David J. DeWitt

Technical Report #1147

April 1993

Batch Scheduling in Parallel Database Systems

Manish Mehta
Computer Science Department
University of Wisconsin-Madison

Valery Soloviev[1]
Computer Science Department
North Dakota State University

David J. DeWitt
Computer Science Department
University of Wisconsin-Madison

Abstract

Current techniques for query scheduling in a parallel database system schedule a single query at a time. This paper investigates scheduling of queries for parallel database systems by dividing the workload into batches. We propose scheduling algorithms which exploit the common operations within the queries in a batch. The performance of the proposed algorithms is studied using a simple analytical model and a detailed simulation model. We show that batch scheduling can provide significant savings compared to single query scheduling for a variety of system and workload parameters.

1. Introduction

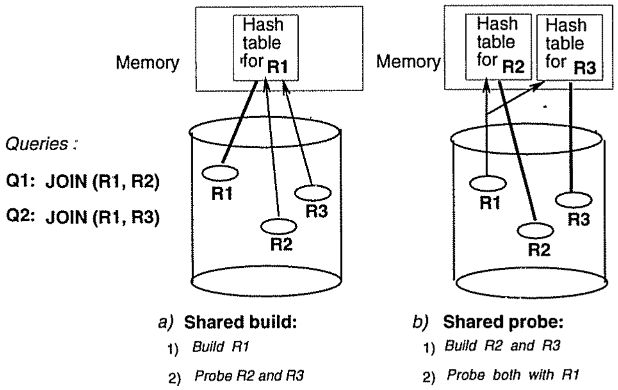
The improvements in database technology over the past decade have made database management systems (DBMSs) an essential component of many application domains. Current trends show that customer applications on these databases are growing in size and complexity with an ever increasing demand for performance. Multiprocessor database machines (GAMMA [DeWi90], Volcano [Grae89], Bubba [Bora90], XPRS [Ston88], DBC/1012 [Tera85], Non-Stop SQL [Tand88]), with their promise of high availability, scalability and performance, are increasingly the systems of choice for large applications. Current parallel database configurations already contain up to several hundred processing nodes and future systems are projected to be even bigger. Efficient resource utilization and workload scheduling is crucial for maximizing the throughput of such systems.

An important portion of the workload for database systems consists of large and highly resource-intensive read-only queries. Unfortunately, scheduling queries for parallel execution in a multi-user environment is still an open question. Research on query scheduling has so far concentrated mainly on efficient processing of single queries. As a result, scheduling single, stand-alone queries is reasonably well understood. However, single-query scheduling techniques are not always directly applicable in a multi-user environment. In particular, most single-query scheduling techniques attempt to minimize the response time of each query by exploiting intra-query parallelism, ignoring, in the process, potential inter-query parallelism. Naive execution of the optimized queries may penalize other concurrently executing queries at run-time and lead to a decrease in system throughput.

[1] This work was done while the author was a visiting scientist at the University of Wisconsin-Madison.

The ideal multiple-query scheduling method would optimize a set of queries together and try to minimize the total execution time for the entire set of queries. It would integrate optimization and scheduling to find plans that take into account issues regarding concurrent execution and run-time resource allocation. This approach, even if it were feasible, would be very complex and expensive. On the other hand, the current solution of single-query optimization followed by run-time scheduling does not always lead to efficient schedules. Compile-time estimates of query execution parameters, like available memory and number of processors, may not be valid at run-time. A simple observation that can be used to increase the effectiveness of compile-time optimization is that quite often the same operation is a part of several queries. If these operations can be shared between different concurrently executing queries, they will only have to be executed once. This means that even though each query is optimized individually, by scheduling the operations within the queries in an appropriate manner it may be possible to reduce the incremental load each additional query imposes on the system.

As an example, consider the two queries shown in Figure 1: query Q1 which joins R1 and R2, and Q2 which joins R2 and R3. Current scheduling algorithms would treat the queries independently and scan relation R1 twice. Instead, the scan of R1 can be identified as a common operation and can be shared by both the queries. Figures 1a and 1b show two possible methods of sharing the scans. R1 can be scanned once to produce a single hash table which can be probed twice, once by R2 and then R3. Alternatively, in-memory hash tables can be built for R2 and R3, and a single scan of R1 can be used to probe both hash tables simultaneously.



a) **Shared build:**
1) *Build R1*
2) *Probe R2 and R3*

b) **Shared probe:**
1) *Build R2 and R3*
2) *Probe both with R1*

Figure 1: Sharing Operators in Queries

Given a multiple query workload, there are several instances where the same operation is common to different queries. These operations are executed multiple times because queries are scheduled independently. The techniques that we have developed identify common operations in batches of queries, scheduling them in such a way that repetition is decreased. This identification is possible in cases where we can batch together a set of queries for execution. This can be done if the query workload is known in advance (e.g. several database customers run a fixed set of queries for generating reports everyday) or in a scenario where the system collects several queries together before executing them.

This paper presents several scheduling algorithms that can be used to exploit the sharing of operators for batches of queries. The performance of these algorithms is compared to the current solution where each query is scheduled individually. We demonstrate the effect of sharing under several different workloads, identify the factors that determine the savings and study their relative importance.

The remainder of this paper is organized as follows. Related work is reviewed in the following Section. An overview of the concept of sharing and cooperative scheduling is presented in Section 3 and a description of the different scheduling algorithms in Section 4. Section 5 describes the simulation model that was used to compare the performance of the algorithms. Section 6 presents the results of our performance study and Section 7 contains our conclusions and suggestions for future work.

2. Related Work

Single query scheduling has received increasing attention recently. Using the optimization techniques first proposed in System R [Seli79] as a basis, simple query optimization techniques have been extended to handle larger and more complex queries ([Kris86], [Ioan87], [Swam88], [Ioan90]). Scheduling single queries for parallel systems was studied in [Schn90], which compared the tradeoffs between various scheduling algorithms for complex join queries and demonstrated that the best scheduling algorithm depends on the structure of the query and the resources available for its execution. This issue was further explored in [Chen92a] which studied segmented right-deep scheduling. [Chen92b] studied processor allocation techniques for executing complex join queries.

While the issues involved in scheduling a single query are now fairly well understood, little is known about the applicability of these techniques in complex, multi-user environments. In particular, none of these earlier studies have determined the best way of scheduling multiple query workloads. The only related work that we are aware of is [Brow92], which examines the effect of different memory allocation schemes in a mixed workload of short transactions and multiple large queries. Even this is only a preliminary study and does not provide answers to important policy questions.

The concurrent execution of multiple queries has also been proposed in multi-query optimization techniques. Several papers observed that database queries frequently share common sub-queries which process the same data ([Chak86, Fink82, Hall74, Sell88]). However, these optimization techniques identified sharing only at the query language level, restricting the degree of sharing that can be exploited. Instead, we consider sharing at a lower level, namely at the level of individual query operators. This allows, for example, two selections with non-overlapping predicates, to be processed using a single relation scan. The previous approaches would not have found anything in common between the two selections. Operator sharing was also proposed in [Gran80] in the context of optimization in deductive databases. A significant difference between our work and previous proposals is that compile-time optimization techniques tend to ignore issues of resource allocation. As will be demonstrated later, changing the amount of memory available can have a significant impact on the performance of the proposed sharing algorithms.

3. Sharing Operators across Multiple Queries

In this section we describe how sharing can be implemented for the basic database operators. While the discussion concentrates on handling selections and joins, techniques for sharing other basic operators are also described briefly.

3.1. Sharing Selects

We assume that all relations are accessed via a select operator and that selection operators on the same relation can be shared. Each shared select operator has multiple predicates, one for each of the original selects, and a corresponding list of operators to which qualifying tuples are to be sent. The operator reads tuples from the input relation, applying each of the predicates in turn. For each predicate that a tuple satisfies, a copy of the tuple is sent to the "next" operator in the corresponding query. The advantage of selection sharing is that the relation is scanned only once, leading to significant I/O savings if the relation is disk-resident.

3.2. Sharing Joins

The join operator is one of the most complex database operators to execute. While a number of different join algorithms have been proposed, we restrict out attention to the hybrid-hash join algorithm [DeWi84, DeWi85]. This algorithm executes in two phases, termed the **build** and **probe** phases. In the build phase, the smaller relation is scanned and an in-memory hash table is constructed by hashing each tuple on the join attribute. In the second phase, the outer relation is scanned and tuples are used to probe the hash table to test for matches. The matched tuples are then joined and produced as output.

If several joins in a batch of queries involve the same relation, the hash table for this relation can be shared between the different joins. This hash table can then be probed multiple times using tuples from each of the different probing relations. Sharing the build phase in this manner has two important effects. First, the total execution time is reduced because the building relation is scanned only once. Second, there is a decrease in the amount of memory used since only one copy of the data is kept in memory and it is shared by all the joins. Since hash-joins are very memory intensive algorithms, reducing memory usage for one set of queries can significantly improve performance by making additional memory available to other queries.

In the case of a multiprocessor join algorithm [Kits83, Vald84, DeWi85, Schn90], the building and probing relations are horizontally partitioned across multiple processors by hashing on the join attribute. At each processor

Q1 : A[a1, a2] $\bowtie$ B[a2, a3]
Q2 : B[a2, a3] $\bowtie$ C[a3, a4]

Fig. 2a: Join Queries

a2	a3
1 -- 10	a -- m
11 -- 20	n -- z

Fig. 2b: Relation B – ranges of partition attributes a2 and a3

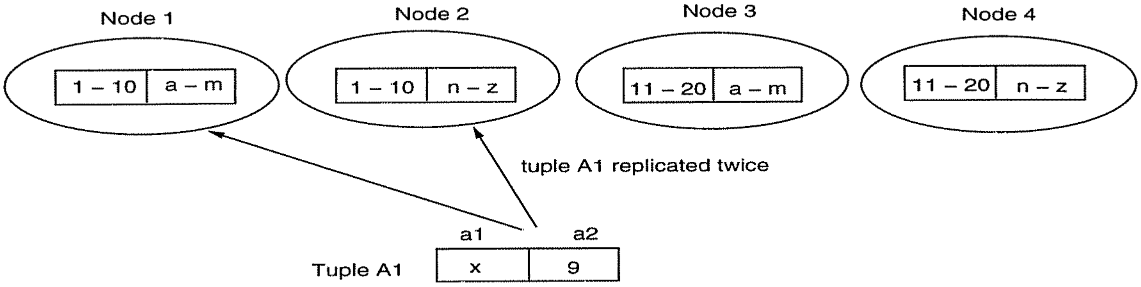


Fig. 2c: Probing with tuple A1 of relation A for query Q1; relation B is in memory

a local join of the fragments of the relations can then be performed in parallel. If two or more joins with a common build relation use different join attributes, the partitioning hash function has to be applied multiple times for each probing tuples; once for each join attribute. This will generally cause the replication of tuples of the probing relation. For example, consider the two queries shown in Figure 2a. The first query joins relations A and B on attribute a2 and the second query joins relations B and C on attribute a3. The partitioning ranges for the two join attributes a2 and a3 are shown in Figure 2b. In addition, assume that the relations are to be partitioned across four nodes. Figure 2c shows one possible way of partitioning relation B and how relation A tuples need to be replicated when they are partitioned. Similarly, relation C tuples are also replicated while partitioning. The degree of replication depends on the number of processors and the number of queries sharing the build relation. Our goal is to employ a hash function that minimizes this replication. The solution models the hashing as an integer programming problem which has been studied in the context of data placement for parallel systems in [Yack92]. The results show that the replication factor can be approximated by $P^{k-1/k}$ where P is the number of processors and k is the actual number of attributes used for hashing[2].

For joins involving the same relation but with join predicates on different attributes, a separate hash table must be constructed for each join. While at first glance one would expect this to lead doubling the join's memory requirement this is actually not the case. Typically, most of the memory consumed is used to hold the building tuples while the actual hash table and the various hash chains occupy a much smaller percentage of the total memory consumed. Replicating the hash table and these hash chains consumes much less memory than what would be required to hold a separate copy of the relation for each join.

3.3. Other Operators

While we have described the application of sharing to only select and join operations, sharing can be used for other operators as well. Consider the sort operator as an example. If two or more sort operators share the same sort attribute, the tuples can be sorted once and the sorted stream can then be used multiple times. This will not only reduce the number of I/Os performed but also the number of CPU cycles consumed. Other examples of operators

[2] We assume that the usage of attributes in join predicates follows a Zipfian distribution [Zipf49] with z = 1.5. This is a skewed distribution where the most probable attribute is likely to be chosen 50% of the time and the other half is distributed among the rest of the elements. The skewed distribution was chosen as it seems very likely that some attributes are used more often as join attributes than others (employee number vs employee address).

which can exploit sharing are aggregates, group-by operators, and operators for duplicate elimination.

4. Scheduling Algorithms

We assume that the workload for the database system consists of batches of queries and that each query is composed of several operators. In addition, there is a partial order defined on the operators in a query. For example, the probe phase of a hash join operator cannot begin until the build phase has completed. There are several ways in which the operators from different queries can be combined into a single schedule. The aim of our scheduling algorithms is to find the global schedule for all queries that minimizes the total execution time for a batch of queries without violating the partial order constraints.

4.1. Workload Description

Before describing the scheduling algorithms, we will first describe our query workload since it determines the amount of sharing possible and consequently the design of the algorithms. Our simplified workload consists purely of single hash join queries. This workload was chosen because it allows us to study the effects of sharing without considering other multi-query scheduling issues. Single join queries can be scheduled in only two ways — the only choice being which of the two relations is used as the building relation. In contrast, for a more complex multi-join query, there are several execution strategies including left-deep, right-deep, and bushy [Schn90]. As the relative performance of these alternatives is not known for a multi-query environment, including multi-join queries in our workload would not have allowed the separation of the effects of sharing from other query scheduling issues. Another simplification is that all select operators have a selectivity of 100% and all the join operators have a join selectivity of 50%. We feel that varying the selectivities would have also unnecessarily complicated the study.

4.2. Algorithms

All the algorithms operate on a query graph defined on the queries in a batch. The nodes of this graph are the relations accessed by at least one query. There exists an edge from node R_i to node R_j for every query Q that references relations i and j. A batch of queries defines a graph which is a collection of disjoint connected subgraphs. Each subgraph represents a subset of the queries in the batch which share some relation with other queries in the same subgraph. Consequently, sharing of operators is possible only within a connected subgraph. Figure 3 shows an example of a query graph for a batch of 7 queries divided into 3 subgraphs.

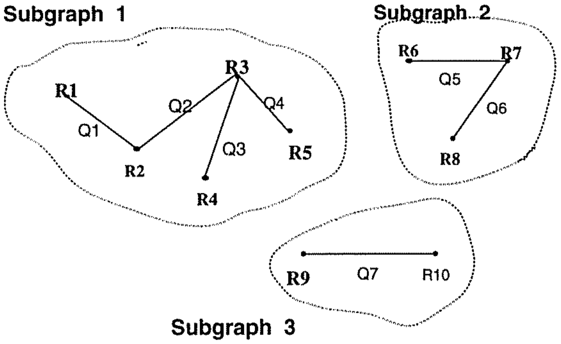


Figure 3: Sample Query Graph

4.2.1. Cost Metric

The scheduling algorithms need to compare different schedules in order to determine the "best" schedule. While several choices of metrics are possible, we elected to use the total number of I/Os performed by the queries in a batch. This metric was chosen for its simplicity and because it is a good indicator of the degree of sharing — the higher the degree of sharing the lower the number of I/Os performed. The analytical results obtained using this cost metric are presented in Section 6.

4.2.2. Non-Sharing Algorithm

The first scheduling algorithm is the algorithm which does not attempt to exploit sharing. In this algorithm, each query is scheduled independently. The scheduler admits as many queries as can run given the global memory of the system. The remaining queries simply wait in a queue and are admitted one at a time as sufficient memory becomes available. This algorithm is the base case with which other sharing algorithms are compared.

4.2.3. Exhaustive Algorithm

The first sharing algorithm developed was an exhaustive algorithm. The algorithm needs to makes three decisions in forming a global schedule:

(1) Select a global ordering of the queries — the complexity of this step is $n!$ where n is the number of queries in the connected subgraph.

(2) For each query, determine the role of each relation. That is, choose the building and the probing relations. This requires examining 2^n choices where n is the number of queries in the subgraph.

(3) Determine the order in which relations are flushed from memory as only a subset of the relations can be held in memory at any point in time. In order to allocate memory for the next relation, one or more of the relations in memory will have to be flushed. The subset of relations to be flushed can be chosen in many ways. This step requires consideration of $m!$ steps where m is the number of relations present in memory.

The exhaustive algorithm was designed to consider all possible alternatives for each of these three factors. The algorithm schedules one subgraph at a time and chooses the best schedule for each subgraph. Since there is no sharing across subgraphs, this also provides the best global schedule. Given a query ordering and role assignment, building relations are read into memory until the next relation can not fit. Then the corresponding probe operations are performed. In order to free up space for the next building relation, all possible orders of flushes of the relations currently in memory are examined. The algorithm evaluates the cost of each of the global orderings and then chooses the schedule with the minimum cost. The overall complexity of the algorithm is $n!*2^n*m!$.

The high complexity of exhaustive enumeration makes it infeasible as a practical scheduling algorithm (it takes a few hours on a 20 MIPS processor to enumerate all choices for a single subgraph of more than 7 nodes). However, the algorithm was still very useful to study since it provided a maximum bound on the savings that can be achieved through sharing. Also, it showed the structure of the search space of global schedules and gave a good indication of the relative importance of the various factors which determine savings.

4.2.4. Heuristic Algorithms

The high cost of the exhaustive algorithm made us look for heuristic algorithms in order to be able to process larger subgraphs. The heuristic algorithms developed to handle larger subgraphs are described next. These algorithms require making two important decisions - selecting the next building relation and the next relation to flush from memory.

4.2.4.1. Selecting Next Building Relation

All the relations are initially ranked according to a ranking function. We studied three ranking functions in this paper:

(1) ProbeSize

This function ranks each relation by summing the sizes of all relations that can potentially be used as probing relations but which have not yet been allocated in memory. Once a relation has been built in memory, all the associated probes can be performed. The result is to give a higher rank to relations that can be shared more often. The relations already in memory are ignored because it is more efficient to use the relation under consideration as the probing relation for other in-memory relations; the build and the probes for these relations can be shared and executed concurrently.

(2) ProbeSubBuild

The previous heuristic uses only the sizes of the probe relations and ignores the usage of memory by the build relation. The second ranking function subtracts the size of the build relation from the sum of all the potential probing relations found by the previous function. Subtracting the size of the build relation biases the ranking towards smaller relations which use less memory.

(3) ProbeDivBuild

The last ranking heuristic divides the sum of all potential probing relations by the size of the building relation. Similar normalization by size has also been found useful in work in areas like file migration, object caching, knapsack problems, etc.

The ranking functions are used to rank all the relations not yet processed. The highest ranked relations are read into memory until no more relations fit. Then all the corresponding probing relations are scanned and the joins are effected. Note that this step also decides the roles of the relations in the query as the relation with the higher rank is chosen as the build relation.

4.2.4.2. Selecting Next Relation to Flush

Before the next query can be initiated, memory space must be reclaimed from one of the relations currently residing in memory. Three alternatives were considered:

(1) Largest

The first heuristic orders the relations in memory by decreasing size and selects the largest one first. Doing so maximizes the space for subsequent building relations which, in turn, increases the possibility of sharing.

(2) LowRank

This heuristics uses the rankings produced by the build heuristic and selects the lowest ranking relation; that is, the one which is shared less. Flushing these relations first tends to keep highly shared relations in memory longer, increasing the possibility of sharing.

(3) HighRank

The third flush heuristic selects the relation with the highest rank.

The algorithms use the building and flushing heuristics repeatedly until all the queries in the batch have been processed. All the heuristics studied in this paper are listed in Table 1.

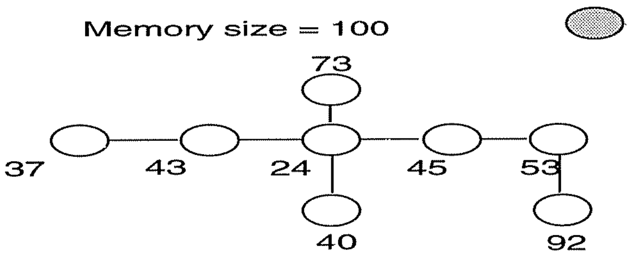
Build Heuristics		**Flush Heuristics**	
ProbeSize	Σ Sizes of probe Relations	Largest	Largest In-Memory Relation
ProbeBuildSize	ΣSizes of probe Relations - Rel Size	LowRank	Lowest Ranking In-Memory Relation
ProbeDivBuild	ΣSizes of probe Relations / Rel Size	HighRank	Highest Ranking In-Memory Relation

Table 1: Heuristics for Building and Flushing Relations

The execution of one of the heuristic algorithms that uses the ProbeSize and Largest heuristics is illustrated for a batch of seven queries in Figure 4. For simplicity, assume that the number of the relation also indicates its size in pages. Figure 4a shows the initial state of the system and the ranks for all the relations. The heuristic first builds the relation with the highest ranking (R24) (Figure 4b). The ranks are reevaluated and R53 is built next.(Figure. 4c). As no more relations can fit in memory, the largest in-memory relation (R53) is then probed with relations 92 and 45 (relation 45 also probes relation 24 simultaneously) (Figure 4d). Next, relation 53 is flushed as all the corresponding probes have completed. Then, relation 37 is built and finally all the remaining probes are executed. (Figures 4e-f)

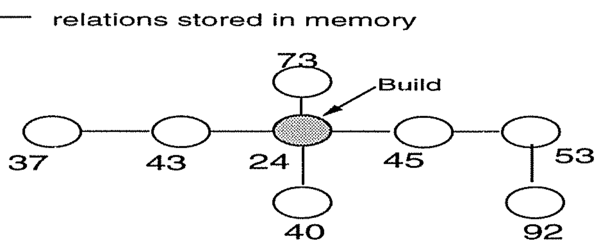
5. Simulation Model

The simulator developed for this work is based upon an earlier, event-driven simulation model of the Gamma parallel database machine. The earlier model was a useful starting point since it had been validated against the actual Gamma implementation; it had also been used extensively in previous work on parallel database machines [Ghan90, Schn90, Hsia91]. The new simulator, which is much more modular, is written in the CSIM/C++ process-oriented simulation language [Schw90]. The simulator accurately captures the algorithms and techniques used in Gamma, and it is currently being used as an experimental vehicle in several ongoing research studies. The remainder of this section provides a more detailed description of the current simulation model.



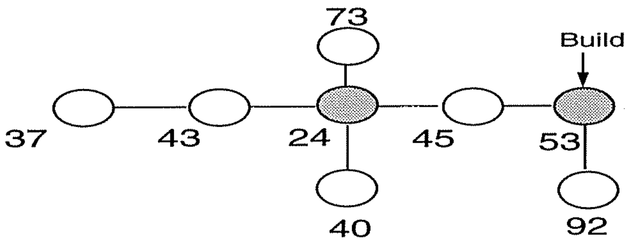
Relation	Weight	
37	43	
43	61	(37 + 24)
24	201	(43 + 73 + 45 + 40)
73	24	
40	24	
45	77	(24 + 53)
53	137	(45 + 92)
92	53	

Figure 4a: Step 1



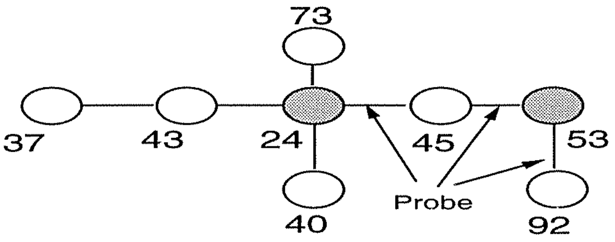
Relation	Weight
37	43
43	37
73	0
40	0
45	53
53	137
92	0

Figure 4b: Step 2



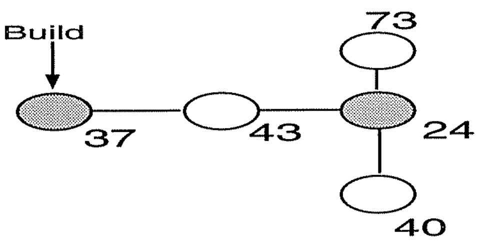
Relation	Weight
37	43
43	37
73	0
40	0
45	0
92	0

Figure 4c: Step 3



Relation	Weight
37	43
43	37
73	0
40	0

Figure 4d: Step 4



Relation	Weight
43	37
73	0
40	0

Figure 4e: Step 5

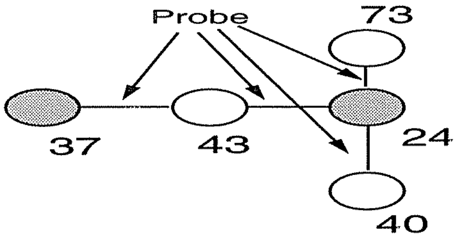


Figure 4f: Step 6

5.1. Simulator Overview

The simulator models a *shared-nothing* [Ston86] parallel database system. The main components of such a system are shown in Figure 5. The system consists of a number of *Processing Nodes* connected in a shared-nothing manner via an *Interconnection Network.* Each processing node consists of one or more CPUs with memory and one or more disk drives. Transactions, which are submitted from a set of external *Terminals*, are first routed to a special *Scheduler* node. This node controls the scheduling and execution of all transactions present in the system. The *Database* is modeled as a set of relations and indices. The relations (and their associated indices) are declustered [Ries78, Livn87] over all the nodes. The simulator models the parallel database system as a closed queueing system, with the workload originating from a fixed set of terminals. Given this background, we can now turn our attention to the details of the various components of the simulator.

5.2. Terminals

The terminals model the external workload source for the system. Each terminal sequentially submits a stream of queries of a particular class. After each query is formulated, the terminal sends it to the Scheduler node for execution and then waits for a response before continuing on to the next query.

5.3. Processing Nodes

Each processing node in the system is modeled as a single CPU, disks, and a buffer pool. The CPU uses a preemptive resume scheduling algorithm based on priorities. The processes are divided into two classes - processes

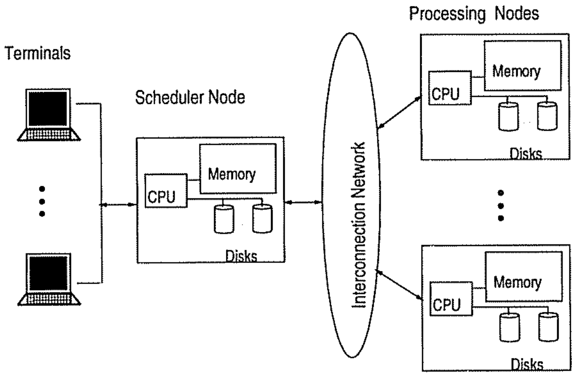


Figure 5: Shared-Nothing Parallel DBMS Architecture.

that perform I/O are in the high priority class and the rest of the processes are in the low priority class. The buffer pool models a set of main memory page frames. Page replacement in the buffer pool is controlled via the LRU policy extended with "love/hate" hints (like those used in the Starburst buffer manager [Haas90]). These hints are provided by the various relational operators when fixed pages are unpinned. For example, "love" hints are given by the index scan operator to keep index pages in memory; "hate" hints are used by the sequential scan operator to prevent buffer pool flooding. In addition, a memory reservation system under the control of the Scheduler node allows memory to be reserved in the buffer pool for a particular operator. This memory reservation mechanism is used by joins to ensure that enough memory is available to prevent their hash table frames from being stolen by other operators running on the same node.

The simulated disk models a Fujitsu Model M2266 (1 GB, 5.25") disk drive. This disk provides a 256 KB cache which is divided into eight 32 KB cache contexts for use in prefetching pages for sequential scans. In the disk model, which slightly simplifies the actual operation of the disk, the cache is managed as follows: each I/O request, along with the required page number, specifies whether or not prefetching is desired. If so, one context's worth of disk blocks (5 blocks) are read into a cache context after the originally requested data page has been transferred from the disk to memory. Subsequent requests to one of the prefetched blocks can then be satisfied without incurring an I/O operation. A simple round-robin replacement policy is used to allocate cache contexts if the number of concurrent prefetch requests exceeds the number of available cache contexts. The disk queue is managed using an elevator algorithm.

Configuration/Node Parameter	**Value**	**CPU Cost Parameter**	**No. Instructions**
Number of Nodes	16 nodes	*Initiate Select*	20000
Number of Disks	1 disk	*Initiate Join*	40000
CPU Speed	30 MIPS	*Terminate Join*	10000
Memory	10 Mb (varied)	*Terminate Select*	5000
Page Size	8 KB	*Apply a Predicate*	100
Disk Seek Factor	0.617	*Read Tuple*	300
Disk Rotation Time	16.667 msec	*Write Tuple into Output Buffer*	100
Disk Settle Time	2.0 msec	*Probe Hash Table*	200
Disk Transfer Rate	3.09 MB/sec	*Insert Tuple in Hash Table*	100
Disk Cache Context Size	4 pages	*Hash Tuple using Split Table*	500
Disk Cache Size	8 contexts	*Copy a Byte in Memory*	1
Disk Cylinder Size	83 pages	*Copy 8K Message to Memory*	10000
Wire Delay for an 8K Message	5.6 msec	*Send(Receive) an 8K Message*	1000
Tuple Size	400 bytes	*Start an I/O*	1000

Table 2: Simulator Parameters and Values.

The key parameters of the processing nodes, along with other configuration parameters, are listed in Table 2. The configuration parameters were chosen to represent a typical shared-nothing parallel database system. The software parameters are based on instruction counts taken from the Gamma prototype when the previous simulator was validated. The disk characteristics approximate those of the Fujitsu Model M2266 disk drive described earlier.

5.4. Interconnection Network

The simulator assumes an interconnection network of infinite capacity. The decision to model such a network was made as a result of our experience with the Intel iPSC/2 Hypercube, on which Gamma runs, where network bandwidth has never been an issue. Given that the next generation of parallel processors (like the CM-5 and Intel Paragon) provide even higher capacity networks, we felt that this was a reasonable assumption. Communication packets do, however, incur a "wire" delay corresponding to the delay encountered on the iPSC/2, and the CPU costs for sending and receiving messages are included in the model.

5.5. Scheduler

The Scheduler is a special processing node that accepts queries from terminals and schedules their execution on the appropriate processing nodes. The scheduling algorithms preprocess the set of input queries, annotating the queries to reflect the global scheduling order. The scheduler accepts the annotated queries and executes the operators accordingly, maintaining the global order determined by the scheduling algorithms. The scheduler starts the operators at the appropriate nodes of the system and also coordinates their execution.

5.6. Operators

All the queries in the work reported here use only two basic relational operators: select and join. Any result tuples from the queries are "sent back to the terminals", an operation that consumes a small amount of processor cycles for network protocol overheads. Each select process also spawns an additional scan process which just reads pages from the disk and passes them on to the select process. The select process then applies the filtering predicate(s) and redistributes the qualifying tuples if necessary.

6. Experiments and Performance Results

In this section results from the analytical and simulation models are presented. We also include a comparison of the performance of the two models. But first we briefly describe the experimental methodology used for the experiments.

6.1. Methodology

The important configuration, database and workload parameters used in the performance section are shown in Table 3. The relation sizes are chosen such that on an average two or more relations can fit in memory and the sizes are distributed linearly. The workload consists of batches of two-way join queries and all queries use the same selectivities. The query batches are generated by partitioning a stream of random queries into batches of different sizes. The relations used in a particular query are chosen uniformly from the relations in the database for most experiments though we also study the effect of skewed relation usage.

6.2. Analytic Comparison

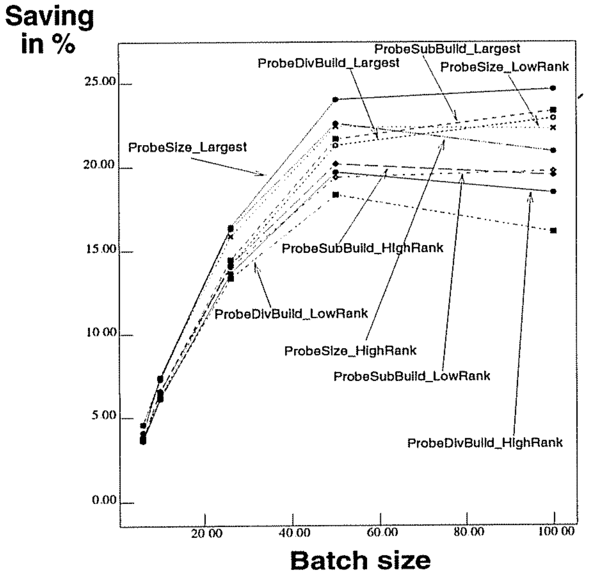
We used our simple cost metric to perform two analytical experiments. The first experiment compares the performance of the various heuristics and the second compares the heuristic algorithm to the optimal one.

6.2.1. Comparison of Different Heuristics

The first analytical experiment compares the performance of the three heuristics for building relations and the three heuristics for flushing relations from memory (Table 1). Combining these heuristics in all possible ways, gives a total of nine algorithms to study. Figure 6.1 shows the performance of the six different heuristic algorithms. The lines have been labelled by the particular combination used; for instance, ProbeSize_Largest means that heuristic ProbeSize is used to build relations and heuristic Largest is used to flush relations. The graph shows the savings obtained by each of the heuristic algorithms for a database of 100 database relations as a function of the batch size (the number of queries the scheduler collects at a time prior to analysis) relative to the standard non-sharing scheduler.

Configuration		**Database**		**Workload**	
#Nodes	16	#Relations	50-200	Batchsize	6-200
CPU	30 MIPS	Sizes	15 - 150 Mb	Selectivity (Join)	0.5
#Disks	1			Selectivity (Select)	1.0
Memory	10Mb - 40Mb			Relation Usage	Uniform/Skewed

Table 3: Experimental Parameters



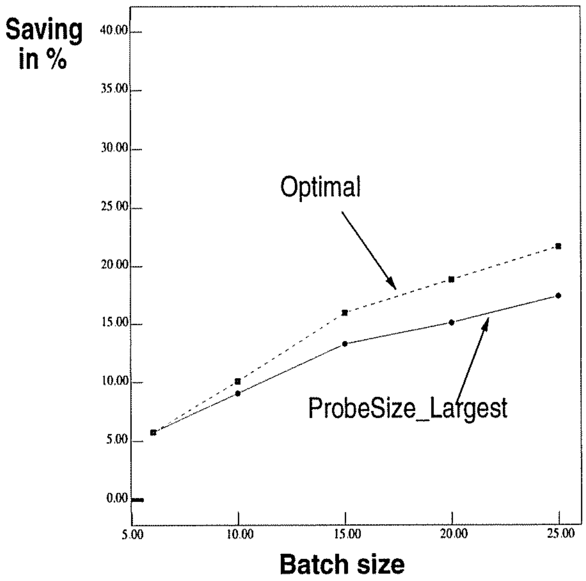


Figure 6.1: Comparing Heuristic Algorithms

Figure 6.2: Comparison with Optimal

Analytical Results

The first thing to note is that for all algorithms the savings increase as a function of the batch size. For small batch sizes, all the heuristic algorithms perform nearly the same as there is only limited sharing in such cases and all the algorithms exploit it successfully. The algorithms show significant differences only for larger batch sizes where there is a significant amount of sharing. All algorithms using the *ProbeSize* heuristic to build relations outperform algorithms that use *ProbeSubBuild* as a heuristic. This shows that trying to incorporate the size of the building relation into the build heuristic is not very helpful. Also, the *Large* heuristic to flush relations performs better overall than other heuristics for flushing. The only exception is with a batch size of 50 where *Lowrank* and *Highrank* provide slightly better performance than the *ProbeBuildSize_Largest* heuristic. This shows that freeing up memory as soon as possible helps in exploiting sharing more than using the ranking function to determine which relation to flush next. As is clear from figure 6.1, the *ProbeSize_Largest* algorithm is the best for all batch sizes. For the remainder of this paper, we only consider this heuristic.

6.2.2. Comparison with the Optimal Algorithm

In the next experiment, the ProbeSize_Largest algorithm is compared to the optimal algorithm to determine the quality of the results obtained by the heuristic algorithm. As mentioned before, the optimal algorithm could not be executed for subgraphs with more than 7 nodes. Thus, the two algorithms were compared only for cases where all subgraphs had less than 7 nodes. This meant that the algorithms could be compared only for small batch sizes (less than 30) because for larger batch sizes nearly all batches contain at least one subgraph larger than 7 nodes. The performance of the ProbeSize_Largest heuristic and the Optimal algorithm is shown in Figure 6.2. The performance of the two algorithms is very close for small batch sizes but the difference between them increases as a function of the batch size. The maximum difference is about 4% for a batch size of 25 queries. The optimal algorithm does better compared to the heuristic algorithm because it considers several orders of magnitude more schedules compared to the heuristic algorithm. Still, the results are quite encouraging as significant savings were obtained even with a very simplistic heuristic algorithm. While one could look for better heuristics, we decided instead to study the performance of the *ProbeSize_Largest* algorithm using the more realistic simulation model.

6.3. Simulation Comparison

6.3.1. Ratio of Batch Size to Number of Relations

The first simulation experiment studies the effect of varying the ratio of the number of queries in a batch and the number of relations in the database. The higher the ratio, the higher the probability that a relation is accessed by multiple queries, which leads to higher sharing within a batch. Figure 7.1 shows the results of varying the ratio from 0.06 to 1.0 for three databases consisting of 50, 100 and 200 relations, respectively.

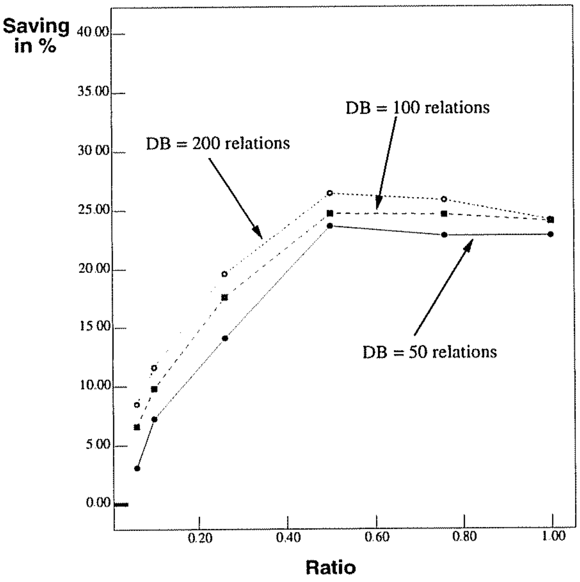


Figure 7.1: Varying Ratio of Number of Queries to Database Size

For all database sizes, the savings increases as a function of the batch size. The improvement is small for small batch sizes; only 4% for a batch of 6 queries and a database of 50 relations. This is because there is only a limited potential for sharing in small batches. The savings increase sharply for batchsizes of 26 and 50 as the sharing increases further. The increase in improvement stops after the ratio exceeds 50% with the savings actually declining slightly for larger databases. This occurs for two reasons. First, to effectively exploit the additional sharing present in larger batch sizes requires more memory; since the memory size is constant for all batchsizes in this experiment, sharing is not exploited as well for larger batchsizes. This effect is also observed in the analytical model where the increase in savings decreases with larger batchsizes (fig. 6.1). Second, as the amount of sharing increases, the replication of tuples used to probe the shared hash table also increases (see Section 3.2). This increases the CPU contention at the nodes and leads to an overall increase in the execution time of the sharing algorithms. This factor further decreases the improvement for larger batch sizes.

These results demonstrate that batch scheduling can significantly improve system throughput. The improvement is nearly 20% even for medium sized batches of 25 queries and it becomes even higher with larger batch sizes. Also, it is interesting to note that the curves for all three database sizes are quite close to each other. This shows that as long as the ratio of batch size to the number of relations in the database remains constant, similar savings can be expected.

6.3.2. Sensitivity to System Memory

The next experiment examines the effect of the amount of available system memory on the different algorithms. A higher aggregate memory implies that more relations can be built into memory concurrently which leads to higher sharing. Figure 7.2 shows the effect of increasing the aggregate memory from 160Mb to 640 Mb for different batch sizes[3].

Consider first the curve for a batch size of 10. Smaller batches have only limited sharing which is exploited even for smaller memory sizes. Increasing memory does not increase sharing but it does help the naive algorithm

[3] Each disk in our initial system configuration had multiple (8) cache contexts. This implies that up to 8 scans can be scheduled in parallel and still a significant number of disk cache hits would be observed. At aggregate memory sizes of 320 Mb and higher, more hash tables can be built in memory. Consequently, there are cases where more than 8 scans are scheduled concurrently. This causes the cache hits to drop considerably and the overall execution time to become worse. The sharing algorithms still perform better than the naive algorithm because they do less I/Os but the experiments are not realistic because of the excessive parallelism which degrades throughput. In order to remove this effect of high parallelism the number of cache contexts was increased to 24 for this experiment.

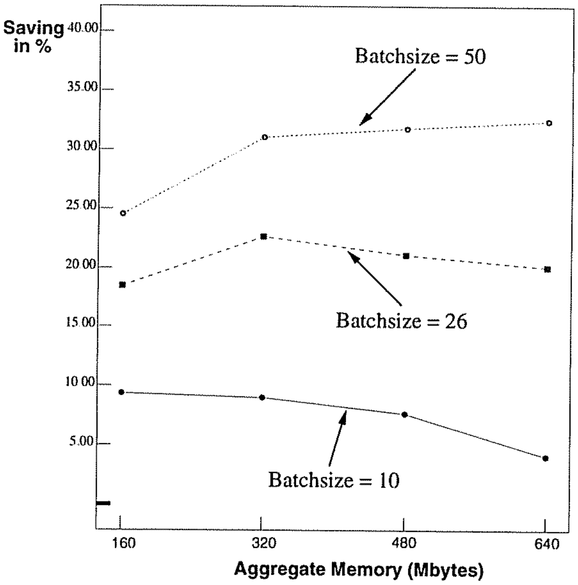


Figure 7.2: Varying Aggregate Memory

which is able to execute more queries in parallel and reduce it's overall execution time. This causes the relative improvement provided by the sharing algorithm to decrease.

Larger batch sizes offer more opportunities for sharing which cannot be fully exploited with small memories. Thus, increasing the amount of memory available enables higher sharing resulting in a corresponding increase in the savings; for a batch size of 26 the savings increases from 18.5 to 22.6 percent as the amount of memory is doubled from 160 to 320 Mb. As memory is increased further the savings again decrease because the naive algorithm also gains due to higher concurrency. We do not see any decrease in the savings for the largest batch size of 50 even for the largest memory sizes. This is because even though the naive algorithm gains due to parallelism there is enough sharing possible to offset this increase.

The experiment demonstrates that, except for very small batch sizes, sharing algorithms can benefit from an increase in the amount of memory available. However, beyond a certain point, increasing the amount of memory available also benefits the naive algorithm leading to a decrease in the improvement due to sharing.

6.3.3. Non-Uniform Relation Usage

So far the relations used by the queries have been chosen uniformly from the set of all relations. In reality, this assumption may not always hold. It is very likely that some relations will be accessed more frequently than others. Skewed relation accesses lead to a greater degree of sharing because the frequently-accessed relations are shared among more queries. This experiment explores the effect of skew in relation usage.

In the absence of real-life data concerning the usage of relations in query workload, a simplistic usage distribution is assumed; 10% of the relations are used 50% of the time and the other relations are chosen uniformly from the remaining 90% of the relations. Figure 7.3 compares the skewed and uniform distributions for a database of 100 relations with varying batch sizes. As with the uniform case, the savings in the skewed case also increase with an increase in the batch size. In addition, the savings are much higher in the skewed case. This is because the sharing algorithms can exploit sharing further by keeping the frequently accessed relations in memory. The increase in improvements with larger batch sizes stops at smaller batchsizes with the skewed distribution; the skewed case does not show any improvement after batch size 26 while the uniform case keeps improving till batch size 50. After a certain point the lack of additional memory prevents sharing and, as the skewed distribution has a higher potential for sharing, it hits the memory boundary earlier. If more memory was available, the skewed distribution would have shown an increase in savings even for larger batch sizes. These results demonstrate that if the usage of the relations in queries is non-uniform the improvements to be gained by exploiting sharing are even more pronounced than in the uniform case.

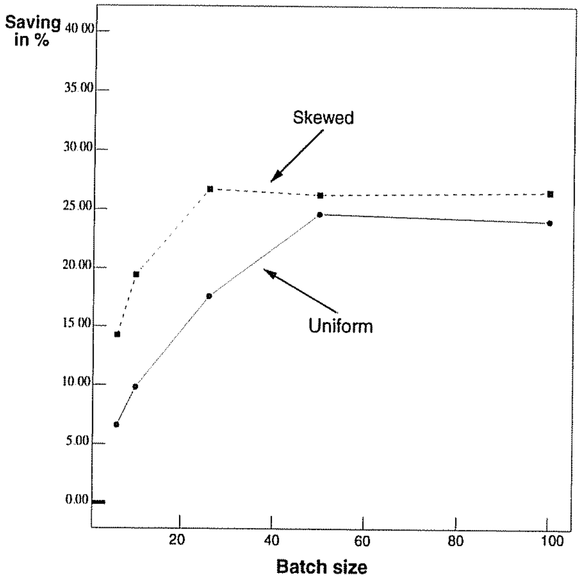


Figure 7.3: Non-uniform Relation Usage

6.3.4. Comparison of Analytical and Simulation Model

The cost metric used by the analytical model (total number of I/Os) ignores parallelism: it assumes all scans are done sequentially while parallel scans are executed in the simulation model. In this subsection, we compare the two models and explore how ignoring parallelism affects the validity of the metric. Figure 8.1 shows the results of the analytical and the simulation model for a database of 100 relations for various batch sizes. The analytical and simulation results are also shown for three different batch sizes with varying memory in figure 8.2. The analytical model matches very closely with the simulation results in figure 8.1 and are also quite similar for low memory sizes in figure 8.2. But the two models show differing trends as memory is increased further. The analytical model predicts an increase in savings with higher memory while the simulation model shows that there is no improvement in savings and for smaller batches the savings actually decrease with higher memory. This is because on increasing memory the naive algorithm improves as memory is available to run more queries. The analytical model which ignores parallelism cannot account for this effect. The level of concurrency is low in Figure 8.1 and for lower memory sizes in figure 8.2 and thus the two models show similar results.

Figures 8.1 and 8.2 show that even though the analytical model ignores parallelism, the results it produces are generally fairly close to those produced by the simulation model. The two models start differing significantly when

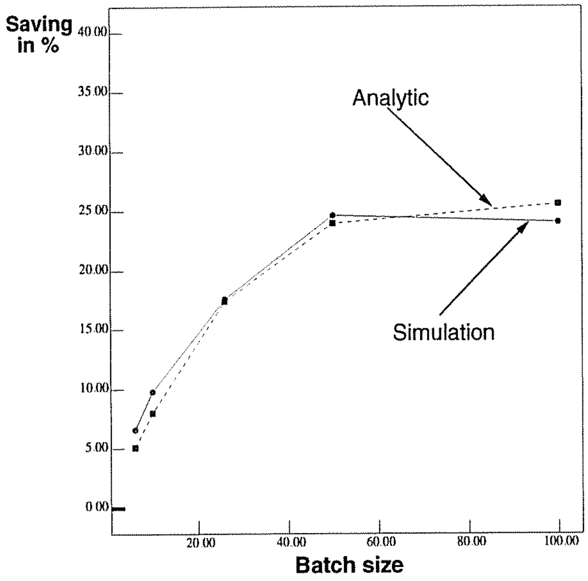


Figure 8.1: Varying batch sizes

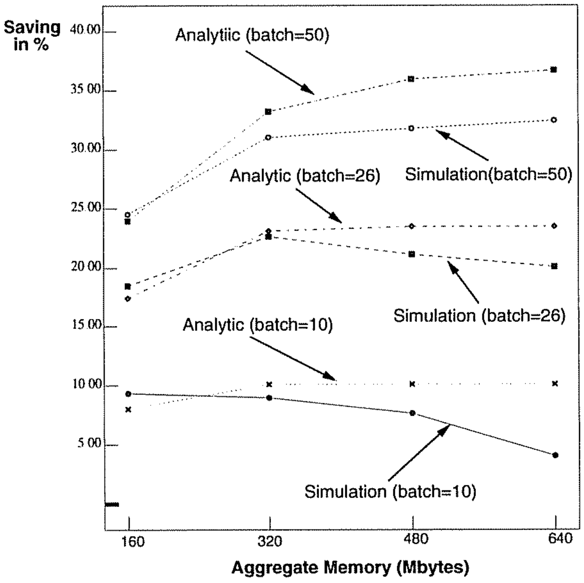


Figure 8.2: Varying Memory

Comparing Analytical and Simulation Models

a large portion of the queries in a batch can be executed concurrently even with the naive algorithm, in which case the analytical model predicts higher savings compared to the simulation model.

7. Conclusions and Future Work

In this paper, batch scheduling of the query workload was studied in the context of a parallel database system. We developed batch scheduling algorithms that exploit the sharing of low-level query operators. These algorithms are quite simplistic and can easily be used even at run-time to schedule batches of queries. The performance of these algorithms was studied for a simplified workload. We demonstrated that sharing can provide significant savings for a variety of database and memory sizes. We also showed that the savings obtained are even larger if some database relations are queried more frequently. The analytical and the simulation models were also compared and the limitations of the simplistic cost metric were explored.

This paper made several assumptions that may not always hold. We have assumed that we can save I/Os by sharing any two select operations. This may not be true with an access mechanism where the filtering predicates can be used to direct the search for relevant tuples. For example, if an index is employed, only the relevant data tuples need to be read. Also, in our performance study, accesses to the relations were made on all nodes of the system. In several cases, if a declustering strategy other than round-robin is used (multi-attribute partitioning [Ghan90], we can restrict access to only a subset of the nodes in the system; if the relations are *range partitioned*, two selects may be scheduled on disjoint nodes in which case sharing is not possible. However, even the presence of such access mechanisms does not completely remove the benefits of sharing. For instance, consider sharing in the presence of indices. Assume that there are several selection operators on a particular relation. Even if one of the operators is a file scan, all the selections can be combined and shared. Additional work is needed to investigate the performance of sharing in the presence of such access mechanisms.

We mentioned earlier that sharing can be used in cases where several queries are collected together before execution to exploit sharing. We want to explore the performance tradeoffs between making queries wait for execution versus scheduling and executing the queries as soon as they arrive.

Acknowledgements

This research was supported by the IBM Corporation through a Research Initiation Grant.

REFERENCES

[Bobe92a] Bober, P., and Carey, M., "On Mixing Queries and Transactions via Multiversion Locking," *Proc. 8th IEEE Data Engineering Conf.*, Phoenix, AZ, Feb. 1992.

[Bobe92b] Bober, P., and Carey, M., "Multiversion Query Locking," *Proc. 18th Int'l. VLDB Conf.*, Vancouver, BC, Canada, Aug. 1992, to appear.

[Bora90] Boral, H. et al, "Prototyping Bubba: A Highly Parallel Database System," *IEEE Trans. on Knowledge and Data Engineering* 2(1), March 1990.

[Brow92] Brown, K., et al, "Resource Allocation and Scheduling Issues for Mixed Database Workloads", *Comp. Sc. Tech. Rep. TR 1095*, University of Wisconsin-Madison, July, 1992.

[Chak86] Chakravarthy, U. S. et al, "Semantic Query Optimization in Expert Systems and Database Systems", *Expert Database Systems: Proc. of 1st International Workshop*, Menlo Park, Calif. 1986.

[Chen92a] Chen, Ming-Syan et al, "Using Segmented Right-Deep Trees for the Execution of Pipelined Hash Joins", *Proc. 18th VLDB Conf.*, to appear, Vancouver, Canada, August 1992.

[Chen92b] Chen, Ming-Syan et al, "Scheduling and Processor Allocation for Parallel Execution of multi-join Queries", *Proc. 8th IEEE Data Engineering Conf.*, Phoenix, AZ, Feb. 1992.

[DeWi84] DeWitt, D., et al, "Implementation Techniques for Main Memory Database Systems," *Proc. ACM SIGMOD Conf.*, Boston, MA, June 1984.

[DeWi85] Dewitt, D. and Gerber, R., "Multiprocessor Hash-Based Join Algorithms", *Proc. 11th VLDB Conf.*, August 1985.

[DeWi90] DeWitt, D., et al, "The Gamma Database Machine Project", *IEEE Transactions on Knowledge and Data Engineering*, March 1990.

[DeWi92] Dewitt., D. and J. Gray, "Parallel Database Systems: The Future of High Performance Database Systems", *CACM*, 35(6), June 1992.

[Fink82] Finkelstein, F., "Common Expression Analysis in Database Applications", *Proc. ACM SIGMOD Conf.*, Orlando, FL, June 1982.

[Ghan90] Ghandeharizadeh, S. and D. J. DeWitt, "Hybrid-Range Partitioning Strategy: A New Declustering Strategy for Multiprocessor Database Machines," *Proc. 16th VLDB Conf.*, Melbourne, Australia, Aug. 1990.

[Grae89] Gray, J., ed., "Volcano: An extensible and parallel dataflow query processing system.", *Computer Science Technical Report*, Oregon Graduate Center, Beavorton, OR, June 1989.

[Gran80] Grant, J. and Minker, J., "Optimization in Deductive and Conventional Relational Database Systems", *Advances in Data Base Systems*, September 1980.

[Haas90] Haas, L. et. al., "Starburst Mid-Flight: As the Dust Clears", *IEEE Trans. on Knowledge and Data Eng.*, 2(1), March, 1990.

[Hall74] Hall, P.V., "Common Subexpression Identification in General Algebraic Systems", *Tech. Rep. UKSC 0060*, IBM United Kingdom Scientific Centre, Nov. 1974.

[Hsia91] Hsiao, H. I. and D. J. Dewitt, "A Performance Study of Three High-Availability Data Replication Strategies", *Proc. 1st Int'l Conf. on Parallel and Distributed Information Systems*, Miami Beach, FA, Dec. 1991.

[Ioan90] Ioannidis, Y. and Kang, Y. C., "Randomized Algorithms for Optimizing Large Join Queries", *Proc. ACM SIGMOD Conf.*, Atlantic City, NJ, May. 1990.

[Kris86] Krishnamurthy, R., Boral., H. and C. Zaniolo, "Optimization of Nonrecursive Queries", *Proceedings 12th VLDB Conf.*, August 1986.

[Kits83] Kitsuregawa, M., Tanaka, H., and M. Takagi, "Application of Hash to Data Base Machine and its Architecture", *New Generation Computing*, Vol. 1, No. 1, 1983.

[Livn87] Livny, M., S. Khoshafian, and H. Boral, "Multi-Disk Management Algorithms", *Proc. ACM SIGMETRICS Conf.*, Alberta, Canada, May 1987.

[Ries78] **Ries, D. and R. Epstein, "Evaluation of Distribution Criteria for Distributed Database Systems"**, *UCBERL Technical Report M78/22*, UC Berkeley, May 1978.

[Schn89] Schneider, D. and D. DeWitt, "A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment", *Proc. ACM SIGMOD Conf.*, Portland, OR, June 1989.

[Schn90] Schneider, D. and D. DeWitt, "Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines," *Proc. 16th VLDB Conf.*, Melbourne, Australia, Aug. 1990.

[Schw90] Schwetman, H., *CSIM Users' Guide*, MCC Technical Report No. ACT-126-90, Microelectronics and Computer Technology Corp., Austin, TX, March 1990.

[Seli79] Selinger, P. G. et. al., "Access Path Selection in a Relational Database Management System", *Proc. ACM SIGMOD Conf.*, 1979.

[Sell88] Sellis, T., "Multiple Query Optimization", *ACM TODS* 13(1), March 1988.

[Ston86] Stonebraker, M., "The Case for Shared Nothing", *Proc. Int'l Conf. on Data Engineering*, 1986.

[Ston88] Stonebraker, M. et. al., "The Design of XPRS," *Proc. 14th VLDB Conf.*, Los Angeles, CA, September 1988.

[Swam88] Swami. A. and A. Gupta, "Optimization of Large Join Queries", *Proc. ACM SIGMOD Conf.*, June 1988.

[Tand88] Tandem Performance Group, "A benchmark of non-stop SQL on the debit credit transaction", *Proc. ACM SIGMOD Conf.*, Chicago, IL, June 1988.

[Tera85] Teradata Corp., "DBC/1012 Data Base Computer System Manual", *Teradata Corp. Document No. C10-0001-02, Release 2.0*, November 1985

[Vald84] Valduriez, P. and G. Gardarin, "Join and Semi-Join Algorithms for a Multiprocessor Database Machine", *ACM Trans. on Database Systems*, 9(1), March, 1984.

[Yack92] Yackel., J. and Meyer., R. R., "Optimal Tilings for Parallel Database Design", *Advances in Optimization and Parallel Computing*, North-Holland, 1992, ed. P. M. Pardalos.

[Yann87] Ioannidis, Y. and Wong., E., "Query Optimization by Simulated Annealing", *Proc. ACM SIGMOD Conf.*, June 1987.

[Zipf49] Zipf., G. K., "Human Behavior and the Principle of Least Effort", *Addison Wesley*, Reading, MA, 1949.