A Methodology
for
Database System Performance Evaluation

Haran Boral
Computer Science Department
Technion - Israel Institute of Technology

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

This research was partially supported by the National Science Foundation under grant MCS82-01870 and the Department of Energy under contract #DE-AC02-81ER10920.

ABSTRACT

This paper presents a methodology for evaluating the performance of database management systems and database machines in a multiuser environment. Three main factors that affect transaction throughput in a multiuser environment are identified: multiprogramming level, degree of data sharing among simultaneously executing transactions, and transaction mix. We demonstrate that only four basic query types are needed to construct a benchmark that will evaluate the performance of a system under a wide variety of workloads. Finally, we present the results of applying our techniques to the Britton-Lee IDM 500 database machine.

1. Introduction

With the recent, but widespread, acceptance of the relational model as THE data model for the 1980s, we have seen the introduction of a large number of relational database systems and database machines. In fact, even for a given computer system there is a number of alternative relational products. Consider, for example, the Digital Equipment Corporation VAX 11 product line. The relational database systems/machines that are currently available for the VAX include the IDM 500 database machine, INGRES, INFORMEX, MISTRESS, ORACLE, RIM, and UNIFY.

A user faced with the task of selecting a system would ideally like to rate the systems on a "DBMS Richter Scale". We believe that the way to assign systems a rating is by quantitatively measuring their performance, that is by benchmarking them. Clearly the benchmarks must be designed to be general enough to model all of the systems' capabilities and their results used in a "fair" metric to assign a system its rating.

How do we measure systems? Although some systems, the IDM-500 for example, have sophisticated statistics gathering mechanisms and make the statistics gathered available to the user, we argue that such statistics should not be used in comparing different systems because of uniformity and portability problems. The thrust of the argument presented in this paper is that simple measures such as elapsed time, cpu time, and amount of I/O activity, which are generally available from all commercial systems, suffice as statistics for benchmarking database management systems/machines.

Developing a fair metric for comparing the performance of different systems is not, however, the thrust of this paper. There are several impediments to developing such a metric. First, the choice of a fair metric is a difficult problem for which many, application/institution dependent parameters must be considered. Clearly, factors such as response time, throughput, cost, and effect on the host computer are important parameters. However, the use of these parameters in a formula that would reduce the measurements of various systems to a point on the DBMS Richter scale is highly dependent on the user, the expected use of the system, etc. Second, comparing the performance of different systems is a hot potato which we want someone else to juggle for a while[1].

The following overview describes the approach that we have developed for benchmarking database systems and machines. The input data to the benchmark is a synthetic database and a set of queries generated by the user. Such databases can be easily generated by programs and examples can be found in [BITT83] and [BOGD83].

[1] Consider, for example, the controversy generated by the comparison of systems done in [BITT83].

The primary advantage of using a synthetic database over a "real" database is in the control exercised over the inputs and outputs during the benchmark runs. The benchmarking effort itself must proceed in two phases. In the first phase a large list of queries must be run in single user mode to obtain performance measures under optimal conditions. In the second phase multi-user benchmarks can be run to study the system's operation under more realistic loads.

We consider the first step to be crucial to the success of a benchmarking effort as it is likely to uncover various anomalies and give a picture of the resources required by the different queries. This in turn can then be used to design a multi-user benchmark that can be run in a reasonable amount of time (this will be described more clearly in the remainder of this paper).

An an illustration of the importance of the single-user benchmarks consider the join benchmark reported in [BITT83]. The version of INGRES from Relational Technology Inc. uses a sort-merge join algorithm while ORACLE uses a nested loops algorithm. Without an index on at least one joining attribute, ORACLE (Release 3.1 on a VAX 11/750) will require over 5 hours to join a 10,000 tuple relation with a 1,000 tuple relation (both relations had 182 byte tuples). INGRES (Release 2.0 on a VAX 11/750) can execute the same query in 2.6 minutes [BITT83]. Thus, if an application is characterized by a large percentage of ad-hoc joins on large relations, there is no point in performing multi-user benchmarks on any system that does not provide adequate performance in the single user case.

In this paper we extend our research on single user benchmark techniques and present a methodology for evaluating the multi-user performance of relational database management systems and relational database machines. Our proposed methodology is described in Section 2. In Sections 3 and 4, we present the design and the results of an experiment based on this methodology. Our conclusions and suggestions for future research directions are summarized in Section 5.

2. Evaluation Techniques and Strategies

2.1. Introduction

There appear to be three principal factors that affect database system performance in a multiuser environment:

- multiprogramming level

- query mix

- degree of data sharing

Each factor defines an axis in the performance space that can be varied independently. For example, holding the level of multiprogramming and degree of data sharing constant, while varying the mix of transactions allows one to observe the effect of transaction mix on system performance. In this section we elaborate on each of these factors and present the methodology that we have developed for multiuser benchmarks.

2.2. Multiprogramming Level

The multiprogramming level factor needs little or no explanation. In our experiments we have used the number of queries being executed concurrently as our measure of multiprogramming level. We have, however, relied on a broad interpretation of the word "executed". After submission, a query passes through a number of stages: parsing, access planning, execution, and finally transmission of the results to a user or an application program. A strict definition of multiprogramming level would be to include only those queries currently in the execution phase. Since controlling the system so that a constant number of queries were in this phase would be difficult or impossible, we choose to define multiprogramming level as being the number of queries in any phase of execution. We did, however, attempt to minimize the times associated with the other phases. The time for parsing and access planning was minimized by using precompiled queries and the time to transmit results back to the user was reduced by selecting queries that either only produced a few tuples or by reducing the number of attributes returned from each result tuple.

2.3. Degree of Data Sharing

Our notion of data sharing is based on the observation that in certain database applications, multiple queries may be accessing the same relations concurrently. Even if accesses to the same data pages are rare, there is a high probability that index pages will be repeatedly accessed by these applications. In this case, the design[2] of the buffer management system can have a significant effect on the multiuser performance as a buffer management system that makes intelligent replacement decisions will increase the frequency that the requested page will be found in the buffer pool. It is interesting to note that although an LRU replacement strategy provides poor[2] performance for many relational operators [STON81, SACC82], it appears that an LRU policy is best for managing the replacement of shared data pages. Thus to achieve optimal performance a database system may have to use different algorithms depending on whether or not a data page is being shared. We are currently investigating this issue.

To explore the effects of data sharing on system performance, we have used a scale of 0% to 100% as our measure of degree of data sharing. If the degree of data sharing is 0%, there is no data sharing amongst the concurrently executing queries and each query will reference its own partition of the database (or a separate database). Furthermore, all queries from the same application will reference relations in the same partition. Each partition must contain a complete set of relations and the number of partitions must equal the maximum multiprogramming level. If the degree of data sharing is set to 100%, all concurrently executing queries will reference the same partition of the database. In between values of 0% and 100%, we have defined the number of active partitions to be a function of the multiprogramming level. For example, with a multiprogramming level of 16 and a degree of data sharing of 50%, the active part of the database will consist of 8 partitions.

There appear to be two different ways of distributing queries across partitions when the degree of datasharing is between 0% and 100%. In the first approach queries are randomly distributed across all active partitions. Thus two successive queries from the same application will most likely reference distinct partitions. In the second approach, application programs are uniformly distributed across the available partitions. For example, with a multiprogramming level of 16 and a degree of data sharing of 50%, queries from the the first two application programs would always reference the first partition, queries from the next two application programs the second partition, etc. While this approach probably reflects reality more accurately, it appears that problems could arise when the multiprogramming level is not a multiple of the number of available partitions (e.g. a multiprogramming level of 15 and 8 partitions). Since we felt that this might lead to anomalous results (ie. the query running by itself in a partition might run slower than the queries sharing a partition), we elected to use the first approach in our experiments. An interesting extension of this research would be to explore whether there are, in fact, any differences between the two approaches.

2.4. Query Mix Selection

The hardest part of developing a methodology for multiuser benchmarks was devising a small set of representative queries. While developing our single user benchmark [BITT83], we began with over 120 queries.

[2] Unless the "hot-set" for each executing query can be kept in the buffer pool [SACC82]

Only after we analyzed the results did we realize that we needed just 21 queries to characterize the performance of a system. We found, for example, that the relative performance of two systems was the same for searches on both integer and string attributes. As another example, instead of having to test selection operations with 1%, 5%, 10%, 20%, and 50% selectivity factors, we found that 1% and 10% were adequate.

We began our development of a multiuser benchmark strategy by paring down the list of 21 queries to what was thought at that time to be a minimal list of 9 queries[3]. This paring was done in more-or-less a "seat of the pants" manner and without any assurances that the list was representative of "all types of queries." Furthermore, expanding the set back to 21 or 121 queries does not guarantee that the problem will resolved and makes running the benchmark prohibitively time consuming and interpreting the results difficult.

2.4.1. The Resource Utilization Approach to Query Mix Selection

Database queries consume two main system resources: CPU cycles and disk bandwidth.[4] CPU cycles are consumed both by the software that actually executes the query and by overhead functions performed by the database such as access path selection and buffer pool management. Additional CPU cycles are consumed by the operating system on behalf of the database system for initiating disk operations. Disk operations are performed in order to get the data required to answer a query into the buffer pool, to store the results of a query on disk, and as the result of swapping activity generated by the page replacement algorithm of the buffer manager.

We have partitioned consumption of each of these resources into two classifications: "low" or "high". Thus only four query types are needed as the basis for a multiuser benchmark:

Type I	- low CPU utilization, low disk utilization
Type II	- low CPU utilization, high disk utilization
Type III	- high CPU utilization, low disk utilization
Type IV	- high CPU utilization, high disk utilization

While we could have obviously divided the utilization of each resource into finer subdivisions (e.g. low, medium, and high), we feel that two values are all that are needed to bracket a database system's performance in a multiuser

[3] These queries were used by Bitton, Martinez, Turbyfill, and Wiley in a multiuser benchmark experiment at Los Alamos Laboratory during August 1983.

[4] See [STRA83] for another viewpoint on the design of multiuser benchmarks.

environment. These four query types should not, however, be used as the basis for a single user benchmark as a system might exhibit anomalous behavior on a particular query (see Section 1).

2.4.2. Application of the Resource Utilization Approach

Application of the resource utilization approach requires selecting a query that exhibits the behavior of each query type. We hypothesized that selecting 1 tuple via a clustered index or hashing would be a type I query and that selecting a small number (100 out of 10,000) tuples using a non-clustered index would be a type II query. We had no solid leads about what queries would correspond to query types III and IV.

To verify our hypotheses and choose candidates for query types III and IV, we took the 21 queries used for our single user benchmarks and ran them on the Britton-Lee IDM 500. Using "query-level" performance monitoring tools found in Release 25 software, we measured CPU and disk utilization for each query[5]. A summary of our results is presented in Table 1.

As expected, query number 1 was found to be a type I query and query number 3 was a type II query. For a type III query, we selected query number 8. While query number 10 is actually a better type III query, we were afraid that its extremely long running time would make our experiments unacceptably lengthy. Query 7 was the obvious choice for a type IV query. While the queries selected are each reasonable choices, they are not perfect. Ideally, each level of resource utilization (low or high) would consume exactly the same amount of the resource. For example, it would be nice to have the type I and type II queries utilize exactly the same amount of CPU time and for the type II and IV queries to perform exactly the same number of disk operations. In general, this is difficult to achieve as initiating a disk operation requires additional cpu time.

2.4.3. Portability Issues

Unfortunately the selection of type I, II, III, and IV queries is NOT independent of the database system. We hypothesize that queries 1 and 3 will be type I and II queries, respectively, on all relational database systems. Selection of type III and type IV queries, however, appears to be dependent on the algorithms used in the database system. For example, while an aggregate function query is a type IV query on the IDM500 (which uses a B-tree to group members of each partition), it would probably be a type II query on a system that implemented aggregate

[5] Queries were run using the ad-hoc query interface (idl).

Table 1
Resource Utilization Figures
for the IDM 500 Database Machine
with a Database Accelerator

Query #	Query	CPU Usage (seconds)	# of Disk Operations
1	Select 1 tuple from 10,000 using a clustered index	0.18	2-3
2	Select 100 tuples from 10,000 using a clustered index	0.56	11
3	Select 100 tuples from 10,000 using a non-clustered index	0.90	91
4	Select 1000 tuples from 10,000 using a clustered index	5.90	104
5	Select 1000 tuples from 10,000 using a non-clustered index	8.67	696
6	Scalar aggregate operation on 10,000 tuple relation	9.83	1,011
7	Aggregate function on 10,000 tuple relation (100 partitions)	35.62	1,008
8	Join 10,000 tuples with 1,000 tuples using a clustered index on join attribute of 10,000 tuple relation	18.96	206
9	Select 1000 tuples from 10,000 using a clustered index followed by a join with a 10,000 tuple relation using a clustered index	18.88	207
10	Select 1,000 tuples from 10,000 Select 1,000 tuples from 10,000 Join two 1,000 tuple relations to form a 1,000 tuple relation which is then joined with another 1,000 tuple relation	107.21	306

functions via hashing. Similarly, query 8 might become a type IV query on a system that executes joins exclusively via using sorting.

2.5. Performance Metric

We have used system throughput measured in queries-per-second as our principal performance metric. Where illustrative, response time has also been used as a performance indicator. The way in which system throughput and response time are measured is specified in Section 4.

3. Experiment Design

3.1. Description of the Test Database

The database used for our experiments is based on the synthetic database described in [BITT83]. Two basic relations were used to generate the database for our multiuser benchmarking experiments. We shall refer to them by the names of "oneKtup" and "tenKtup" as they contain, respectively, one and ten thousand tuples. Each tuple is 182 bytes long and consists of a number of integer and string attributes. The first attribute, "unique1", assumes unique values throughout the relation (and hence constitutes a key). For the "thoustup" relation, "unique1" assumes the values 0, 1, ..., 999. For the "tenthoustup" relation, the values of "unique1" are 0,1, ..., 9999. The second attribute, "unique2", has the same range of values as "unique1" but a random number generator was used to scramble the values of "unique1" and "unique2" when the relations were generated. The remaining integer attributes are named after the range of values each attribute assumes. That is, the "two", "ten", "twenty", "hundred",..., "tenthous" attributes assume, respectively, values uniformly distributed over the ranges (0,1), (0,1,...,9), (0,1,...,19), (0,1,...,99), ... ,(0,1,...,9999). Finally, each tuple contains three 52 byte string attributes.

The structure of these relations facilitate the formulation of a wide variety of queries. For example, the following two INGRES queries will both retrieve 100 tuples from the tenKtup relation:

```
range of x is tenKtup
retrieve (x.all) where 100 < x.unique2 < 201
        or
retrieve (x.all) where x.hundred = 35
```

Similarly, the following query computes an aggregate function query with exactly 100 partitions of 10 tuples each

```
range of x is oneKtup
retrieve (minvalue = min(t.tenthous by t.hundred ))
```

To permit benchmark runs with a multiprogramming level of 16 and no data sharing between simultaneously executing queries, the database was populated with 16 copies of both the oneKtup and tenKtup relations (each copy of the basic relations was given a unique name). Each relation was sorted on its unique2 attribute (thus leaving the relation unsorted on its unique1 attribute). For each of the tenKtup relations, a clustered index was constructed on the unique2 attribute and a non-clustered index was constructed on the unique1 attribute. No indices were constructed on the oneKtup relations.

3.2. Hardware Configuration

To evaluate the benchmarking techniques described in Section 2, an IDM 500 database machine was used as a test vehicle. The IDM hardware consists of a very high-speed bus and 6 different board types [UBEL81]:

(1) *The Database Processor*, which is responsible for controlling the other boards and implements most of the system functionality. The database processor is implemented using a Zilog Z8000 microprocessor chip. The processor runs a special-purpose operating system that tunes process and I/O management to the special needs of the database software.

(2) *The Database Accelerator (DAC)*, a specially designed ECL processor, that achieves very high speed by having a few well defined tasks microcoded.

(3) *A channel*, consisting of a microprocessor, memory, and hardware to implement 8 serial (rs232c) or one parallel (IEEE-488) interface. This channel implements a communication protocol with the host.

(4) *A memory timing and control board.* The memory is accessed in two modes: a byte-mode for the Database Processor and a faster word-mode for the Accelerator and the disk controller.

(5) *A memory board*, which provides for up to 6 megabytes of disk buffers and additional space for user processes (As a consequence of the 16 bit address space limitation of the Z8000, a maximum of 3 megabytes can be used for buffer space.)

(6) *A disk controller*, that can be expanded to interface with up to 32 gigabytes of disk storage.

The IDM 500 utilized for our benchmarks had five megabytes of memory, one disk controller, a 675 Mbyte CDC drive (40 tracks per cylinder), a parallel channel interface to the host processor (a PDP 11/70 running a variant of Unix 2.8), and a DAC that could be switched off and on remotely. Release 25 of the IDM 500 software was used for the benchmarks. Except where specified, one megabyte of memory was allocated for use as buffer space[6].

[6] To be exact, 471 2Kbyte buffers were used

3.3. Query Specifications

In Section 2, we described the four basic query types which form the basis of a multiuser benchmark. In this section we present the specific queries used for our experiments on the IDM 500. Instead of using the ad-hoc query interface (ie. idl), we used an embedded query facility in which the query is encapsulated in a program that runs on the host computer. Furthermore, except where otherwise noted, the stored query facility provided by the IDM 500 was always used. This facility provides a mechanism by which a query can be precompiled and stored in the database [IDM500].

The queries used for Query Types I to IV are shown below in Figures 1 to 4. There are several important points to observe about these queries. First, for query types I and II, the selection criterion is randomly generated for each query executed. This insures that access to the data and index pages pages will be randomly distributed across all data and index pages. Such an approach is critical in a multiuser benchmark. This step was not, however, necessary for query types III and IV. In the case of query type IV, each time the query is executed the entire tenKtup relation (which is twice as large as the buffer pool) will be accessed. The situation for query type III is slightly more complicated. Each time query type III is executed, the entire oneKtup relation will be accessed plus approximately 10% of index and data pages of the tenKtup relation. However, the 10% of the tenKtup relation accessed will always be the same[7]. When multiple instances of query type III are executed concurrently with a high degree of data sharing, this 10% will most likely become resident in main memory as it occupies approximately 200K bytes (the buffer pool was approximately 1 megabyte in size). While one might interpret this as a potential problem, one should remember that the objective of the type III query was high cpu activity and low i/o activity. Thus, having part of the database resident in the buffer pool, actually helps achieve this objective[8].

A second point about this set of queries, is that each query returns only 2-4 attributes of each qualifying tuple to the host. Finally, note that in order to maximize the performance of the system, no operations are performed on the data that is returned to the host.

[7] Those tuples in tenKtup whose unique2D attribute falls in the range of 0 to 999.

[8] While one could apply the same argument to query type I, we decided that this doing so would result in artificially high transaction rates.

```
qryI()  /* Select 1 tuple using clustered index */
{
    int a, b, value;   long randomnumber;

    randomnumber = r0random(&seed);   /* select a random key value  */
    value = (randomnumber % 10000);   /* between 0 and 9999 */

    range of x is tenKtup
    retrieve (a = x.unique1D, b = x.unique2D) where x.unique2D = value
}
```

Figure 1
Query Type I

```
qryII()   /* Select 100 tuples out of 10,000  */
          /* using a non-clustered index.  */
{
    int lowervalue,uppervalue, a, b;

    randomnumber = r0random(&seed);  /* select a lower range value */
    lowervalue = (randomnumber % 9901);  /* between 0 and 9900 */
    uppervalue = lowervalue + 100;

    range of x is tenKtup
    retrieve (a = x.unique1D, b = x.unique2D)
    where (x.unique1D >= lowervalue) and (x.unique1D < uppervalue)
}
```

Figure 2
Query Type II

```
qryIII()   /* Join using clustered index on join attribute unique2D */
{             /* Query produces 1000 tuples */
    int a,b,c,d;

     range of t is tenKtup
     range of w is oneKtup
     retrieve (a = t.unique1D, b = t.unique2D, c = w.unique1A,
        d = w.unique2A)  where t.unique2D = w.unique2A
}
```

Figure 3
Query Type III

```
qryIV()   /* Aggregate function min with 100 partitions */
{
    int xmin;

     range of x is tenKtup
     retrieve (min = min(x.twothousD by x.hundredD))
}
```

Figure 4
Query Type IV

3.4. Multiuser Benchmark Program Design

To simplify the task of performing a wide variety of experiments, a program for this purpose was designed and implemented. The structure of this program is shown in Figure 5.

```
For each iteration:
   Select a query type
   Select a partition of the database
   Execute Query
```

Structure of Benchmark Program
Figure 5

The number of copies of this program executed concurrently corresponds to the multiprogramming level of an experiment. The parameters to each program instance include the multiprogramming level, the number of iterations to be executed, the percentage of each query type to be executed, and the degree of data sharing. In general, all instances executed concurrently receive the same parameter values. During each iteration, one query is executed. As the first step of an iteration, a query type is randomly[9] selected according to the distribution of query types specified by the input parameters. Once a query type has been selected, a partition of the database is chosen. During an experiment with no data sharing, each instance of the program will use a separate partition of the database. For an experiment with 100% data sharing, all programs will select the same partition. For levels of data sharing between 0% and 100%, a partition is randomly selected by generating a random number between one and $\lceil MPL * DS \rceil$ where MPL is the current multiprogramming level. Finally, an instance of the chosen query type is executed using the selected database partition. To gather timing information for later analysis, the time-of-day clock is read and recorded immediately before and after each query is executed.

We had initially hoped to let each program instance terminate execution after reaching steady state. This approach proved difficult to implement. Instead, each concurrent program was run for a fixed number of iterations. The number of iterations performed was varied according to query type. For type I queries, 250 iterations were performed. Ten iterations were used for type IV queries. In general, the goal was to run each program long enough to enable the system, as a whole, to reach steady state. In several instances, subsequent analysis of the output data revealed that the system had never reached steady state - forcing us to rerun the experiment.

[9] To insure randomness and repeatability of experiments, each program instance is provided with its own random number seed.

4. Experimental Results and Analysis

4.1. Introduction

In this section we describe the experiments we conducted using our multiprogramming benchmark and analyze the results obtained. While a wide variety of interesting experiments are possible using this benchmark, we present the results of those that we found most interesting.

The first experiments explore system performance as a function of multiprogramming level and degree of data sharing for each of the four query types. Our results demonstrate that these four query types are sufficient for evaluating the multiuser performance of a system.

The next set of experiments conducted were designed to determine the effect on system performance of a number of different factors. We began by considering two different mixes of concurrently executing queries. The first mix consisted of query types I through IV. The second mix consisted of type I queries and queries that update single tuples. As the next experiment we explored the effect of precompiled (stored) queries on system throughput. Third, we examined the performance of the IDM500 with and without a database accelerator. Finally, we looked at system performance while varying the size of the buffer pool.

4.2. Measurement of System Throughput and Response Time

As discussed in Section 3, the time-of-day clock is read and recorded immediately before and after each query is executed by each concurrently executing program. When all iterations of an experiment are concluded, each program dumps its measurements to an output file. Thus, with a multiprogramming level of eight, eight output files are produced.

To determine the transaction throughput and/or response time for an experiment, the output files are analyzed in the following manner. Let MPL donate the multiprogramming level of an experiment and N the number of iterations each concurrent program executes. Let $S_{i,j}$ and $E_{i,j}$ represent, respectively, the starting and ending times of the jth query of the ith concurrent program for $1 \le i \le MPL$ and $1 \le j \le N$. The output data is first analyzed to find:

$$T_{last-to-start} = \max \{ S_{i,1},\ 1 \le i \le MPL \}$$

$T_{last-to-start}$ is the time that the multiprogramming level first reaches the level specified for the experiment. The next step is to find:

$$T_{first-to-finish} = \min \{E_{i,N},\ 1 \le i \le MPL\ \}$$

After this point the multiprogramming level is below the desired level. Once $T_{last-to-start}$ and $T_{first-to-finish}$ have been determined, the number of queries that were completely executed within the "last-to-start" and "first-to-finish" interval is calculated (for the jth query of the ith program to be included in this total $S_{i,j} \ge T_{last-to-start}$ and $E_{i,j} \le T_{first-to-finish}$.) Transaction throughput is defined to be this number of queries divided by the length of the interval in seconds. An average response time is calculated by dividing the sum of the execution times for the set of queries completely executed within the "last-to-start" and "first-to-finish" interval by the number of queries. While this approach eliminates termination effects, the system may not have yet reached steady state at $T_{last-to-start}$. Determining exactly when a database system reaches steady state appears to be an interesting area for future research.

4.3. System Performance for Query Types I, II, III, and IV

The first tests conducted were to measure system throughput as a function of multiprogramming level and degree of data sharing. The multiprogramming level was varied from 1 to 16 in increments of 1. Three levels of data sharing were tested: 0%, 50%, and 100%. The results of these experiments are shown in Figures 6a, 7a, 8a, and 9a for query types I, II, III, and IV respectively.

One can make a number of general comments about these results. First, we claim that the four query types are sufficient for evaluating the multiuser performance of a database system (ignoring, for a moment, transactions that update the database). The number of queries executed per second ranged from a maximum of almost 25 per second for query type I to a minimum of 0.025 per second for query type IV (a range of three orders of magnitude). Second, the extent to which concurrently executing queries reference the same portion of the database does indeed have an effect on performance. For query type II there was a factor of 2 difference in throughput between 0% data sharing and 100% data sharing. It is important to note, however, that even with 100% data sharing the relation being accessed is twice the size of the available buffer pool. Thus, the increase in performance is due almost exclusively to data sharing among the concurrently executing queries. For query type III, the difference was a factor of 3. The degree of data sharing, however, had little or no effect on query types I and IV. This seems reasonable for query type I in which single records are retrieved. We cannot explain, however, why query type I exhibits a lower throughput rate with 50% data sharing than with 0% data sharing. Degree of data sharing has a very minimal effect on the performance of query type IV as this query makes a sequential scan through a 2 million byte relation. Since the buffer pool has only a million bytes, even with 100% data sharing the concurrently executing queries will almost certainly be trying to simultaneously load different portions of the relation. The fact that the performance of the IDM 500 degrades only marginally for query type IV as the multiprogramming level is increased indicates that the replacement strategy for the buffer pool works properly.

The degradation in performance of query type III is an interesting illustration of the effect of data sharing. As discussed in Section 3.3, each time a type III query is executed, the entire outer relation (100 pages) will be accessed plus the same 100 pages pages of the inner relation. With 100% data sharing, both sets of pages will remain memory resident. With a 500 page buffer pool and 0% data sharing, the pages of the inner relations can be kept in memory until the multiprogramming level reaches 5. With 50% data sharing, the drop in performance occurs (as it should) at a multiprogramming level of 9-10. It is interesting to wonder what differences in throughput would occur if a FIFO or random buffer pool replacement policy were used instead of LRU.

In Figures 6b, 7b, 8b, and 9b we have plotted the average response time as a function of the multiprogramming level for query types I, II, III, and IV, respectively. These figures provide another interesting view of the performance of the IDM 500. Consider, for example, Figure 6b. Although the response time remains constant only through a multiprogramming level of 3, a 0.6 second response time at a multiprogramming level of 16 is very impressive. Figures 7b and 8b provide further verification of the effects of data sharing and in Figure 8b the same transitions found in Figure 8a can be observed. Finally, it is clear from Figure 9b that one instance of query type IV is sufficient to saturate the system.

4.4. Two Experiments Involving a Mix of Query Types

In addition to considering all one type of query we also evaluated the performance of the system for two different mixes of concurrently executing queries. The first mix (Figure 10) consisted of 70% type I queries, 10% type II queries, 10% type III queries, and 10% type IV queries. This query mix was intended to reflect the mix of query types one might find in a "real" environment. Performance of the system is relatively constant with 100% data sharing but shows a moderate decline with increases in the multiprogramming level for data sharing levels of 0% and 100%. We were not able to isolate the cause for this degradation in performance.

The second mix consisted of type I queries and queries which update single tuples. The tuples to be updated were randomly selected from the ten thousand tuple relation and were accessed through their key attribute. Updating a tuple involved replacing the value of the key attribute with itself. Since the IDM 500 does not check to see whether an update really changes anything, this trick allowed us to execute update operations on the database without actually changing it. Replacing a key attribute causes the IDM 500 to first delete the tuple from the clustered index and then reinsert the modified tuple.

For this experiment we fixed the multiprogramming level at 16 and varied the query mix from 100% update queries and 0% type I queries to 0% update queries and 100% type I queries. The results are shown in Figure 11. One can make a number of interesting observations from these results. First, the fact that system throughput was the same for both 0% and 100% data sharing indicates that although conflicts did occur[10], they had an insignificant effect on performance. Second, as expected, updates are more than twice as expensive as retrievals due to the cost of maintaining the primary index and writing recovery information on the log.

4.5. Impact of Query Precompilation

The next experiment we conducted was to examine the effect of precompiled queries (stored commands) on system throughput. Since the impact of this feature is minimal for long queries, we evaluated the feature only for type I queries by rerunning the experiments on query type I without using stored command facility. A data sharing level of 100% was used to minimize the number of disk operations performed. Figure 12 contains the results of this experiment. The factor of 3 improvement in performance obtained clearly indicates the value of precompiled queries in enhancing system performance.

4.6. Performance Enhancement Provided by the Database Accelerator

The database accelerator (DAC) is a customized ECL processor that is microcoded to provided enhanced CPU performance for the most frequently executed database operations. In the single-user, static benchmarks described in [BITT83], the DAC provided a speedup of 1.14 for query type I and 1.28 for query type III. We were interested in seeing whether the DAC would provide a higher level of enhancement in a multiuser environment. Figures 13 and 14 present the results of our tests on query types I and III, respectively. These query types were selected as both perform a minimum number of disk operations (hence emphasizing CPU utilization). Disk traffic was also minimized by using 100% data sharing for both tests. The results indicate that the DAC indeed provides additional enhancement in a multiuser environment. For query type I, the maximum speedup is 1.4 (at multiprogramming level 10). For query type III, the dynamic speedup achieved is 1.71.

[10] Conflicts were observed by examining the statistics relation maintained by the IDM 500.

4.7. Effect of Buffer Size

The final tests were designed to explore the effects of buffer size on system performance. Query type II (selection through a non-clustered index) was selected for this test as it exhibits a high disk to CPU utilization ratio. The multiprogramming level was fixed at 16. The results are displayed in Figure 15. While somewhat surprising, the results for 0% data sharing are easily explained. With 100 pages of buffer space, the index pages for each of the 16 relations will fit in memory. During each iteration, each of the concurrently executing queries randomly selects 100 tuples to retrieve. Since the retrieval is done through a non-clustered index, each tuple will most likely be stored on a different data page. Furthermore, during the next iteration a different set of 100 pages will be referenced. We would expect that the curve would remain flat until the buffer pool became large enough to hold a significant fraction of all 16 relations being accessed (16,000 pages). The test with 100% data sharing illustrates the interaction between the size of the buffer pool and the effect of data sharing. A more detailed analysis of this relationship appears to warrant further investigation.

5. Conclusions and Future Research

In this paper we have presented a methodology for evaluating the performance of database management systems and database machines in a multiuser environment. We have identified three main factors that affect transaction throughput in a multiuser environment: multiprogramming level, degree of data sharing among simultaneously executing transactions, and transaction mix. Furthermore, by considering resource utilization as the main criteria for choosing a representative transaction mix, we are able to construct a benchmark that will evaluate the performance of a system under a wide variety of workloads with only four basic query types. As an example, we have applied our methodology to the Britton-Lee IDM 500 database machine. While multiuser benchmarks are critical for evaluating the performance of a database system, we feel that single user benchmarks such as presented in [BITT83] and [BODG83, STRA83] are still essential for uncovering anomalies in a particular system’s performance.

There are a number of interesting areas that warrant further investigation. A more careful exploration of the relationship of data sharing and buffer management should yield improved buffer management algorithms in a multiuser environment. Additional work is also warranted on updating transactions. In this paper we have explored only the effect of single tuple updates through a clustered index. We have not examined, for example, whether updates on non-indexed attributes or attributes with secondary indices have different effects on throughput.

In addition, we have not considered "bulk" update operations. It would also be interesting to attempt to verify some of the analytically derived locking granularity results of Ries and Stonebraker [RIES77, RIES79].

Another area worthy of investigation is the effect of back-end communications bandwidth on system performance[11]. On possible experiment would be to fix the multiprogramming level, the degree of data sharing, and the query mix and vary the number of bytes returned by each query. Finally, techniques need to be developed for benchmarking distributed database systems. In a distributed database environment a query consumes communication resources in addition to CPU and disk resources. Furthermore, a query may consume both local and remote CPU and disk resources.

6. References

[BOGD83] Bogdanowicz, R., Crocker, M., Hsiao, D., Ryder, C., Stone, V., and P. Strawser, "Experiments in Benchmarking Relational Database Machines," *Database Machines*, Springer-Verlag, 1983.

[BITT83] Bitton, D., DeWitt, D. J., and C. Turbyfil, "Benchmarking Database Systems: A Systematic Approach," Computer Sciences Department Technical Report #526, Computer Sciences Department, University of Wisconsin, December 1983. This is a revised and expanded version of the paper that appeared under the same title in the Proceedings of the 1983 Very Large Database Conference, October, 1983.

[EPST80] Epstein, R. and P. Hawthorn, "Design Decisions for the Intelligent Database Machine," Proceedings of the 1980 National Computer Conference, pp. 237-241.

[IDM500] IDM 500 Reference Manual, Britton-Lee Inc., Los Gatos, California.

[RIES77] Ries, D., and M. Stonebraker, "Effects of Locking Granularity on Database Management System Performance," ACM Transactions on Database Systems, Vol. 2, No. 3, September 1977.

[RIES79] Ries, D., and M. Stonebraker, "Locking Granularity Revisited," ACM Transactions on Database Systems, Vol. 4, No. 2, June 1979.

[SACC82] Sacco, G. M. and M. Schkolnick, "A Mechanism for Managing the Buffer Pool in a Relational Database System Using the Hot Set Model," Proceedings of the 1982 Very Large Database Conference, (September 1982).

[STRA83] Strawser, Paula, "A Methodology for Benchmarking Relational Database Machines," Ph.D. Dissertation, Ohio State University, December 1983.

[STON81] Stonebraker, M., "Operating System Support for Database Management," Communications of the ACM, Vo. 24, No. 7, July 1981, pp. 412-418.

[UBEL81] Ubell, M., "The Intelligent Database Machine," Database Engineering, Vol. 4, No. 2, Dec. 1981, pp. 28-30.

[11] Suggested by Paula Strawser