A Multiuser Performance Analysis of Alternative Declustering Strategies

by

Shahram Ghandeharizadeh
David J. DeWitt

Computer Sciences Technical Report 855
June 1989

A Multiuser Performance Analysis of Alternative Declustering Strategies

Shahram Ghandeharizadeh
David J. DeWitt

Computer Sciences Department
University of Wisconsin - Madison

This research was partially supported by the Defense Advanced Research Projects Agency under contract N00039-86-C-0578, by the National Science Foundation under grant DCR-8512862, by a Digital Equipment Corporation External Research Grant, and by a research grant from Tandem Computer Corporation.

Abstract

In multiprocessor shared-nothing database machines, the storage organization for a relation is independent of the strategy used to partition the relation. A database administrator for such a system has a wide range of alternatives when creating a relation. In this paper, we analyze the impact of three alternative partitioning strategies on the selection queries using different storage/access structures in a multiuser environment. We quantify the tradeoffs of each organization in the context of the Gamma database machine. The response time and throughput of the system are used as the performance metric for evaluating the alternative partitioning strategies.

1. Introduction

During the past few years, the multiprocessor database machine architectures have become increasingly popular. Shared-nothing multiprocessor database machine architectures are especially attractive since they are both scalable and reliable [STON86, BORA88, DEWI88]. Commercially available systems of this type include Non-Stop SQL [TAND88] and the DBC/1012 database machine [TERA85]; research prototypes include Gamma at the University of Wisconsin [DEWI86], Bubba at MCC [ALEX88], and ARBRE at IBM Almaden [LORI88]. In multiprocessor database machines that utilize the concept of **horizontal partitioning** [RIES78] to distribute the tuples of each relation across multiple disk drives, the strategy used for partitioning a relation is independent of the storage structure used at each site. The database administrator (DBA) for such a system must consider a variety of alternative organizations for each relation. The Gamma database machine currently provides the DBA with three alternative partitioning strategies: 1) round-robin, 2) hash partitioning, and 3) range partitioning with administrator-specified key values. In the first strategy, the tuples in a relation are distributed in a round-robin fashion among the disk sites. In effect, this results in a completely random placement of tuples with a uniform distribution of tuples across the sites with disks. In the hash partitioning strategy, a randomizing function is applied to the partitioning attribute of each tuple to select a home site for that tuple. In the last strategy, the DBA specifies a range of key values for each partition or site. This strategy gives a greater degree of control over the distribution of tuples across the sites. For each relation, the DBA must also select a storage structure. Gamma currently provides three alternative storage structures : 1) heap, 2) clustered index, and 3) non-clustered index.

In order to motivate this study, consider the following relation from a stock market database domain:

STOCK (ticker-symbol, name, highest, lowest, closing, opening, P/E)

The STOCK relation consists of a ticker_symbol which uniquely identifies a company (e.g., AXP represents American Express), the name of the company (e.g., American Express), the highest, lowest, closing and opening price of a share of that company, and the price earning estimate attribute (P/E) which provides the potential buyers with an approximate appraisal of a company. Assume that 80% of the queries (termed **type-one queries**) retrieve a single record by specifying the ticker_symbol of that company (e.g., retrieve STOCK.all where STOCK.ticker_symbol = AXP). The other 20% of the queries (termed **type-two queries**) retrieve the companies whose price to earning estimate ratio is within a specified range of values (e.g., retrieve STOCK.all where STOCK.P/E > 10). For this relation and workload, the optimal horizontal partitioning strategy is to hash partition the stock relation on the ticker_symbol attribute and the appropriate access structure at each site is a non-clustered index structure on the ticker_symbol and

a clustered index structure on the P/E attribute. By hash partitioning on the ticker_symbol attribute, the query optimizer can direct the type-one queries to the processor that will contain the record corresponding to the specified company. After the query arrives at the processor, the processor uses the non-clustered index on the ticker_symbol attribute to efficiently retrieve the specified tuple. Type-two queries will be directed to all the processors, where they will be processed using the clustered index on the price attribute (avoiding a sequential scan of the entire relation).

Directing all type-one queries to a single site is very advantageous, since executing a query in a distributed fashion has two major costs: 1) communication overhead for scheduling the query, and 2) the overhead of the distributed commit protocol. In [DEWI88], we demonstrated that if a query requires a limited amount of CPU and disk resources, using parallelism to execute it will result in the above two costs appearing as pure extra overhead. In this study, the number of processors in the environment was increased from one to eight while observing the response time of a 0% selection range query using a clustered or non-clustered access method. As the number of processors was increased, the response time of the query actually increased from 0.25 seconds to 0.58 seconds. This occurred because the cost of initiating and committing the query at each of the sites was higher than the cost of performing one or two I/O operations to search the index structure. Therefore, directing the single record retrieval queries to only one of the sites would result in the best response time for 80% of the workload.

In this paper we study the impact of alternative partitioning strategies and storage organizations on different selection query types in a multiuser environment. We also quantify the tradeoffs of each organizations in the context of the Gamma database machine. The remainder of this paper is organized as follows. Section 2 contains the description of the workload that we developed for this performance evaluation. A brief overview of the Gamma database machine is presented in Section 3. The performance of the alternative partitioning strategies is presented in Section 4. Our conclusions appear in Section 5.

2. Workload Definition

Early in this study, we recognized the need for a multiuser workload to evaluate the performance of the alternative partitioning strategies since the response time of a query and the processing effort of the system (total amount of CPU, I/O bandwidth and other resources in the environment) required to execute the query with each of the alternative partitioning strategies are not necessarily correlated. For example, the response time of a single tuple retrieval query via a heap storage structure is the same for both the range and round-robin partitioning strategies.

However, the range partitioning strategy directs the query[1] to a single processor while the round-robin partitioning strategy directs the query to all the processors containing the fragments of the referenced relation. Consequently, the amount of resources (CPU and I/O) consumed by the round-robin partitioning strategy would be n times that consumed by the range partitioning strategy (where n is the total amount of processors used to execute the query). Thus, the decision was made to use both the response time and throughput of the system in a multiuser environment as the performance metric for evaluating the alternative partitioning strategies.

The transaction processing benchmark (TP1) [ANON84] was not sufficient for this performance evaluation since the only query in this workload that measures throughput (i.e., DebitCredit) can only evaluate a single aspect of the alternative partitioning strategies. For example, consider the hash and range partitioning strategies. The hash partitioning strategy directs single tuple selection queries on the partitioning attribute to a single site and range selection queries to all the sites. The range partitioning strategy, however, localizes the execution of both the single tuple and range selection queries. Consequently, since the DebitCredit query deals only with single tuple operator (selection and update), it cannot be used to evaluate the difference between these two partitioning strategies for range queries.

The workload that we designed for evaluating the performance of the alternative partitioning strategies is very general. It was not made specific because it would have limited the capability of the benchmark while making the experiments appear very dependent on the specific characteristics of the benchmark. Rather, the goal was to design a general workload model with the potential for scalability.

There are three principal factors that affect the performance of the queries with the alternative partitioning strategies: 1) the multiprogramming level, 2) the degree of data sharing, and 3) query mix. Each factor defines an axis in the performance space that can be varied while keeping all the other factors constant. The multiprogramming level refers to the number of concurrently executing queries. The parsing and access path selection stages of query execution were removed by embedding the query in a program and compiling it. Below, we will analyze and discuss a general methodology developed for supporting the remaining two factors.

---

[1] Assuming the selection predicate is applied to the same attribute as the one used to partition the relation.

2.1. Degree of Data Sharing

This variable controls the degree of data sharing [BORA84] in the buffer pool. With a high degree of data sharing, the concurrently executing queries would almost certainly compete for only the CPU. However, a low degree of data sharing results in competition for CPU, buffer pool pages, and I/O bandwidth among the concurrently executing queries.

We decided to implement the different degrees of data sharing by controlling the frequency of accesses made to the relations in a fixed size database. The decision was made to support three degrees of data sharing: low, high, and medium. To implement them, a fixed database size with **m** relations was designed. The cardinality of each relation was kept at **n** tuples (total size of database (in bytes) = m * n * size of each tuple). Below, we will describe how the frequency of accesses made to the relations was manipulated in order to provide the desired effect.

We interpreted a low degree of data sharing to mean a database in which the frequency of access to each relation is the same. Thus, for a single user accessing a database of m relations, each relation in the database has a 1/m probability of access. A random number generator was used to create a uniform distribution of accesses to the database.

A high degree of data sharing is interpreted to mean a database in which a small number of the relations are

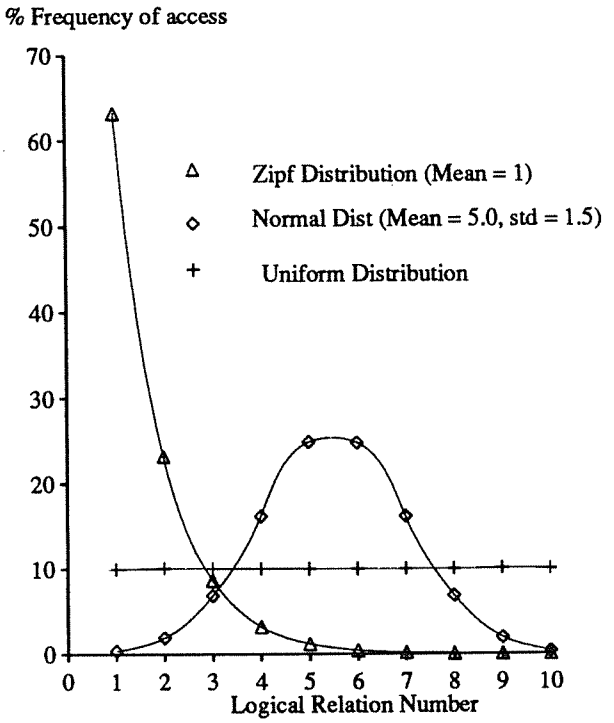


Figure 1

accessed most frequently. For example, 80 to 90 percent of all accesses are made to 15 to 25 percent of the database. For this, we used the Zipfian distribution with a mean of 0.1 * m (recall that m is the number of relations in the database) which results in 86% of accesses going to 20% of the database.

We interpreted a medium degree of data sharing to mean a database in which the frequency of access to the relations is not as skewed as the high degree of data sharing and not as uniform as the low degree of data sharing. For our purposes we chose to divide the database into five equal size regions. In addition, we chose the probability of access to each region to be 45%, 30%, 15%, 7%, and 3% respectively. Within each region we wanted a uniform distribution of access (e.g., if the complete database consists of 100 relations, the first region will have 20 relations and each relation within that region has a 2.25% probability of access, the second region consists of 20 relations and each relation within that region has 1.5% probability of access, etc.). A normal distribution with a mean of m/2 and a standard deviation of 0.15 * m has these characteristics.

Figure 1 presents each of the three different distributions of accesses to a database consisting of ten relations (i.e., m = 10). This figure provides a comparison point for the three different distributions of accesses supported by the multiuser benchmark used for this evaluation.

In order to illustrate how the alternative distributions of access result in different degrees of data sharing in the buffer pool, consider the execution of an exact match query using a non-clustered index. With the Zipfian distribution of access, most of the concurrently executing queries access a single relation. Consequently, the probability of buffer pool hits for index pages of the B-tree corresponding to that relation (especially the root page of the B-tree) is very high. On the other hand, with the uniform distribution of access, the frequency of access to a relation is the same as any other relation in the database. Thus, the percentage of buffer pool hits for index pages is much lower. Furthermore, at high multiprogramming levels, the index pages of several relations might compete for buffer pool page frames and result in a thrashing behavior[2] of the buffer pool.

While the frequency of accesses chosen for each region of database within a given degree of data sharing is quite arbitrary, these patterns of access were selected as what we believe to be representative of high, low, and medium degrees of data sharing. Our contribution is not the definition of these absolute access frequencies, rather, it is the definition of the degree of data sharing as a pattern of accesses to a fixed size database.

---

[2] Assuming an LRU replacement policy.

2.2. Query Types

While the performance of each of the relational algebra operators is impacted by the alternative partitioning strategies and storage structures, we decided to only study their impact on the performance of selection queries. We made this decision because the performance of this operator has a significant impact on the performance of other complex query plans. In particular, the alternative partitioning strategies directly affect the response time and throughput of selection queries since the alternative partitioning strategies either localize the execution of selection queries to a subset of the nodes with disks (e.g., range) or distribute its execution across all the nodes with disks hosting the fragments of the relation(e.g., round-robin).

The selection queries we studied include: 1) single tuple retrievals via a heap storage structure, 2) 10% selection queries using a clustered access method, 3) single tuple retrievals using a non-clustered access method, and 4) 0.125% selection queries using a non-clustered index. The single tuple retrieval query via a heap storage structure is representative of I/O intensive selection queries with sequential disk accesses. The 10% selection query via a clustered access method is representative of CPU intensive queries as the index enables the system to retrieve only the relevant tuples from the disk. In addition, the disk accesses are sequential which tends to make the query CPU bound[3].

The single tuple retrieval query using a non-clustered index access method[4] represents the class of queries which consume the minimal amount of resources. It is extremely important to provide the best response time and throughput for this query type. As we will show in Section 4, the alternative partitioning strategies have a significant impact on the performance of this query type. The 0.125% selection using a non-clustered access method represents queries with random disk accesses.

These selection queries were designed to retrieve the result tuples back to the host and not to store them in the database. Storing the results of a selection query in the database will interfere with the processing of the selection operators. This is undesirable, since our goal is to measure the impact of the alternative partitioning strategies on the selection operator without interference from any external variable.

---

[3] This is due to the one to one correspondence between the order of the index records and the data records.

[4] The performance of the single tuple query using a non-clustered index is identical to that using a clustered index since the depth of the B-tree is traversed and the appropriate data tuple is retrieved for both access methods.

2.3. Description of Benchmark Relations

The benchmark relations used for this performance evaluation are based on the standard Wisconsin Benchmark relations [BITT83]. The database used for this performance evaluation consisted of ten relations. The cardinality of each relation is 100,000 tuples. Each relation consists of thirteen 4-byte integer attributes and three 52-byte string attributes. Thus, each tuple is 208 bytes long.

3. Overview of the Gamma Database Machine

In this section, we present a brief overview of the Gamma database machine. For a complete description of Gamma database machine see [DEWI86, GERB86]. Included in this discussion is the current hardware configuration[5] and software techniques used in implementing Gamma. More detailed description of the algorithms used for implementing the various relational operations are presented in the performance evaluation of this paper.

Presently, Gamma consists of 17 VAX 11/750 processors, each with two megabytes of memory. An 80 megabit/second token ring [PROT85] is used to connect the processors to each other and to another VAX 11/750 running Berkeley UNIX. This processor acts as the host machine for Gamma. Attached to eight of the processors are 333 megabyte Fujitsu disk drives (8") which are used for database storage. One of the diskless processors is currently reserved for query scheduling and global deadlock detection. The remaining diskless processors are used to execute join, projection, and aggregate operations. Selection and update operations are executed only on the processors with disk drives attached.

Gamma is built on top of an operating system developed specifically for supporting database management systems. NOSE provides lightweight processes with shared memory and reliable, datagram communication services between NOSE processes on Gamma processors and to UNIX processes on the host machine using a multiple bit, sliding window protocol [TANE81]. Messages between two processes on the same processor are *short-circuited* by the communications software.

File services in Gamma are based on the Wisconsin Storage System (WiSS) [CHOU85]. These services include structured sequential files, B^+ indices, byte-stream files as in UNIX, long data items, a sort utility, a scan mechanism, and a 2-phase concurrency control algorithm.

---

[5]Recently, we have ported the Gamma software to an Intel iPSC-II Hypercube with thirty-two 386 processors and thirty-two disk drives and we will be retiring the VAX configuration in the near future.

Gamma uses traditional relational techniques for query parsing, optimization [SELI79, JARK84], and code generation. Queries are compiled into a tree of operators with predicates compiled into machine language. After being parsed, optimized, and compiled, the query is sent by the host software to an idle scheduler process through a dispatcher process. The scheduler process, in turn, activates operator processes at each query processor selected to execute the operator. The task of assigning operators to processors is performed in part by the optimizer and in part by the scheduler assigned to control the execution of the query. For example, the operators at the leaves of a query tree reference only permanent relations. Using the query and schema information, the optimizer is able to determine the best way of assigning these operators to processors. The results of a query is either returned to the user through the ad-hoc query interface or through the embedded query interface to the program from which the query was initiated.

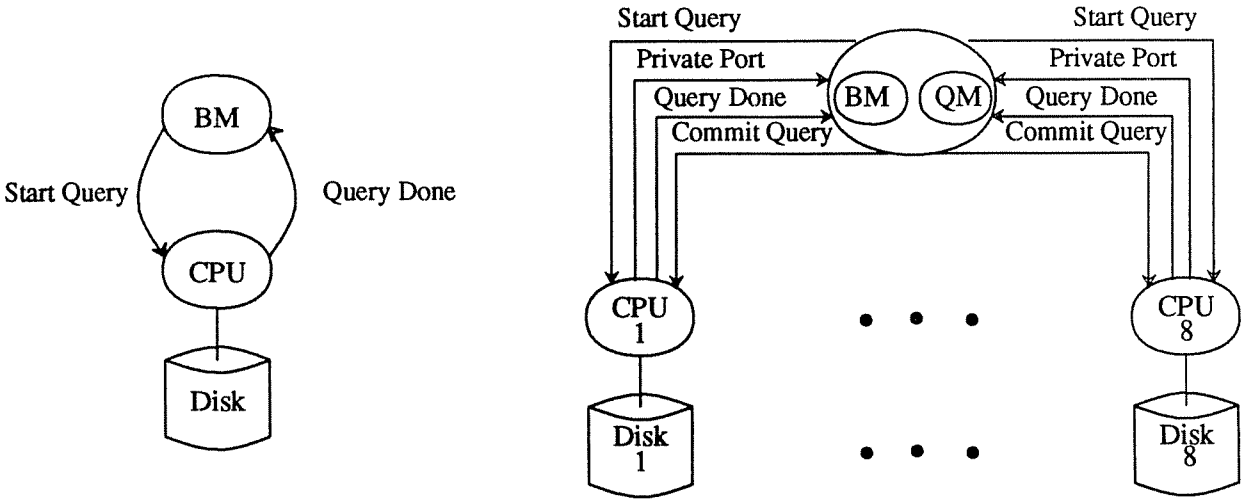
4. Performance of Alternative Partitioning Strategies for Selection Queries

In this section, we will present the performance of the various selection query types with the alternative partitioning strategies in an eight processor Gamma configuration with a disk drive attached to each processor. Each of the alternative partitioning strategies distributed the tuples of each relation uniformly across the eight processors. Before analyzing the performance of each query, we will describe the query execution paradigm for each of the alternative partitioning strategies.

As illustrated by Figure 2, Gamma executes single site queries differently than multisite queries. The benchmark process (BM) and the query manager process (QM) are two independent entities that are generally located on different processors, but were placed on the same processor for these experiments. There can be several BM and QM processes per processor.

The BM process acts as a terminal, generating queries and maintaining the response time of each individual query. The BM process utilizes a uniform random number generator to generate the attribute value(s) of the tuple(s) retrieved by a query. Since uniform random number generator results in an equal probability of accesses to the tuples of a relation, in the multiprocessor configuration, the probability of a tuple being selected from a fragment of a relation is equivalent to the probability of a tuple being selected from any other fragment of the same relation. The results in this section will reveal that this random nature of the workload can adversely affect the performance of these partitioning strategies that direct a give query type to a subset of the sites rather than to all the sites.

The QM process controls the execution of a query requiring the participation of two or more sites (a



a. Single Site Query

b. Multi Site Query

Figure 2

multisite query). When a multisite query is submitted to the QM process, the QM sends the query to each processor that participates in the execution of the query and requests the communication port of the process that will be executing the query at that site (see Figure 2.b). When the process at a site finishes executing the query, it sends the QM a "query done" message. If the QM receives a "query done" message from all the processors participating in the execution of the query, it sends a "commit" message to each site and closes all the communication ports. Finally, it notifies the process that submitted the query of its successful. If, on the other hand, the query is aborted at a site, that site sends an "abort" message to the QM. The QM, in turn, sends an "abort" message to each participating site and then notifies the process that submitted the query.

As discussed in the introduction of this paper, the optimizer utilizes the information provided by the range and hash partitioning strategies to direct certain queries to a single processor (a single site query). When the BM process creates a single site query, the query is sent directly to the appropriate processor for execution (see Figure 2.a). Appropriate flags are set in the query packet to notify the processor that it is in complete control of the query and that it should notify the BM process with the outcome of the query.

When the query requires the participation of two or more sites, the BM process submits the query to a QM process. Because the BM process lacks the distributed commit protocol outlined above, it is not capable of controlling the execution of a multisite query.

Figure 2 reveals that fewer messages are required to control the execution of a single site query than a multisite query. This is an important factor and will have a significant impact on the performance of the alternative partitioning strategies.

The physical characteristics of each site is as follows. The disk page size at each site is 8 Kbytes. The disk scheduler uses an elevator algorithm. The maximum number of queries executing concurrently at a processor is limited to 30 queries since each sequential scan query requires two buffer pool pages, one for processing the current page and the other for the read-ahead to read the next data page into[6]. Due to memory limitations, the size of the buffer pool had to be limited to sixty-one 8 Kbyte pages (i.e., 500 kilobytes). Consequently, the maximum number of concurrently executing queries at each processor was limited to 30 queries ($\frac{61}{2}$). The buffer pool replacement policy is LRU.

In the sections below, when we refer to the performance of the multiprocessor configuration with a specific partitioning strategy for a query type, we are implying that the selection predicate of the query is applied to the same attribute as the one used to partition the relation. When the selection predicate is applied to an attribute other than the partitioning attribute (with or without an access method on the non-partitioning attribute), the execution of the query is identical to that with the round-robin partitioning strategy[7]. Thus, whenever the performance of the multiprocessor configuration with the round-robin partitioning strategy is discussed, the discussion and the associated performance evaluation can also be viewed as that of the multiprocessor with the range or hash partitioning strategies where the selection predicate is applied to an attribute other than the partitioning attribute.

In the rest of this section, we will present and discuss the response time and throughput of each query type with each of the alternative partitioning strategies. CPU and disk utilization measurements of the system are also presented in order to help clarify our explanations and to provide additional insight into the performance of the system with the alternative partitioning strategies.

---

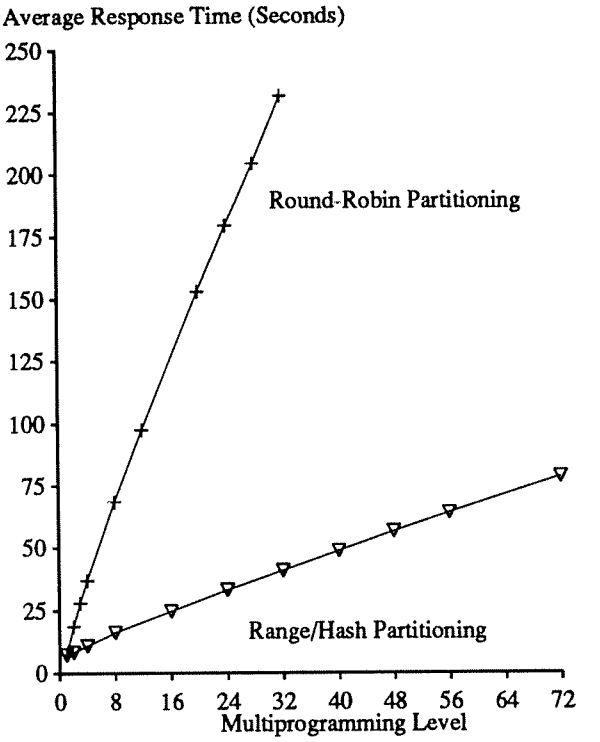
[6] A one page read-ahead mechanism is utilized when scanning a file sequentially.

[7] Assuming no correlation exists between the value of the partitioning attribute and the value of the non-partitioning attribute that the selection predicate is applied to.

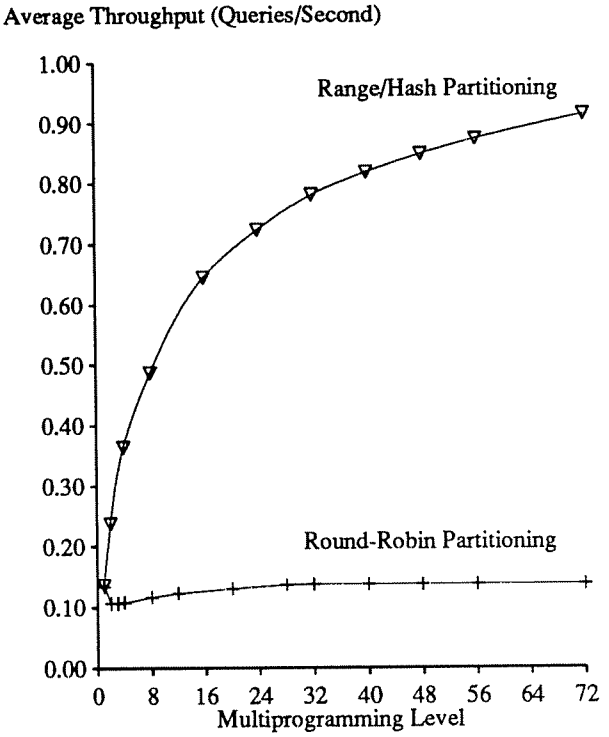
4.1. Single Tuple Selection Via a Heap Storage Structure

With a heap storage structure, all the data pages in the relation must be sequentially retrieved from the disk and processed. A one page read-ahead mechanism is utilized when scanning a file sequentially, enabling the processing of one page to be overlapped with the I/O for the subsequent page. The response time and throughput for the alternative partitioning strategies with a low degree of data sharing is presented in Figures 3 and 4[8], respectively. First, consider the performance of the system with the round-robin partitioning strategy.

The round-robin partitioning strategy directs this query to the processors containing the fragments of the referenced relation. The first interesting fact to note is that the throughput for this partitioning strategy actually



Single Tuple Selection Via a Heap Storage Structure
Low Degree of Data Sharing
Figure 3

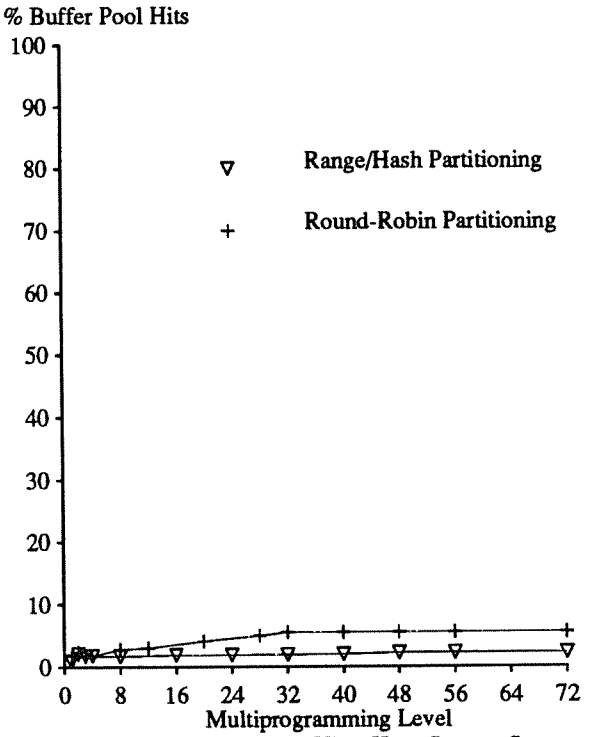


Single Tuple Selection Via a Heap Storage Structure
Low Degree of Data Sharing
Figure 4

[8] Due to the random nature of the workload, all the experiments presented in this paper had to run long enough to provide meaningful measurements. In addition, statistical analysis of measurements is required to establish the validity of the results. We used batch-mean analysis for establishing a confidence interval. The number of batches in each presented experiment was set to 10. Analysis of the response time measurements indicates that all the confidence intervals are within ±2.5% of the mean response time (95% confidence interval). The throughput of the system was calculated using the average response time.

decreases from a multiprogramming level of one to two. At a multiprogramming level of two, the two concurrently executing queries generally access different relations. This results in a sequence of random disk requests rather than a series of sequential requests which would occur had each query been submitted individually. The random disk requests result in a higher average seek time than the sequential disk requests.

The throughput of the system with the round-robin partitioning begins to increase at a multiprogramming level of three. At first glance, one might attempt to attribute this to a higher percentage of buffer pool hits. However, as shown in Figure 5, the percentage of buffer pool hits increases only slightly with higher multiprogramming levels[9]. Rather, the major reason for an increase in the throughput of the system is the utilization of the elevator algorithm by the disk controller of the system. By utilizing this algorithm, the distance traveled by the head of the disk decreases for several (more than two) pending I/O requests and, thus, the average service time of the disk decreases at multiprogramming levels greater than two.



Single Tuple Selection Via a Heap Storage Structure
Figure 5

[9] There is a 30% increase in the throughput of the system from a multiprogramming level of 2 to 28, while the percentage of buffer pool hits increases by less than 3%.

Finally, the throughput of the system with the round-robin partitioning strategy levels off at multiprogramming levels higher than 30 since the maximum number of queries concurrently executing at a processor is limited to 30.

With the range and hash partitioning strategies, the optimizer has enough information to enable it to localize the execution of this query to a single processor[10]. Therefore, the BM process sends each query directly to the appropriate processor, circumventing the QM.

Since there are eight processors in the multiprocessor configuration, one would have expected the throughput of the system to increase linearly from a multiprogramming level of one to eight. While the random number generator used to generate selection predicates results in a uniform distribution of queries across all the processors, this does not guarantee that, at any instant in time, there will be an equal number of queries executing concurrently at each processor. To illustrate this point, in Table 1 the percentage of queries executed at each processor and the number of queries executed concurrently at each processor at a multiprogramming level of two is presented. The results reveal that each processor executes approximately the same percentage of queries (approximately 12.5%). However, these results also indicate that there were two concurrently executing queries on the same processor almost 15% of the time. If the two concurrently executing queries had been distributed uniformly across the eight processors at every instant in time, then a single processor should have never observed two concurrently executing queries. Thus, the throughput of the system cannot increase linearly from a multiprogramming level of one to two, since two queries were competing for resources on a single processor almost 15% of the time, resulting in a

Table 1
Distribution of Workload Across the Processors (MPL = 2)

Processor Number	Percentage of Total Queries Executed	Percentage of Time Executing One Query	Percentage of Time Executing Two Queries
1	13.00%	88.46%	11.54%
2	12.25%	84.44%	15.56%
3	12.50%	90.00%	10.00%
4	13.00%	84.38%	15.63%
5	12.00%	89.58%	10.42%
6	13.00%	88.46%	11.54%
7	12.25%	84.44%	15.56%
8	12.00%	90.91%	10.09%

[10] The results for both partitioning strategies are identical and are presented as a single curve.

sequence of random disk requests rather than strictly sequential disk requests at that processor.

The results presented in Figures 3 and 4 reveal a lower system response time with the range and hash partitioning strategies than for the round-robin partitioning strategy at all multiprogramming levels. In addition, the throughput of the system is significantly higher with the range and hash partitioning strategies than with the round-robin partitioning strategy. The major reason for this is that with the range and hash partitioning strategies, the execution of each concurrently executing queries is localized to a single processor; resulting in less interference among the concurrently executing queries. Consequently, the percentage of random disk requests at each processor is lower with the range and hash partitioning strategies.

While the results obtained with a high degree of data sharing are interesting, due to lack of space, we elected to eliminate them from this discussion. Rather, we will summarize the results obtained. For a high degree of data sharing, the drop in throughput of the system for the round-robin partitioning strategy from a multiprogramming level of one to two was smaller due to a lower average disk seek time. With a high degree of data sharing, two concurrently executing queries generally access a single relation, while for a low degree of data sharing, the two concurrently executing queries may access any relation in the database. Consequently, the average disk seek time is lower since the head of the disk drive must travel a shorter distance for each disk request.

At high multiprogramming levels, the concurrently executing queries become "synchronized"[11] for the round-robin partitioning strategy with a high degree of data sharing. This resulted in a significant increase in the percentage of buffer pool hits (up to 80%) and the maximum throughput of the system was more than twice that with a low degree of data sharing. By introducing think time in the workload model, we speculate that this "synchronization" affect would disappear.

A high degree of data sharing did not have as significant an impact on the range and hash partitioning strategies as it did on the round-robin partitioning strategy since the probability of queries becoming "synchronized" is much lower for these partitioning strategies. For the range and hash partitioning strategy, not only does each query randomly select a relation to access, but it also randomly selects a processor to execute on. Since the random nature of the workload results in an unequal number of concurrently executing queries at each site, the load characteristics of each processor is slightly different than the others. Consequently, the probability of a set of queries becoming

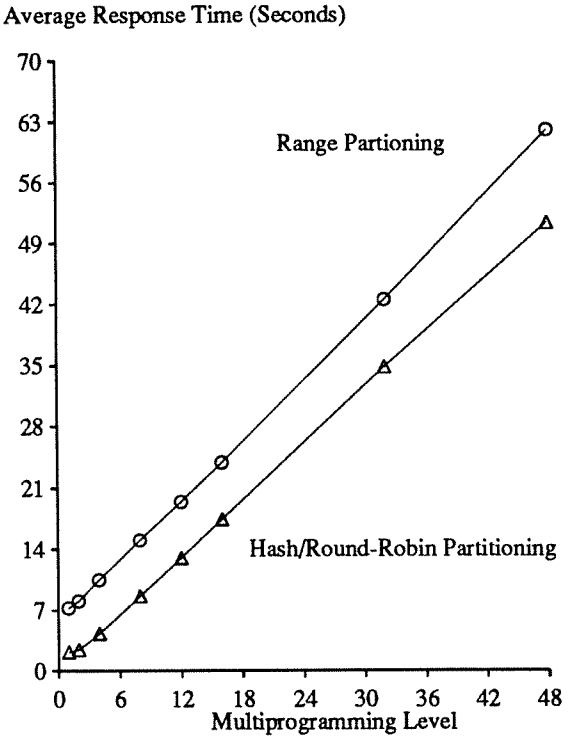
[11] A set of queries is considered "synchronized", if their access to the data pages of a relation are close enough together in time that they can share each others data pages in the buffer pool.

"synchronized" and capable of sharing data pages in the buffer pool is very low. For a high degree of data sharing, the throughput of the system with the range and hash partitioning strategies was again significantly higher than that for the round-robin partitioning strategy.

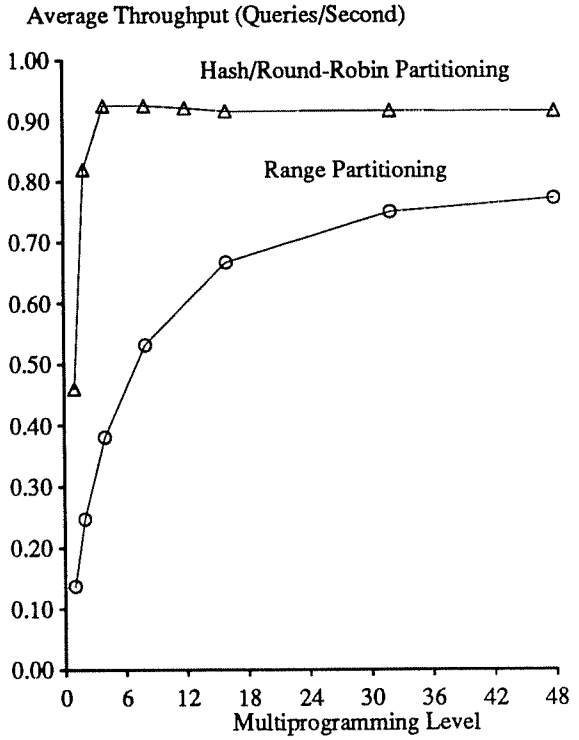
4.2. 10% Selection Query Using a Clustered Index Structure

With a clustered index structure, the order of the values of the index records in the sequence set of the B-tree is the same as the order of the data records in the relation. Two types of disk requests are made by range selection queries that use a clustered index structure: random and sequential. The traversal of the B-tree to locate the upper and lower limits of the query with respect to the actual data records is random, while the retrieval of data records between the lower and upper limit markings is sequential.

In Figures 6 and 7, the response time and throughput of Gamma for a 10% selection query with a clustered index is presented for each of the multiprogramming levels. We have included only results for a high degree of data sharing (Zipfian distribution of access) as the performance of the system is almost identical for the other two



10% Selection Using a Clustered Index
High Degree of Data Sharing
Figure 6



10% Selection Using a Clustered Index
High Degree of Data Sharing
Figure 7

degrees of data sharing. With the hash partitioning strategy, all the sites must participate in the execution of a range query regardless of which attribute the selection predicate is applied to. Therefore, the execution paradigm of the 10% selection query is identical for both the hash and round-robin partitioning strategies and is presented as a single curve rather than two overlapping curves.

The average response time of the system with the hash and round-robin partitioning strategies increases only slightly as the multiprogramming level is increased from one to two (the throughput of the system almost doubles). Both, the average CPU and disk utilization of the system are very low at a multiprogramming level of one since the overhead of controlling the execution of the query constitutes the majority of the query execution time. At a multiprogramming level of two, the idle CPU and I/O bandwidth are consumed by the additional query. The throughput of the system does not exactly double since the queries are sometimes scheduled simultaneously on the same processor.

The average CPU utilization of the system reaches 100% at a multiprogramming level of 4 for the hash and round-robin partitioning strategies and, consequently, the throughput of the system levels off at this point

Since the range partitioning strategy will localize the execution of all range queries on the partitioning attribute, a 10% selection query will be localized to either one or two processors. When the execution of the query is localized to a single processor, that query must retrieve 80% of the tuples of the fragment of the source relation residing at that processor ($\frac{10,000}{12,500}$). The throughput of the system with the hash or round-robin partitioning

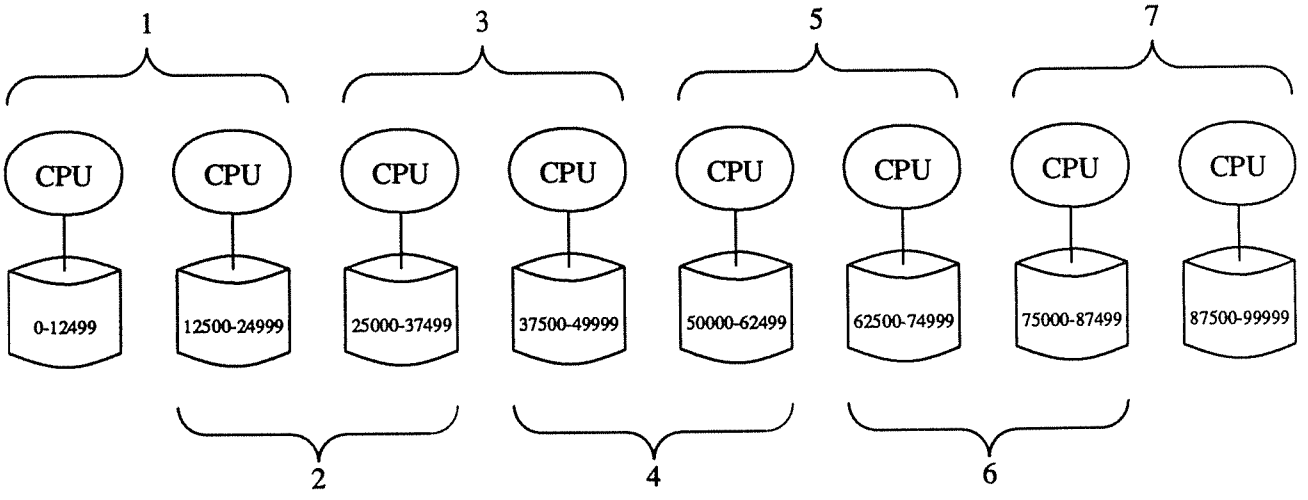
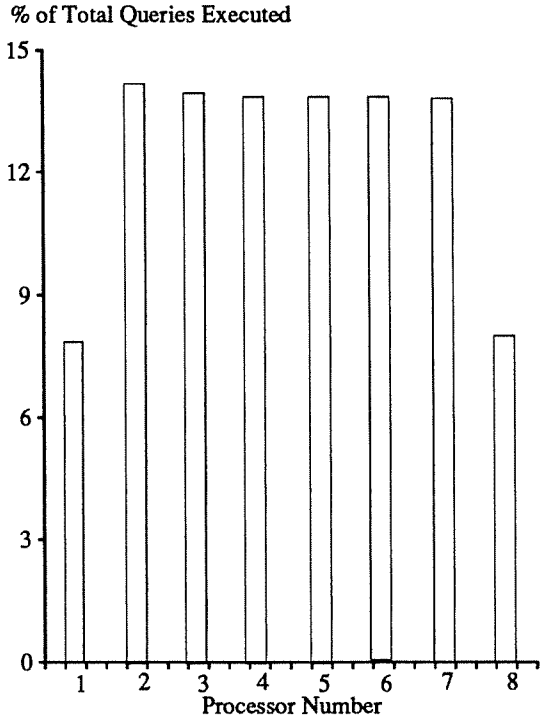


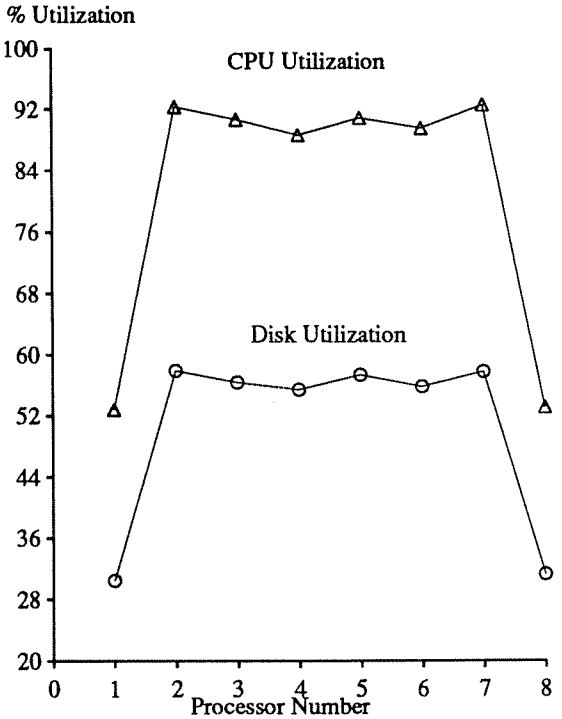
Figure 8

strategy is, however, only 3.34 times higher at a multiprogramming level of one. There are two main reasons why the difference is not a factor of eight. First, the hash and round-robin partitioning strategies incur the overhead associated with controlling the execution of the query at all eight processors. Second, for the range partitioning strategy, the range of the attribute values retrieved by a query generally overlaps the range of values of two adjacent processors since the selectivity factor of the query is fairly large. This results in the participation of two sites rather than a single site in the execution of the query; further reducing the number of pages that must be processed by a single processor.

The throughput of the system at a multiprogramming level of one with the range partitioning strategy is 0.14 queries per second. Since there are eight processors with disks in the environment, one would expect a maximum throughput of 1.01 queries per second at high multiprogramming levels. Figure 7 reveals, however, that the throughput of the system at a multiprogramming level of forty eight is only 0.77 queries per second. In order to explain this, consider all possible combinations of two sites participating in the execution of a 10% range selection query. As illustrated by Figure 8, there are only seven possible ways for a range query to overlap two processors.



10% Selection Using a Clustered Index (MPL = 48)
Figure 9



10% Selection Using a Clustered Index (MPL=48)
Figure 10

Observe that processors containing the upper and lower limits of the partitioning attribute values (processors 1 and 8) can participate in only one combination, while all the other processors each participate in two possible combinations. Therefore, at all multiprogramming levels, processors 1 and 8 have a lower probability of being accessed. In Figure 9, the percentage of queries executed by each processor at a multiprogramming level of forty eight is presented. These results reveal that processors one and eight participated in the execution of fewer queries than the remaining processors. This is also evident from the CPU and disk utilization of the individual processors (see Figure 10). Thus, the range partitioning strategy utilizes only six out of the eight processors to their maximum capacity while the hash and round-robin partitioning strategies fully utilize all the eight processors. Consequently, the throughput of the system with the range partitioning strategy is lower than that for the hash and round-robin partitioning strategies. Furthermore, the maximum throughput of the eight processor configuration with the range partitioning strategy is only six times the throughput at a multiprogramming level of one.

Unlike the single tuple retrieval via a heap storage structure, the round-robin and hash partitioning strategies outperform the range partitioning strategy for this query type. With a clustered index access method, only the relevant tuples are retrieved from the database. The range partitioning strategy results in the sequential retrieval of all the tuples from a single (or two) site(s), while the round-robin and hash partitioning strategies utilize all the sites and retrieves approximately $\frac{1}{8}$ of the relevant tuples from each site. Due to the high selectivity factor of this query, utilizing intra-query parallelism results in a lower response time and, hence, the round-robin and hash partitioning strategies outperform the range partitioning strategy.

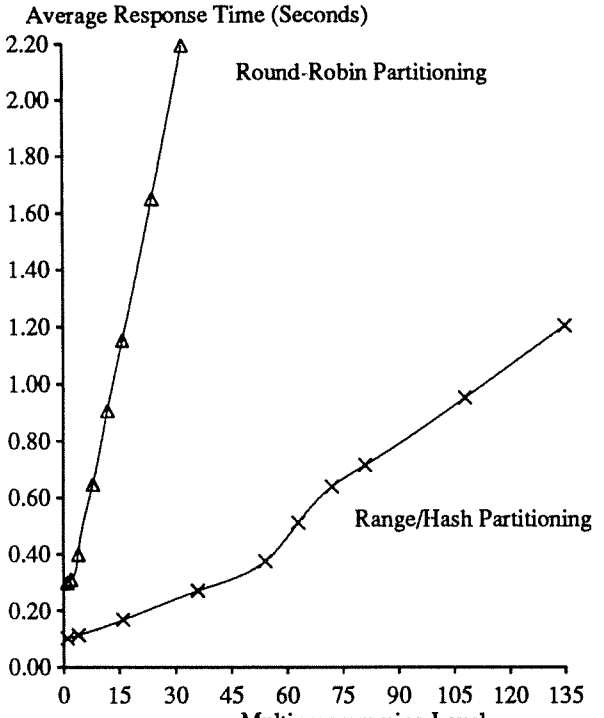
Although the results are not presented here, we also analyzed the impact of the alternative partitioning strategies for a 1% selection query using a clustered index. For this query, the round-robin and hash partitioning strategies outperform the range partitioning strategy at multiprogramming levels lower than 12 since again the round-robin and hash partitioning strategies utilize intra-query parallelism while the range partitioning strategy does not. At multiprogramming levels higher than 12, the range partitioning strategy outperforms the round-robin and hash partitioning strategies. Recall that there is an overhead associated with intra-query parallelism, namely, the overhead of messages required for controlling the execution of a multisite query. At multiprogramming levels higher than 12, the resources in the system are highly utilized with all the partitioning strategies, however, with the range partitioning strategy, the query does not incur the overhead associated with a multisite query. For this query, the range partitioning strategy achieves a maximum throughput of nine queries/second at a multiprogramming level of

80 while the range and hash partitioning strategies obtain a maximum throughput of six queries/second.

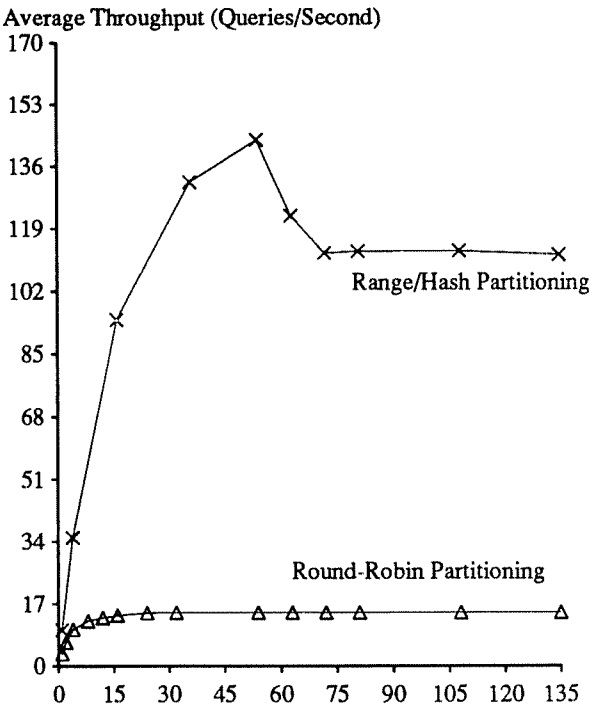
4.3. Single Tuple Selection Using a Non-clustered Index Structure

For a single tuple selection using a non-clustered access method, the B-tree is searched to locate the appropriate index record and a final random disk request is performed to retrieve the appropriate data record. The response time and throughput for the alternative partitioning strategies for this type of query for a low degree of data sharing are presented in Figures 11 and 12, respectively. With the range and hash partitioning strategies, the optimizer has enough information to localize the execution of the exact match queries on the partitioning attribute value to a single processor. Thus, these two partitioning strategies perform identically and are presented as a single curve in Figures 11 and 12.

At a multiprogramming level of one, the response time of the system with the range and hash partitioning strategies is approximately $\frac{1}{2}$ that of the round-robin partitioning strategy since the query is directed to all the processors by the round-robin partitioning strategy. The execution time of the query at each processor is so low that the



Single Tuple Selection Using a Non-Clustered Index
Low Degree of Data Sharing
Figure 11



Single Tuple Selection Using a Non-Clustered Index
Low Degree of Data Sharing
Figure 12

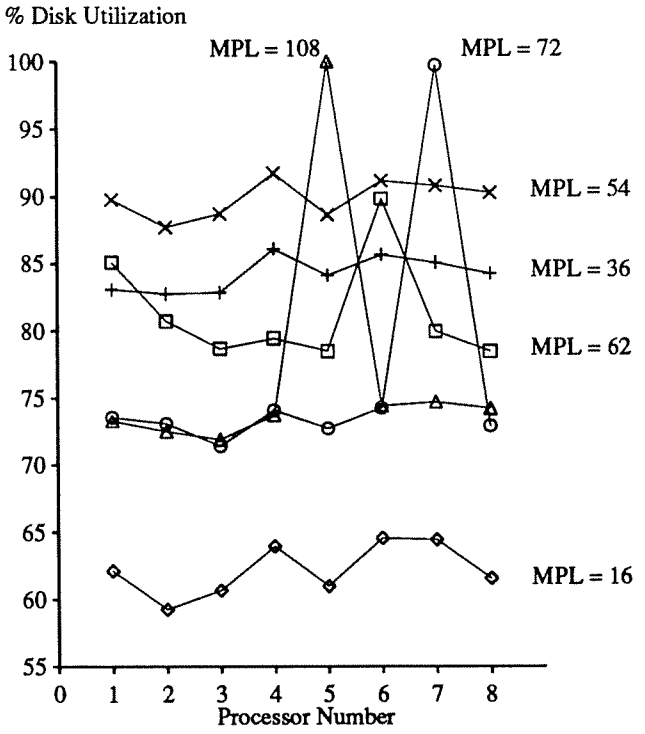
overhead associated with controlling the execution of the query at all the sites constitutes the majority of the response time of the query. With the range and hash partitioning strategies, however, the query can always be directed to a single processor. Thus, the overhead associated with a multisite query is not incurred.

With the round-robin partitioning strategy, one of the eight processors will find the desired tuple while the other processors will traverse the depth of their B-tree index, only to not find a matching tuple. At a multiprogramming level of 24, the CPU of all the processors becomes 100% utilized and the throughput of the system levels off.

With the range and hash partitioning strategies, one might have expected a linear increase in throughput from a multiprogramming level of one (9.8 queries/second) to eight (63.5 queries/second) since the multiprocessor configuration consists of eight processors with disks. This does not occur since the random nature of the workload cannot guarantee that there will be exactly the same number of queries executing concurrently at each processor at every instant in time.

For the range and hash partitioning strategies, the throughput of the system decreases sharply from a multiprogramming level of 54 to 72 and then levels off at multiprogramming levels higher than 72 (see Figure 12). From a multiprogramming level of 54 to 72, one of the processors in the system has started to become a bottleneck. At a multiprogramming level of 52, the bottleneck is temporary but as the multiprogramming level is increased further, the interval of time for which the bottleneck exists increases. For a low degree of data sharing, when a processor attempts to execute more than 16 queries of this type concurrently, its buffer pool begins to thrash. Consequently, the response time of the queries executing on the bottleneck processor is higher than the other queries executing on the other processors. Therefore, the average response time increases, while the average throughput decreases. At a multiprogramming level of 72, one of the processors becomes a permanent bottleneck. However, since the maximum number of concurrently executing queries at a processor is fixed at 30, the thrashing behavior at the bottleneck processor stabilizes. At this multiprogramming level, the maximum rate at which the bottleneck processor executes queries determines how many queries will be executed by the other nodes, and increasing the multiprogramming level results in the formation of a backlog of queries waiting for service outside of the bottleneck processor. The throughput levels off since increasing the number of concurrently executing queries cannot effect the rate at which the bottleneck processor can execute queries.

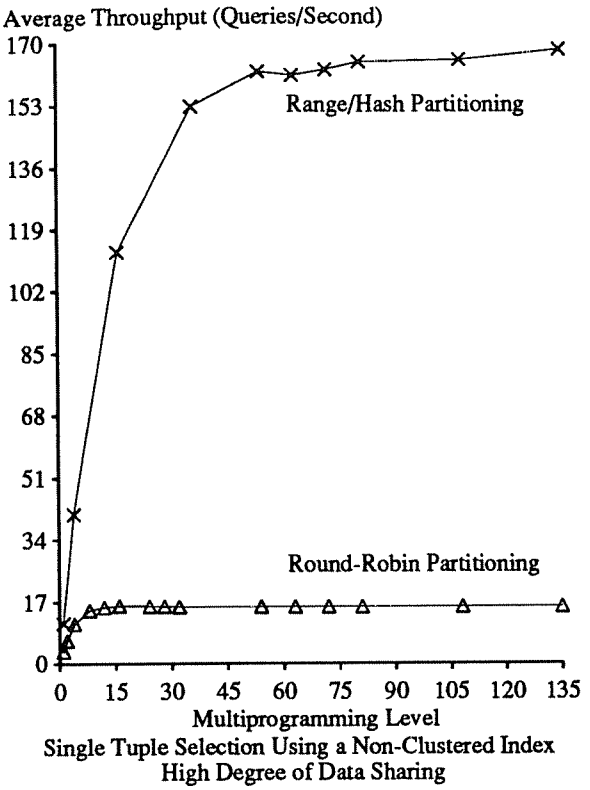
The disk utilization of each processor provides further evidence for this claim (see Figure 13). In this figure, each line represents the utilization of each individual processor at a certain multiprogramming level. As the multiprogramming level is increased from 1 to 54, the disk utilization of the system increases. At a



Low Degree of Data Sharing
Figure 13

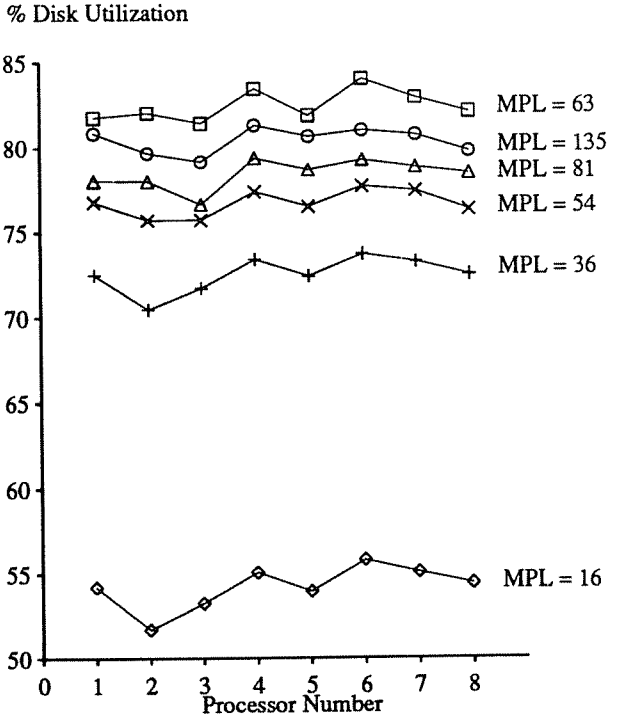
multiprogramming level of 63, the disk utilization of those processors that temporarily become a bottleneck (i.e., processor 1 and 6) is higher than the other processors. This occurs because the buffer pool of these processors thrash, resulting in a high degree of disk utilization. At a multiprogramming level of 72, the disk utilization of the processor that permanently became a bottleneck is 100%. The utilization of the resources in the other processors levels off at a multiprogramming level of 72. Increasing the multiprogramming level beyond 72 does not increase the utilization of these processors since the newly introduced queries will wait outside of the bottleneck processor for service (i.e., the processor observing 30 concurrently executing queries).

Next, consider the performance of this query type with a high degree of data sharing. As shown in Figure 14, the throughput for the range and hash partitioning strategies does not decrease in this case because the buffer pool of the bottleneck processor does not thrash at high multiprogramming levels. Furthermore, a single processor never becomes a bottleneck permanently. Rather, when a bottleneck forms, it is very temporary and moves from one site to another. As evidenced by the disk utilization of the individual processors (Figure 15), the disk utilization of each processor is approximately the same at high multiprogramming levels (i.e., a processor never becomes a permanent bottleneck for the system).



Single Tuple Selection Using a Non-Clustered Index
High Degree of Data Sharing

Figure 14



High Degree of Data Sharing

Figure 15

Why is the bottleneck migratory for a high degree of data sharing and stationary for a low degree of data sharing? Two important factors are required in order for a bottleneck to form: 1) the multiprogramming level, and 2) the response time of that processor which has a larger than average fraction of the concurrently executing queries (say processor N) compared to the response time of the other processors in the system. The second variable is larger for a low degree of data sharing than for a high degree of data sharing since the buffer pool of processor N thrashes for a low degree of data sharing. Consequently, the queries on the lightly loaded processors complete executing much more rapidly than the queries executing on processor N. After a few query iterations, the random nature of the workload will direct these queries to processor N[12]. For a high degree of data sharing, the buffer pool of processor N never begins to thrash. Since the random number generator is not biased toward any certain processor, as soon as it directs a slightly larger percentage of queries to another processor (say processor M), the current

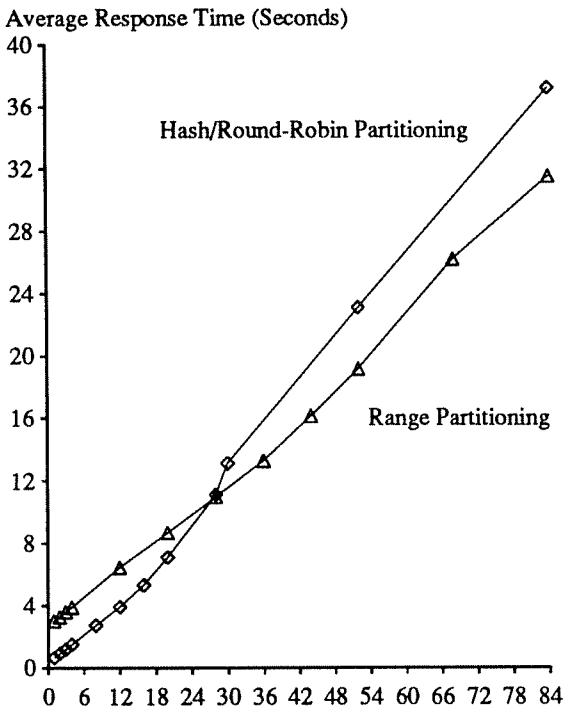
[12] Processor N can be thought of as a black hole that pulls the concurrently executing queries toward itself. By increasing the multiprogramming level, the mass of this black hole increases (the number of concurrently executing queries waiting to be executed by this processor).

bottleneck (i.e., processor N) unwinds while processor M becomes the new bottleneck for the system. Thus, the bottleneck is migratory.

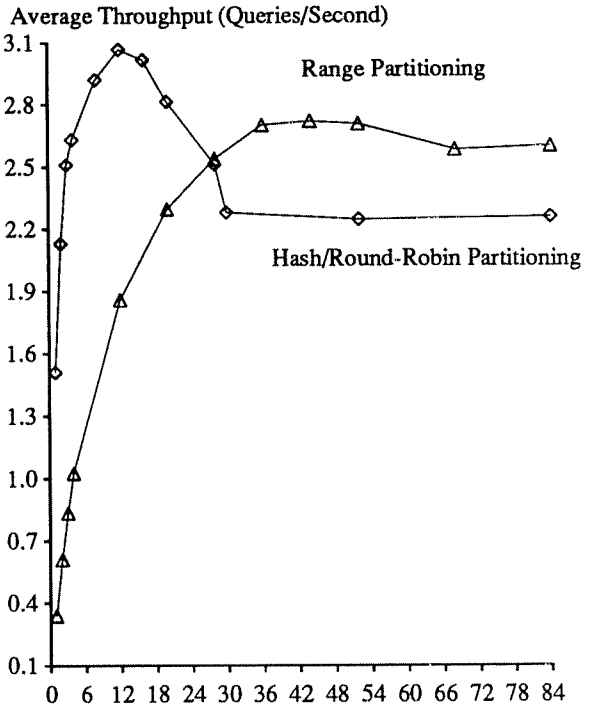
4.4. 0.125% Selection Using a Non-Clustered Index Access Method

A 0.125% selection query using a non-clustered index results in a high degree of random disk requests since the order of the values of the index records is not the same as the order of the data records in the relation. In Figures 16 and 17, the response time and throughput for the alternative partitioning strategies for this query with a high degree of data sharing is presented. The hash and round-robin partitioning strategies direct the query to all the sites, while the range partitioning strategy localizes the execution of the query to a single site[13].

For the hash and round-robin partitioning strategies, the throughput of the system begins to decrease from a multiprogramming level of 16. This query retrieves 125 tuples from the database and since 125 is not a multiply of



0.125% Selection Using a Non-Clustered Index
High Degree of Data Sharing
Figure 16

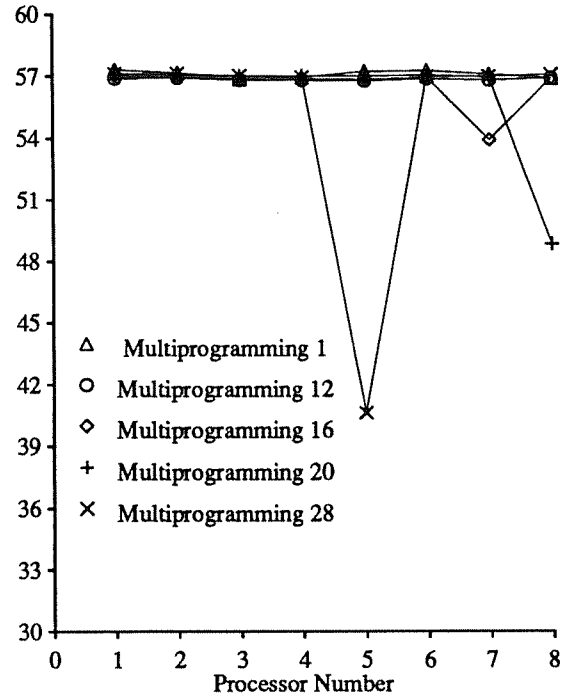


0.125% Selection Using a Non-Clustered Index
High Degree of Data Sharing
Figure 17

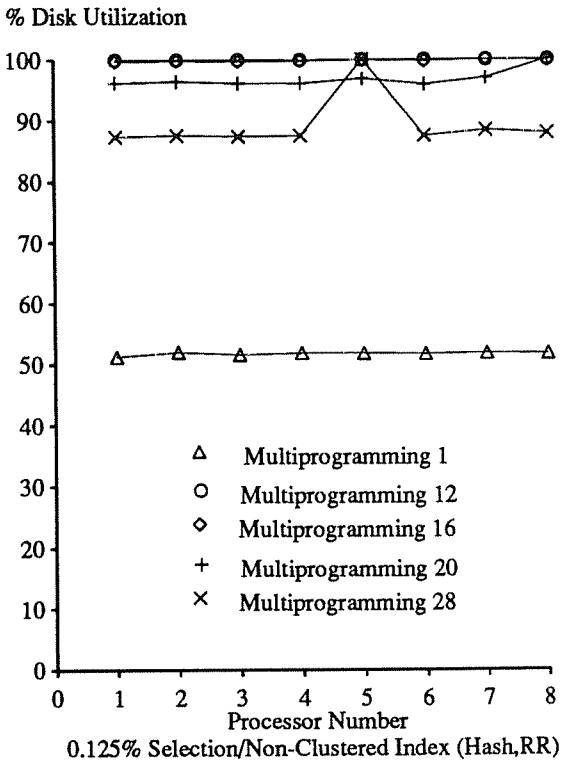
[13] Assuming that the selection predicate is applied to the same attribute as the one used to partition the relation.

8, neither the hash nor the round-robin partitioning strategy can distribute the work evenly among the eight processors. Therefore, some processors retrieve more tuples from their fragment of relation than the others. Since each additional tuple that is retrieved via a non-clustered index generally requires an additional disk I/O, the processing effort of some processors will be higher than the others. Consequently, at a multiprogramming level of twelve, the buffer pool of one of the highly loaded processors begins to thrash and the throughput starts falling. Furthermore, since a query cannot commit and return to its corresponding terminal unless it commits at all the sites participating in its execution, if a processor begins to thrash, the response time of this processor will determine the response time for each query. In addition, the processor with the thrashing behavior cannot stop thrashing, since as soon as it commits one query, a new query is submitted to it with the round-robin and hash partitioning strategies.

The following discussion provides support for this hypothesis. In Figure 18, the percentage of buffer pool hits for each processor at various multiprogramming levels is presented. The results reveal that at multiprogramming levels higher than 12, the buffer pool for one of the processors thrashes. In Figure 19, the disk utilization of each processor is presented. Note that the disk utilization of each processor in the multiprocessor configuration is



0.125% Selection/Non-Clustered Index (Hash,RR)
Figure 18

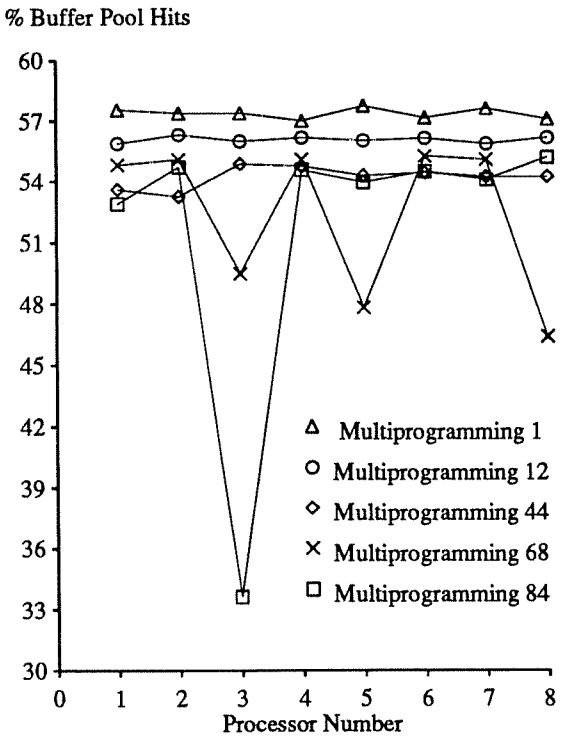


0.125% Selection/Non-Clustered Index (Hash,RR)
Figure 19

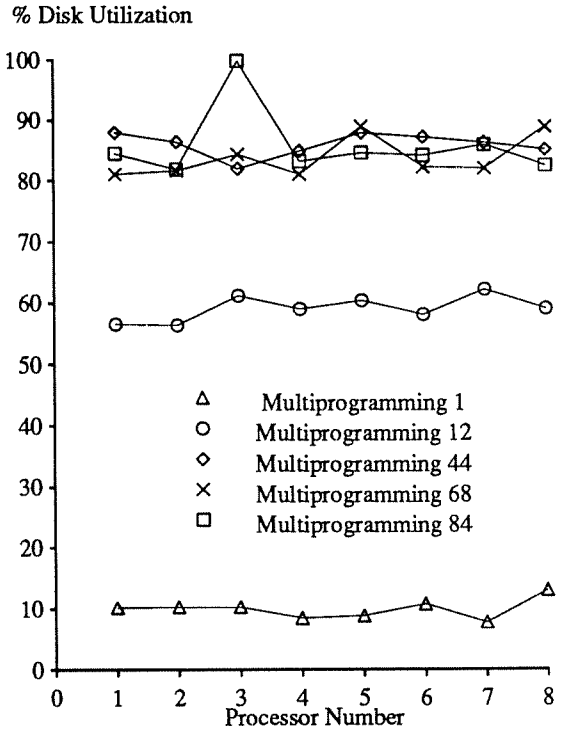
the same from a multiprogramming level of 1 to 12. At multiprogramming levels higher than 12, the processor with the thrashing behavior has a disk utilization of 100%. Furthermore, the disk utilization of the processors that do not thrash decreases since they remain idle waiting for a new query to be submitted.

For the range partitioning strategy, the throughput of the system increases from a multiprogramming level of 1 to 44, and begins to decrease at a multiprogramming level of 44. Since the range partitioning strategy localizes the execution of a query to a single processor, most of the concurrently executing queries do not interfere with each other. As shown in Figure 20, from a multiprogramming level of 1 to 44, the percentage of buffer pool hits is almost the same for each processor.

From a multiprogramming level of 44 to 68, one of the processors temporarily becomes a bottleneck due to the random nature of the workload. Consequently, the buffer pool of this processor thrashes, resulting in a higher response time for the queries executing at this site. At any instant in time, the random nature of the workload might direct more than $\frac{1}{8}$ of the concurrently executing queries to some other processor. The result is that the processor that is the current bottleneck unwinds, while a new processor becomes the bottleneck. In Figure 20 we can observe



0.125% Selection/Non-Clustered Index (Range)
Figure 20



0.125% Selection/Non-Clustered Index (Range)
Figure 21

the lower percentage of buffer pool hits for the processors that were temporarily the bottleneck for the system. Due to a lower percentage of buffer pool hits, the disk utilization for these bottleneck processors is also higher than for the other processors (see Figure 21). At multiprogramming levels above 68, one of the processors permanently becomes a bottleneck and, the disk drive of this processor becomes 100% utilized (Figure 21). Thus, the throughput and the utilization of other processors level off since the maximum rate at which the bottleneck processor can execute queries has been obtained.

The range partitioning strategy generally localizes the execution of the query to a single processor and, consequently, retrieves 1% of the tuples located at that site. At a multiprogramming level of one, the throughput of the hash and round-robin partitioning strategies is 4.46 times higher than that of the range partitioning strategy. While their throughput should actually be a factor of eight higher, the overhead associated with controlling the execution of these multisite queries accounts for the difference.

The highest throughput is obtained with the hash and round-robin partitioning strategies. While one processor eventually becomes a bottleneck for all three partitioning strategies, the hash and round-robin partitioning strategies obtain a 100% disk utilization at every processor before the formation of the bottleneck processor at a multiprogramming level of 12. The range partitioning strategy does not achieve a 100% disk utilization at each processor because the workload cannot guarantee an equal number of concurrently executing queries at each site.

When a processor becomes the bottleneck, the throughput decreases at a faster rate for the hash and round-robin partitioning strategies. On the other hand, with the range partitioning strategy, all queries are not affected by the thrashing behavior of the bottleneck processor. The response time of queries executing on the lightly loaded processors, to a certain degree, compensates for the high response time of queries at the bottleneck processor. The result is a gradual decrease in the throughput of the system. With the hash and round-robin partitioning strategies, however,the response time of every query is affected by the bottleneck processor (since the query is directed to every site). Consequently, the throughput of the system decreases at a very fast rate.

The throughput with the hash and round-robin partitioning strategy levels off at a multiprogramming level of 30 since the maximum number of concurrently executing queries permitted at each site is limited to 30. The throughput of the multiprocessor with the range partitioning strategy levels off at a multiprogramming level of 68 at the point that one of the processors becomes a permanent bottleneck. The throughput with the range partitioning strategy levels off at a higher level primarily because of the higher percentage of buffer pool hits and the lower response time of queries executing on the lightly loaded processors. It is important to note that the peak throughput

of the multiprocessor with the hash and round-robin partitioning strategies can be sustained by limiting the maximum number of concurrently executing queries to twelve at each processor.

5. Conclusions and Future Work

The results presented in this paper reveal that for a shared-nothing multiprocessor database machine, no partitioning strategy is superior under all circumstances. Rather, each partitioning strategy outperforms the others for certain query types. The major reason for this is that there exists a tradeoff between exploiting intra-query parallelism by distributing the work performed by a query across multiple processors and the overhead associated with controlling the execution of a multisite query. Localizing the execution of queries requiring minimal amount of resources, results in the best system response time and throughput since the overhead associated with controlling the execution of the query is either minimized or eliminated. On the other hand, for queries requiring more resources, certain tradeoffs are involved. For instance, with the 0.125% selection using a non-clustered index, the peak throughput of the system is higher with the hash and round-robin partitioning strategies. Conversely, at high multiprogramming levels (e.g., a multiprogramming level of 30), the range partitioning strategy outperforms the hash and round-robin partitioning strategy. In general, with access methods that result in the retrieval of only the relevant tuples from the disk, if the selectivity factor of the query is very low, it is advantageous to localize the execution of the query to a single processor. While the hash partitioning strategy localizes the execution of the exact match selection queries (e.g., single tuple retrieval using a non-clustered index), the range partitioning strategy attempts to localize the execution of all query types regardless of their selectivity factor. At the other end of the spectrum, the round-robin partitioning strategy directs a query to all the processors containing the fragments of the referenced relation.

For sequential scan queries, the best response time and throughput is achieved by localizing the execution of each query to a single processor. The system generally performs best when the query executes all by itself at a site and performs a series of sequential disk requests. By localizing the execution of the query to a single processor, there is a higher probability of maintaining the sequential nature of disk requests made by a query, free from interference of the other concurrently executing queries. Thus, for the sequential scan queries, the optimal partitioning strategy is the range partitioning strategy.

In the multiprocessor configuration with the range partitioning strategy, some processors are utilized more than others for the range queries whose selectivity factor generally overlap the range of values of two adjacent

processors (e.g., 10% selection query using a clustered index). The processors corresponding to the upper and lower limits of the range of the values of the partitioning attribute do not participate in the execution of as many queries as the other processors in the environment. This is a consequence of how the range partitioning strategy partitions a relation across the sites.

A simple solution to the above problem is to analyze the queries accessing a relation and determine the percentage of time the range of the queries overlap two or more sites. Using this information, more tuples can be assigned to the fragments corresponding to the upper and lower limits of the partitioning attribute value of the relation. By assigning more tuples to these fragments, the probability of access to the processors containing the upper and lower limits of the query is increased. This is not an optimal solution since it might degrade the performance of other query types accessing the relation. To illustrate, assume a query that retrieves one tuple via a sequential scan using a non-partitioning attribute. The response time of this query will be higher with the non-uniform distribution of tuples across the processors because the query is directed to all the processors and the processors corresponding to the upper and lower limits of the partitioning attribute value will have to process more tuples than the other processors.

For several query types, one of the processors became the bottleneck with each of the three partitioning strategies. It is important to note that in the presence of a bottleneck, the partitioning strategies that direct a query to a subset of sites will not obtain the maximum possible throughput. For example, the maximum expected throughput with the range and hash partitioning strategies for the single tuple retrieval using a non-clustered index with a high degree of data sharing is 200 queries per second since the maximum throughput of a single processor for this query type is 25 queries per second and there are 8 processors in the multiprocessor configuration. The peak throughput obtained is, however, only 170 queries per second. The major reason for not obtaining the full potential of the system is that at a high multiprogramming level, due to the random nature of the workload, the CPU of the bottleneck processor is 100% utilized and queries start to wait in the CPU queue of this processor while one or more additional processors remains idle, waiting for work.

In the future, we intend to extend the performance evaluation of the alternative partitioning strategies presented in this paper. First, we are planning to analyze the performance of the alternative partitioning strategies for update queries. The major reason for considering update queries is to determine the cost associated with preserving the partitioning constraint of the hash and range partitioning strategies. When a modify query updates the partitioning attribute value of a tuple, the tuple may have to be moved to another processor in order to preserve

the integrity of the partitioning constraint. We are interested in analyzing the impact of this on the performance of the multiprocessor with the hash and range partitioning strategies.

Second, the performance evaluation presented in this paper assumes a uniform distribution for the values of the partitioning attribute. In the future, we plan to analyze the impact of the skewed distribution of the partitioning attribute values on the performance of the alternative partitioning strategies. The major reason for performing this study is that the range and hash partitioning strategies cannot guarantee a uniform distribution of the tuples across the sites. However, the round-robin partitioning strategy can always guarantee a uniform distribution of tuples across the sites. We are interested in analyzing the significance of this aspect of the round-robin partitioning strategy, and its impact on the performance of a multiprocessor; especially compared to the other partitioning strategies.

6. References

[ALEX88] Alexander, W., et. al., "Process and Dataflow Control in Distributed Data-Intensive Systems," Proc. ACM SIGMOD Conf., Chicago, IL, June 1988.

[ANON84] Anon, et. al., "A Measure of Transaction Processing Power," Datamation 1984.

[BITT83] Bitton D., D.J. DeWitt, and C. Turbyfill, "Benchmarking Database Systems - A Systematic Approach," Proceedings of the 1983 Very Large Database Conference, October, 1983.

[BORA84] Boral, H, and DeWitt, D. J., "A Methodology for Database System Performance Evaluation," Proceedings of the 1984 SIGMOD Conference, Boston, MA, June, 1984.

[BORA88] Boral, H., "Parallelism and Data Management," Proceedings of the 3rd int'l. Conf. on Data and Knowledge Bases, Jerusalem, Israel, June 1988.

[CHOU85] Chou, H-T, DeWitt, D. J., Katz, R., and T. Klug, "Design and Implementation of the Wisconsin Storage System (WiSS)" Software Practices and Experience, Vol. 15, No. 10, October, 1985.

[DEWI86] DeWitt, D., Gerber, B., Graefe, G., Heytens, M., Kumar, K. and M. Muralikrishna, "GAMMA - A High Performance Dataflow Database Machine," Proceedings of the 1986 VLDB Conference, Japan, August 1986.

[DEWI88] Dewitt, D., Ghandeharizadeh, S., and D. Schneider, "A Perfromance Analysis of the Gamma Database Machine", Proceedings of the 1988 SIGMOD Conference, Chicago, IL, June 1988.

[GERB86] Gerber, R., "Dataflow Query Processing using Multiprocessor Hash-Partitioned Algorithms," PhD Thesis and Computer Sciences Technical Report #672, University of Wisconsin-Madison, October 1986.

[JARK84] Jarke, M. and J. Koch, "Query Optimization in Database System," ACM Computing Surveys, Vol. 16, No. 2, June, 1984.

[LORI88] Lorie, R., et al., "Adding Intra-Transaction Parallelism to an Existing DBMS: Early Experience" IBM Research Report RJ6165. Almaden Research Center, March 1988.

[PROT85] Proteon Associates, Operation and Maintenance Manual for the ProNet Model p8000, Waltham, Mass, 1985.

[RIES78] Ries, D. and R. Epstein, "Evaluation of Distribution Criteria for Distributed Database Systems," UCB/ERL Technical Report M78/22, UC Berkeley, May, 1978.

[SELI79] Selinger,P. G., et. al., "Access Path Selection in a Relational Database Management System," Proceedings of the 1979 SIGMOD Conference, Boston, MA., May 1979.

[STON86] Stonebraker, M., "The Case for Shared Nothing," Database Eng. 9, 1, March 1986.

[TAND88] Tandem Performance Group, "A Benchmark of Non-Stop SQL on the Debit Credit Transaction", Proceedings of the 1988 SIGMOD Conference, Chicago, IL., June 1988.

[TANE81] Tanenbaum, A. S., **Computer Networks**, Prentice-Hall, 1981.

[TERA85] Teradata Corp., *DBC/1012 Data Base Computer System Manual, Rel. 2.0*, Teradata Corp. Document No. C10-0001-02, November 1985.