BENCHMARKING DATABASE SYSTEMS
A SYSTEMATIC APPROACH

Dina Bitton
David J. DeWitt
Carolyn Turbyfill

Computer Sciences Department
University of Wisconsin-Madison

ABSTRACT

This paper describes a customized database and a comprehensive set of queries that can be used for systematic benchmarking of relational database systems. Designing this database and a set of carefully tuned benchmarks represents a first attempt in developing a scientific methodology for performance evaluation of database management systems. We have used this database to perform a comparative evaluation of the "university" and "commercial" versions of the INGRES database system, on one hand, and the IDM 500 database machine, on the other hand. We present a subset of our measurements (for the single user case only), that constitute a preliminary performance evaluation of these systems.

1. Introduction

During the past decade a large number of database machines encompassing a wide variety of architectures and possessing a range of different characteristics have been proposed to enhance the performance of database management systems. Today, it is not clear that specialized architectures offer any significant performance advantages over general purpose computers.[1] This paper is a first attempt to provide an answer to this question by presenting the results of benchmarks run on several conventional database management systems and on one database machine. More specifically we have measured and compared the performance of the Britton-Lee IDM/500 machine with and without a database accelerator (DAC) [IDM500, EPST80, UBEL81] to the "commercial" and "university" versions of the INGRES database system [STON76, STON80].[2]

Since database machines have already been an active field of research for an entire decade, and a few machines have now been implemented, we feel that the time has come for measuring the actual performance enhancement that can be expected from using special purpose hardware and software for database management. When database machines designs such as CASSM [SU75], RAP [OZKA75, OZKA77], DBC [BANE78], DIRECT [DEWI79] were proposed, the future of database machines looked bright. It seemed that both successful research efforts and advances in hardware technology would lead to the widespread use of commercial database machines. However, while most projects appeared promising initially, it is only very recently that the first of these special purpose computers are becoming commercially available. The ICL CAFS machine [MCGR76, BABB79] has been shipped in small quantities. The Britton-Lee IDM (Intelligent Database Machine) appears to be the first database machine to reach the market place in large volumes.[3] Despite these exceptions, the overwhelming evidence is that the majority of the database machine designs proposed will never be more than laboratory toys and, in most cases, will never even leave their promising paper status.

While the first database machines are being marketed, better database management systems are now being offered that do not rely on special hardware for enhancing performance. For example, several years of experience with the INGRES database management system has led to the development of a commercial version of this system on several general purpose computers. This development, added to the apparent slowdown of research on database machines has provided a strong motivation for the experiments described in this paper.

Previous performance evaluation studies of database machines [HAWT82, DEWI81] have given us some insight on the problems that various database machine architectures face. These studies were, however, based on simplified analytical models. We feel that it is

[1] Especially in the face of limited I/O bandwidth. See [BORA83] for a discussion of the impact of trends in mass storage technology on the future of highly parallel database machines.

[2] Space limitations prevented us from including a presentation and evaluation of the database machine DIRECT [DEWI79, BORA82]. A complete version of this paper is available as a technical report from the Computer Sciences Department.

[3] As of April 1983, approximately two hundred IDM 500 and IDM 200 database machines had been shipped.

necessary to extend them by empirical measurements. These measurements, in addition to providing a comparison of the different systems available, will provide a means of evaluating the accuracy of the performance evaluation tools that we have been using to compare alternative database machine architectures [DEWI81], [HAWT82]. Our results also provide some insight into to the extent to which a "conventional" operating system "gets in the way of" a database management system [STON81] as the IDM 500 without a database accelerator really represents the performance of a database management system running on a conventional processor (a ZILOG Z8000) but WITHOUT a general purpose operating system getting in the way.

In addition to looking at the relative performance of a database machine and two versions of a conventional database management system, this paper also describes a customized database and a comprehensive set of queries that can be used for systematic benchmarking of relational database systems. Designing this database and a set of carefully tuned benchmarks represents a first attempt in developing a scientific methodology for performance evaluation of database management systems. In addition to providing a mechanism for comparing the performance of different systems, we feel that such a benchmark will become a useful tool for database system implementors to use for evaluating new algorithms and query optimizers.

The paper is organized as follows. In Section 2, we provide a brief description of the four systems. The hardware configuration is described in some detail for each of the machines, and the basic software structure is outlined. In Section 3, we explain how we designed our experiments, and motivate the framework of our benchmarks. In Section 4 we present and analyze the results of our comparisons. Finally, in Section 5 we summarize our conclusions and indicate how we plan to extend the present study.

2. Description of the Three Systems Evaluated

In this section, we describe the basic architecture and software structure of the three systems compared: the INGRES database management system (in two different configurations: the "university" version on a VAX 11/750 running 4.1 Berkeley Unix and the "commercial" version on a VAX 11/750 running the VMS operating system) and an IDM 500 connected to a PDP 11/70 host. Detailed descriptions of the design and implementation stages for the research project that led to the current version of INGRES can be found in several published papers referenced throughout this section. On the other hand, there is less written documentation on the development of the IDM 500 as it is a relatively recent system that, from the start, was intended to be a commercial product. However, in Section 2.2 we have tried to present a complete enough description of the design of the IDM 500 to provide the reader with enough background for comparing this system with the other two.

The hardware configurations that we have used to run our benchmarks have been made as fair as possible. In particular, each system was evaluated using disk drives with similar characteristics, similar disk controller interfaces, and, where possible, equivalent amounts of buffer space for use by the database system.

2.1. The Two INGRES Systems

The INGRES project began in 1973, at the University of California at Berkeley. INGRES was first implemented on the top of the Unix operating system, and since 1976 has been operational as a multiuser DBMS. Since the original version, the system has been improved and enhanced in a number of ways to improve usability and performance. Recently, a commercial version of INGRES has been completed, which is now reaching the market place.

In this section, we will shortly summarize the main features of university-INGRES, and describe the system configuration on which our benchmarks have been run. We will then describe the enhancements added to commercial-INGRES.

2.1.1. University-INGRES

The version of university-INGRES tested was that delivered on the Berkeley 4.1 distribution tape. This version of INGRES runs as two Unix processes: a monitor process for interacting with the user and a second process which is responsible for performing all operations on the database. Query execution is done in an interpretative fashion.

The VAX 11/750 on which university-INGRES was tested has 2 megabytes of memory and four disk drives connected to the VAX with a System Industries 9900 controller using a CMI interface. The operating system run was Berkeley 4.1 Unix which utilizes 1,024 byte data pages. The database was stored on a Fujistu Eagle disk drive (474 Megabytes). Immediately before the database was loaded, a new Unix file system was constructed on this drive thus maximizing the probability that two logically adjacent blocks would be physically adjacent on the disk (an atypical situation for a typically scrambled Unix file system).

University INGRES does no buffer management of its own relying instead on the Unix operating system to buffer database pages. As discussed in [STON81], buffer management strategies that are good at managing virtual memory pages are frequently very poor at choosing the "right" page to eject in a database environment. In particular for repeated access to the inner relation of a join, LRU is absolutely the worst algorithm to use for selecting pages of the inner relation to eject. This is exactly the algorithm used by Berkeley 4.1 Unix.

2.1.2. Commercial-INGRES

Commercial-INGRES also runs as two processes. The VAX 11/750 on which Commercial-INGRES was evaluated had 6 megabytes of memory, a RM80 attached to the processor with a mass-bus interface, and a Fujistu Eagle drive connected to the processor through the CMI bus with an Emulex SC750 controller. The INGRES software was stored on the RM80 drive and the test database was stored on the Fujistu drive. The operating system used was VMS release 3. VMS provides an extent based file system (i.e. logically adjacent blocks are almost always physically adjacent on the disk). For our test 800K bytes of main memory was allocated for buffer space and 200K bytes were allocated for sort space. Buffer management was done using a random replacement policy.

Version 2.0 of the commercial version of INGRES under the VMS operating system includes a number of performance enhancements not present in the university version. While a number of routines have been rewritten for improved performance, the major changes have occurred in the following areas:

(1) new query optimizer - develops a complete query execution plan before execution of the query is initiated

(2) sort-merge join strategies

(3) 2K byte data pages (versus 1K byte pages in 4.1 Unix)

(4) caching of query trees - permits repetitive queries to be reexecuted without re-parsing

(5) buffer management under the control of the database system - permits implementation of replacement strategies that are tuned to enhance database operations and sharing of database pages by multiple transactions

2.2. The IDM/500 Database Machine

The Intelligent Database Machine appears to be the first widely used commercial database machine. It was developed by Britton-Lee, Inc., and the first machines were marketed in 1981. 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. It uses a standard 16-bit microprocessor chip (Zilog Z8000). The processor runs a special-purpose operating system, that schedules disk accesses intelligently. Unlike most operating systems, the IDM operating system tunes process and I/O management to the special needs of the database software.

(2) *The Database Accelerator (DAC)*, is a specially designed ECL processor, that achieves very high speed by having a few well defined tasks microcoded. The IDM may be configured with or without the Accelerator (depending on the cost and performance desired). When the Accelerator is not physically available, it is emulated by the Database Processor.

(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. It buffers commands coming from the host to the IDM, or result data returning from the IDM to 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 two megabytes of memory, one disk controller, a 675 Mbyte CDC drive, 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. One megabyte of memory was allocated for use as buffer space.

While the CDC disk drive has more tracks per cylinder than the Fujistu Eagle (40 vs. 20), its track-to-track seek time and transfer rate are slower than that of the Eagle. We calculated that the time to read a 10,000 tuple relation (182 bytes/tuple) would be 9.3 seconds on the CDC drive and 8.0 seconds on the Fujistu drive. Thus, the results presented in Section 4, may be slightly biased against the IDM 500. It is important to realize, however, that the degree of this bias is highly dependent on the query type, the availability of suitable indices, and whether the performance of the IDM is CPU limited or I/O limited.

3. Benchmark Description

The starting point for our experiments was the design of a database. This database had to be customized for extensive benchmarking. Previous efforts in this area have generally been relatively unscientific. In particular, the benchmarks that we are aware of involve using an existing database system and run a rather restricted set of queries. In some cases, the database (e.g. the supplier-parts database that INGRES users are familiar with) would be so small that the results of the benchmarks would not provide any insight about "real world" database management systems. In other cases, although the size of the database was large enough, the data values would not provide the flexibility required for systematic benchmarking. To be more specific, there would be no way to generate a wide range of retrieval or update queries, and control the result of these queries. For example, the existing data would not allow one to specify a selection query that selects 10% or 50% of the source relation tuples, or a query that retrieves precisely 1,000 tuples. For queries involving joins, it is even harder to model selectivity factors and build queries that produce a result relation of a certain size.

An additional shortcoming of empirical data (versus "synthetic" data) is that one has to deal with very large amounts of data before it can be safely assumed that the data values are randomly distributed. By building our own database, we were able to use random number generators to obtain uniformly distributed attribute values, and yet keep the relation sizes tractable.

In this section, we describe the guidelines along which we have designed our benchmark. Our design effort has resulted into a simple but carefully tuned database and a comprehensive set of queries. In Section 3.1, we describe the structure of the relations in our database. Section 3.2 contains a description of the queries that were run in our benchmarks. In both sections, we have made our descriptions as explicit as possible, while explaining the design principles that motivated the choice of a particular attribute value or a specific query. In Section 3.3, we describe the experiment itself: the environment in which the queries were run and the performance parameters that were measured.

3.1. The WISC Database

The database is designed so that a naive user can quickly understand the structure of the relations and the distribution of each attribute value. As a consequence, the results of the queries that are run in the benchmark are easy to understand and additional queries are simple to design. The attributes of each relation have distributions of values that can be used for partitioning aggregates, controlling selectivity factors in selections and joins, and varying the number of duplicate tuples created by a projection. It is also straightforward to build an index (primary or secondary) on some of the attributes, and to reorganize a relation so that it is clustered with respect to an index.

There are four "basic" relations in the database. We refer to them by the names of "thoustup", "twothoustup", "fivethoustup", and "tenthoustup" as they respectively contain 1000, 2000, 5000 and 10000 tuples. A fragment of the thoustup relation is shown in Figure 1 below. All the tuples are 182 bytes long, so that the four relations occupy approximately 4 megabytes of disk storage. However, in order to build queries that operate on more than one operand relations, we often generate two or more relations of the same size. For example, the join queries described below operate on two 10,000 tuple relations: "thoustupA" and "thoustupB". The attributes are either integer numbers (between 0 and 9999), or character strings (of length 52 characters). The first attribute ("unique1") is always an integer number that assumes unique values throughout the relation. We have made the simplest possible choice for the values of "unique1". For example, for the 1000 tuples relation "thoustup" unique1 assumes the values 0, 1, ... 999. For the relations with 10,000 tuples, the values of "unique1" are 0,1, ..., 9999. The second attribute "unique2" has the same range of values as "unique1". Thus both "unique1" and "unique2" are key attributes. However, while we have used a random number generator to scramble the values of "unique1" and "unique2", the attribute "unique2" is often used as a sort key. When relations are sorted, they are sorted with respect to this attribute. When we need to build a clustered index, again it is an index on "unique2". For instance, we may execute the following INGRES query to observe the effect of a primary index on a selection that retrieves 10% of the "twothoustup" relation:

```
range of t is twothoustup
retrieve (t.all) where t.unique2 < 200
```

After the "unique1" and "unique2" attributes come a set of integer-valued attributes that assume non-unique values. The main purpose of these attributes is to provide a systematic way of modeling a wide range of selectivity factors. Each attribute is named after the range of values the attribute assumes. That is, the "two", "ten", "twenty", "hundred",..., "tenthous" attributes assume, respectively, values in the ranges (0,1), (0,1,...,9), (0,1,...,19), (0,1,...,99), ... ,(0,1,...,9999). For instance, each relation has a "hundred" attribute which has a uniform distribution of the values 0 through 99. Depending on the number of tuples in a relation, the attribute can be used to control the percentage of tuples that will be duplicates in a projection or the percentage of tuples that will be selected in a selection or join query. For example, in the "twothoustup" relation, the "hundred" attribute can be used for projecting into a single attribute relation where 95% of the tuples are duplicates (since only 100 values are distinct among the 2000 attribute values). The INGRES format for this query would be:

```
range of t is twothoustup
retrieve (t.hundred)
```

The same "hundred" attribute can be used for creating 100 partitions in aggregate function queries. For example, we may query for the minimum of an attribute that assumes values randomly distributed between 0 and 4999 ("fivethous"), with the relation partitioned into 100 partitions:

```
range of t is twothoustup
retrieve (minvalue = min(t.fivethous by t.hundred ))
```

Finally, each of our relations has 3 string attributes. Each string is 52 letters long, with three letters (the first, the middle and the last) being varied, and two separating substrings that contain only the letter x. The three significant letters are chosen in the range (A,B,...,V), to allow up to 10,648 (22 * 22 * 22) unique string values. Thus all string attributes follow the pattern:

```
$xxxx ... xxx$xxx ... xxx$
   {25 x's}      {24 x's}
```

where "$" stands for one of the letters (A,B,...,V). Clearly, this basic pattern can be modified to provide for a wider range of string values (by replacing some of the x's by significant letters). On the other hand, by varying the position of the significant letters, the database designer can also control the cpu time required for string comparisons.

The first two attributes in this category are string versions of the "unique1" and "unique2" integer valued attributes. That is, "stringu1" and "stringu2" may be used as key attributes, and a primary index may be built on "stringu2". For example, in the thousand tuple relation, "thoustup", the stringu2 attribute values are:

A Fragment of the Thoustup Relation
(some attributes have also been omitted)

unique1	unique2	two	ten	hundred	thousand
378	0	1	3	13	615
816	1	0	4	4	695
673	2	0	6	26	962
910	3	0	2	52	313
180	4	0	0	20	74
879	5	1	9	29	447
557	6	1	7	47	847
916	7	0	4	54	249
73	8	0	6	26	455
101	9	0	2	62	657

Figure 1

```
"Axxxx ... xxxAxxx ... xxxA"
"Bxxxx ... xxxAxxx ... xxxA"
"Cxxxx ... xxxAxxx ... xxxA"
              .
              .
              .
"Vxxxx ... xxxAxxx ... xxxA"
"Axxxx ... xxxBxxx ... xxxA"
              .
              .
              .
"Vxxxx ... xxxBxxx ... xxxA"
"Axxxx ... xxxCxxx ... xxxA"
              .
              .
              .
"Vxxxx ... xxxVxxx ... xxxA"
"Axxxx ... xxxAxxx ... xxxB"
              .
              .
              .
"Ixxxx ... xxxBxxx ... xxxC"
"Jxxxx ... xxxBxxx ... xxxC"
```

The following two queries illustrate how these string attributes can be utilized. Each query has a selectivity factor of 1%.

```
range of t is tenKtup1
retrieve (t.all) where
t.stringu2D < "Axxxx ... xxxExxx ... xxxQ"

range of t is tenKtup2
retrieve (t.all) where
(t.stringu2E > "Bxxxx ... xxxGxxx ... xxxE")
        and
(t.stringu2E < "Bxxxx ... xxxLxxx ... xxxA")
```

The "stringu2" variables are initially loaded in the database in the same order in which they were generated, shown above, which is not sort order. The attribute "stringu1" assumes exactly the same string values as "stringu2" except that their position in the relation is randomly determined. As can be seen in the outline above, the leftmost significant letter varies most frequently (from A to V) and the rightmost significant letter varies least frequently (from A to C) in the thoustup relation. Thus, these strings give any special hardware or algorithms that can do short circuit comparison of strings ample opportunity to demonstrate their efficacy.

A third string attribute, "string4", assumes only four unique values:

```
"Axxxx ... xxxAxxx ... xxxA"
"Hxxxx ... xxxHxxx ... xxxH"
"Oxxxx ... xxxOxxx ... xxxO"
"Vxxxx ... xxxVxxx ... xxxV"
```

"String4" can be used to select with different selectivity factors and for partitioning (like the integer attribute "four").

3.2. The Wisconsin Benchmark

We have developed a standard set of queries which measure the cost of different relational operations:

(1) Selection with different selectivity factors.

(2) Projection with different percentages of duplicate attributes.

(3) Single and multiple joins.

(4) Simple aggregates and aggregate functions.

(5) Updates: append, delete, modify.

In addition, for most queries, we have designed two variations: one that can take advantage of a primary index, and one that can only use a secondary index. Typically, these two variations were obtained by using the "unique2" attribute in one case, and the "unique1" attribute in the other. When no indices are available the queries are the same.

3.3. Measurements

After the database and the queries had been built, we had to decide how to actually measure the time and resources consumed by each run. Our first decision was to start with an extensive sequence of stand-alone runs. We made sure that, when our benchmarks were run, our systems were in single user mode. Then, we built a mechanism to set up runs where the queries were run one at a time, in a strictly sequential pattern. This way, all the measurements that we obtained indicated the performance of each query, as a separate, stand-alone program. The impact of system overhead (e.g. the "open database" command) was diminished by running several similar queries in sequence and taking the average time.

While each system evaluated provided facilities for gathering detailed statistics on the resources (ie. CPU, disk transfers) consumed by a query, after thorough consideration, we decided to use elapsed time as the main performance measure. For the IDM 500, this time was taken as the elapsed time on the host machine[4].

3.4. Effects of Database and Buffer Size

In our first benchmark tests, the queries primarily referenced one 2000 tuple relation. Since this relation is approximately 320,000 bytes long, when a million bytes of buffer space are available, the active portion of the database fits into memory. While the results of these tests were interesting, they did not fit most users' view of reality. Therefore, we modified the queries to reference the 10,000 tuple relations (each of which is approximately. 1.8 megabytes in size). In addition, in order to minimize the effect of the buffer size when running repeated queries, each query was run ten times alternating between the two 10,000 tuple relations. When this strategy is combined with 1 megabyte of buffer space (the most allocated to any of the systems tested), query i will leave almost nothing in the buffer pool that is of use to query i+1.

4. The Benchmark: Measurement and Analysis

In this section, we present a subset of our benchmark measurements, and analyze the results. We have divided this section into five subsections. There is one subsection for each of the relational operations (selection, projection, join), one for aggregates, and one for updates (delete, append, modify). For each type of query, we first describe the main criteria that were used to compare the different systems and the effects

[4] The command "time" was used on Unix. On VMS, "date" was used.

that we were attempting to measure. Determining some of these criteria, however, was not always straightforward. Over the period of time that we were running the benchmarks, preliminary results forced us to change certain queries in order to gain more insight into the impact of a particular parameter.

For example, it was only after a long series of benchmarks that we first realized that the cost of duplicate record elimination was a factor that made many of our comparisons meaningless. There are two alternative ways of measuring the time required for a query. One is to retrieve the selected tuples into a relation (that is writing them to disk). The other was to display them on a user's terminal. Unfortunately, both alternatives have drawbacks. Producing a result relation (by an INGRES "retrieve into" statement), has the side effect of checking for and removing duplicate tuples from the result relation. Thus, the time obtained for a retrieval query includes the time to perform duplicate elimination. The other alternative was to retrieve result tuples to the screen. In this case, however, times for queries that retrieve a large number of tuples would have mainly measured the time to transfer a large amount of data to a terminal (rather than the time required by the database management to execute the query).

The principal solution we choose was to place the result tuples in a relation but to do so without eliminating duplicate tuples (by using the "-rheap" option of INGRES, we discovered that duplicate elimination can be turned off). However, we also wanted to examine the impact of the communications channel between the IDM 500 and the host. Thus, for some selected queries, we also "retrieved" the results to the screen.[5]

Another problem that we faced was filtering the meaningful results from the vast quantities of raw data produced by the original benchmark runs (which contained over 100 queries). Rather than showing an impressive but overwhelming collection of numbers, we decided to choose a representative sample of results for each query type. The sample had to be small enough to be presented in this paper, without omitting the information necessary to support our conclusions. These choices resulted in a number of tables that show the elapsed time in seconds for the representative queries in the 5 classes. Our analysis in each of the 5 subsections then concentrates on the numbers shown in these tables.

4.1. Selection Queries

The speed at which a database system or machine can process a selection operation depends on a number of different factors including:

(1) storage organization of the relation

(2) impact of the selectivity factor (how many result tuples are produced by the query)

(3) impact of specialized hardware

(4) cost of sending the result tuples to the screen (compared to the cost of storing them in a new relation)[6]

Our benchmark investigated the impact of each of these factors. In determining the impact of the storage organization on the performance of the query, we evaluated four different storage organizations:

(1) *heap organization* - this is an unstructured storage organization in which the tuple order corresponds to the order in which the tuples were loaded into the relation. This organization has no suitable secondary storage structures for enhancing performance. We evaluated this organization for two reasons. First, it provides information as to how fast a system can process an arbitrary adhoc query. While we understand that in most real systems, there will generally be an appropriate index, one of the "nice" features of a relational system is that users can pose arbitrary (and unanticipated) queries to the database system. In addition, by measuring the response time for the heap organization, when the same query is run in the presence of a suitable index, we are better able to understand the performance improvement that can be obtained by having the appropriate index available.

(2) *index on key attribute* - in this case the relation is sorted (clustered) on the same attribute on which an index has been constructed. Both the university and commercial versions of INGRES use an ISAM organization for this case. The IDM 500 first sorts the file on the key attribute and then constructs a B-tree on the key attribute.

(3) *hash on key attribute* - in this case tuple placement is randomized by applying a hashing function to the key attribute. This access mechanism was available only with INGRES. It was used only for those queries that returned a single tuple (see Table 3).

(4) *index on non-key attribute* - in this case the relation is sorted on a different attribute from the one on which the index has been constructed. For both versions of INGRES, we used a hashed, dense secondary index to obtain this storage structure. Dense implies one index entry per attribute value; hashed means that after the secondary index is constructed, the index entries are hashed on the index attribute value. This permits one to take an attribute value and in one disk access find the index page containing the appropriate index entry and in another disk access locate the data page containing the tuple with the desired value. The IDM 500 uses a B-tree mechanism to support this type of index.

To determine how the selectivity factor of a query influences performance, for each storage structure (and each system) we varied the selection criteria to produce result relations with a range of different sizes. The selectivity factors considered were 1%, 10%, 20%, 50%, and 100%. In addition, we also measured the time to retrieve a unique tuple (Table 3). Examination of the results of these tests revealed that the queries with selectivity factors of 1 tuple, 1%, and 10% were

[5] Actually, the result tuples were sent to "/dev/null" (the Unix equivalent of a black hole) to make sure that the actually printing speed of the different terminals would not have an effect.

[6] Although the cost of formatting tuples for screen display could also have been measured in the context of queries other than selection queries, we found it easier to isolate it from other cost factors in this context.

representative of the relative performance of the various systems.

The impact of specialized hardware was evaluated by running the same queries both with and without indices on the same IDM 500 with and without the database accelerator (DAC) turned on.

The results of our experiments are shown in Tables 1, 2 and 3 below. The response times presented represent an average time based on a test set of ten different queries (each, however, with the same selectivity factor).

One can draw a number of conclusions from these results. Both IDM and Commercial INGRES perform selection faster than University INGRES. However, the improvement is not dramatic. When a clustered index is available (the most common situation, probably), C-INGRES and IDMnodac outperform U-INGRES by factors of 2 and 3 respectively. On most selection queries, the IDMdac is about twice faster than C-INGRES. In one case, however, when a non-clustered index exists for the source relation, and only 100 tuples (out of 10,000) are retrieved, the IDM (with or without dac) is extremely fast, and outperforms C-INGRES by a factor of 15. This situation demonstrates clearly the superiority of the B-tree mechanism for supporting a non-key index.

Table 1
Selection Queries without Indices
Integer Attributes
Result Tuples Inserted into Relation
Total Elapsed Time in Seconds

System	100	1000
U-INGRES	53.2	64.4
C-INGRES	38.4	53.9
IDMnodac	31.7	33.4
IDMdac	21.6	23.6

Table 2
Selection Queries with Indices
Result Tuples Inserted into Relation
Integer Attributes
Total Elapsed Time in Seconds

Number of Tuples Selected from 10,000 Tuple Relation

	Clustered Index		Non-Clustered Index	
System	100	1000	100	1000
U-INGRES	7.7	27.8	59.2	78.9
C-INGRES	3.9	18.9	51.2	60.8
IDMnodac	2.0	9.9	3.8	27.6
IDMdac	1.5	8.7	3.3	23.7

Table 3
Selection Queries with Clustered Indices
Integer Attributes
Result Tuples Displayed on Screen
Total Elapsed Time in Seconds

Number of Tuples Selected from 10,000 Tuple Relation

System	1	100
U-INGRES	3.8	6.9
C-INGRES	0.9	5.0
IDMnodac	0.8	2.9
IDMdac	0.7	2.7

When estimating the speedup obtained by the database accelerator (by comparing the IDMdac and IDMnodac numbers), we were somehow surprised to find out that it was at most 1.47 (in Table 1), and as low as 1.07 for selection on an indexed attribute (in Table 3).

One interesting result illustrated by Tables 1 and 2 is that for C-INGRES, the selections with a non-key index are actually slower than with the heap organization (the same anomaly is observed for U-INGRES, but the discrepancy is within the margin of error). The most plausible explanation is that when the non-key (and hence non-clustered) index is used, a number of pages are accessed multiple times. With 2,048 byte pages the source relation occupies approximately 909 data pages. Scanning the relation in a heap fashion requires 909 page accesses. On the other hand selecting 1000 tuples (10% selectivity factor), through a non-key index may require more than 1000 page accesses. The main conclusion to be drawn is that the query optimizer failed to recognize that the index should not be used to process the query.

In Table 3, we have included selected measurements that provide a clear estimate for the cost of formatting result tuples for display on the screen. Only the index case is shown, as the differences for the non-index would hidden by the long retrieval time. Also, we only show very low selectivity factors (a single tuple, or 1%), since it is unlikely that a user would look at a table of a 1000 tuples on the screen. By comparing the INGRES and the IDM numbers in Table 3, we conclude that the performance of a backend database machine is only marginally affected by the cost of transferring result tuples to the host computer. Another conclusion that we may draw by comparing Table 2 and Table 3, is that for all systems the cost of formatting results for screen display is relatively high (and it is about the same for all systems). Note that when retrieving into a relation, our measurements account for the cost of writing the result relation to the disk, without eliminating duplicate records.[7] Thus when comparing Tables 2 and 3, we are truly comparing the cost of writing results to the disk, to the cost of formatting and displaying tuples on the screen. While measuring the cost of duplicate elimination is also important, it was not possible to isolate it from other cost components in the selection queries. For this reason, we chose to do this measurement in the context of projection queries (Section 4.3, below).

4.2. Join Queries

In looking at join queries we were interested in investigating a number of different issues. First, we were interested in how query complexity affected the relative performance of the different systems. Thus, we considered a range of queries, with different degrees of complexity. Second, we were curious about the different join algorithms the systems used. Running join queries on a stand-alone basis would make it possible to verify how efficiently the buffer management strategy of each system supported these algorithms (since the join algorithm determines the page reference string). We knew, a priori, that:

(1) Without indices, university INGRES used a nested loops join in which the storage structure of a copy of the inner relation is converted to a hashed organization before the join is initiated

(2) Commercial INGRES used primarily sort-merge join techniques.

(3) The IDM 500 with and without the DAC used a simple nested loops join ($O(n^2)$) algorithm.

Third, we were interested in how the different systems took advantage of secondary indices on joining attributes, when these were available. Finally, we wanted to see how the database accelerator impacted join times.

With the above criteria in mind, we built a set of ten representative join queries. The source relations were always the ten thousand tuple relations. However, when a selection was performed before the join, the size of the operand relation was reduced by a factor of ten. Ten thousand tuples of length 182 bytes in each source relation were enough to cause substantial I/O activity, and make visible the effect of varying input parameters (such as query complexity and join selectivity factors).

Query complexity was modeled by performing before the join zero, one or two selection operations (e.g. joinAselB selects on relation B, and joins the selected relation with A, while joinselAselB selects on both A and B before the join). A more complex join query involves two selections, followed by two joins (see "joinCselAselB", below).

After a preliminary analysis, we have again decided to filter the results of our measurements, and to present timings for a smaller set of join queries. These appear in Tables 4, 5 and 6. The names of the queries describe their contents. However, the reader may wish to refer to Appendix I, where the join queries have been explicitly listed.

Our first observation is that, for joins, more than for any other type of queries, each system's performance varies widely with the kind of assumptions that are made (e.g. indices versus no indices, special hardware versus no special hardware, complex versus simple join, etc). However, our measurements clearly show that for joins without indices commercial INGRES is the only system to always provide acceptable performance. The dramatic improvement over university INGRES is due to the use of a sort-merge algorithm. The IDM, on the other hand, still uses a slow nested-loops algorithm. In previous experiments (whose results are not presented here), we found out that the DAC could achieve a reasonable level of performance for joins without indices when the relations were smaller, and thus mostly fit in memory. On the other hand, with the 10,000 tuple relations and no suitable indices, the IDM performance (with or without the DAC) is unacceptable. However, by building an index "on-the-fly", the IDM user (or a smarter query optimizer), can obtain excellent performance. For example, consider the query joinAselB in which B is first restricted to form B' and then B' is joined with A to produce the result relation. If instead of writing this query

Table 4
Join Queries Without Indices
Integer Attributes
Total Elapsed Time in Seconds

System	Query: joinAselB	joinABprime	joinCselAselB
U-INGRES	611 secs.	581 secs.	563 secs.
C-INGRES	109 secs.	156 secs.	127 secs.
IDMnodac	> 5 hours	> 5 hours	> 5 hours
IDMdac	> 5 hours	> 5 hours	> 5 hours

Table 5
Join Queries with Indices
Integer Attributes
Total Elapsed Time in Seconds
Primary (clustered) Index on Join Attribute

System	Query: joinAselB	joinABprime	joinCselAselB
U-INGRES	126.5	99.5	544.5
C-INGRES	54.0	103.0	64.0
IDMnodac	31.0	35.5	44.5
IDMdac	23.5	27.5	35.0

Table 6
Join Queries with Indices
Total Elapsed Time in Seconds
Secondary (nonclustered) Index on Join Attribute

System	Query: sjoinAselB	sjoinABprime	sjoinCselAselB
U-INGRES	269.5	191.0	633.0
C-INGRES	118.0	108.0	144.5
IDMnodac	84.5	48.5	108.5
IDMdac	71.5	35.5	88.0

as one IDL command, the user first forms B' (without the help of any permanent indices), then constructs an index on the join attribute of B', and final performs the join, we observed that the execution time for the query could be reduced from over 5 hours to 108 seconds!

When the appropriate indices exist, the IDM achieves an excellent level of performance on join operations. However, the DAC only adds to this performance a speedup of 1.3. Another interesting result is that the performance of commercial INGRES gets closer to the IDMdac for complex joins (joinCselAselB runs only 1.8 times faster on IDMdac than on commercial INGRES, compared to 3.7 times faster for joinABprime). The query optimizer in commercial INGRES appears to be very efficient in the case of complex join queries. Note that the query joinCselAselB performs two selections on 10,000 tuple relations, followed by two joins on 1,000 tuple relations (see Figure 2).

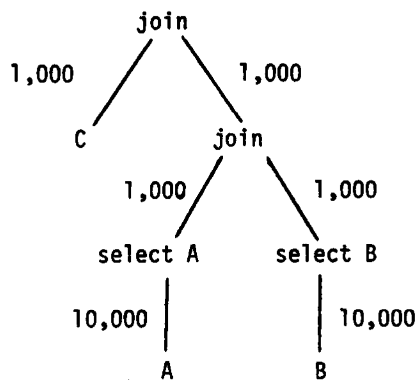

Figure 2: joinCselAselB

However, the cost of this query is only slightly higher than the cost of the two selections (127 secs compared to 107.8 secs when there are no indices).

One curious anomaly is the fact that joinAselB (a selection followed by a join) ran faster than joinABprime (the same join without selection) on commercial INGRES. One possible explanation could be that the query optimizer allocated more memory for executing joinAselB than for joinABprime because the operand relation B is larger than Bprime.

4.3. Projection Queries

Implementation of the projection operation is normally done in two phases. First a pass is made through the source relation to discard unwanted attributes. A second phase is necessary in order to eliminate any duplicate tuples that may have been introduced as a side effect of the first phase (i.e. elimination of an attribute which is the key or some part of the key). The first phase requires a complete scan of the relation. The second phase is normally performed in two steps. First, the relation is sorted to bring duplicate tuples together. Next, a sequential pass is made through the sorted relation, comparing neighboring tuples to see if they are identical. Secondary storage structures such as indices are not useful in performing this operation.

While our initial benchmark contained other queries which projected on different attributes and thus produced result relations of a variety of sizes, the following two queries are indicative of the results observed. The first query projects the 10,000 tuple relation with a projectivity factor of 1%. Thus, it eliminates 99% duplicate records and produces 100 tuples. The second query is a projection of the 1,000 tuple relation, with a 100% projectivity factor. In this case, although no duplicate tuples are produced by the projection, the result relation was still sorted and scanned. Thus, this particular query provides us with an estimate for the cost of duplicate elimination involved in any retrieval "into" a result relation (see Section 4.1). In order to make this estimate as accurate as possible, it was desirable to minimize the time of getting the relation off the disk. This effect was achieved by actually running in sequence 10 copies of the same query, and dividing the total run time by 10.

Our first observation from this table is the relatively high cost of projection compared to selection. For commercial INGRES and IDM (dac and nodac), it takes more than 3 times longer to project on 1% of the tuples in the 10,000 tuple relation than to select 1% of the tuples of the same relation. This discrepancy is due to the sort phase in the projection. Sorting 10,000 tuples - even if duplicates are gradually eliminated [BITT82] - requires a long time, compared to the cost of scanning the relation once only (as required by the selection).

Another striking result is the speedup achieved by the dac in the case of a high projectivity factor. While the dac only improved selection by a factor of 1.3, the speedup observed here is 1.8.

4.4. Aggregate Queries

We have considered both simple aggregate operations (e.g. minimum value of an attribute) and complex aggregate functions in which the tuples of a relation are first partitioned into non-overlapping subsets. After partitioning, an aggregate operation such as MIN is computed for each partition. For the complex aggregate functions, we have repeated our experiments for a wide range of partition sizes (by selecting, as the partitioning attribute, attributes with different selectivity factors).

In the following tables, we have retained only the results for three of the most representative queries: a minimum on a key attribute and two aggregate

Table 7
Projection Queries
(Duplicate Tuples are Removed)
Total Elapsed Time in Seconds

	Query Type	
System	100/10,000	1000/1000
U-INGRES	64.6	236.8
C-INGRES	26.4	132.0
IDMnodac	29.3	122.2
IDMdac	22.3	68.1

functions; each with 100 partitions. One objective of these three queries was to examine whether any of the query optimizers would attempt to use the indices available to reduce the execution time of the queries. For the min.key query, a very smart query optimizer would recognize that the query could be executed by using the index alone. For the two aggregate function queries, we had anticipated that any attempt to use the secondary, non-clustered index on the partitioning attribute would actually slow the query down as a scan of the complete relation through such an index will generally result in each data page being accessed several times. One alternative algorithm is to ignore the index, sort on the partitioning attribute, and then make a final pass collecting the results. Another algorithm which works very well if the number of partitions is not too large is to make a single pass through the relation hashing on the partitioning attribute.

We got very mixed results from these tests. First, we were puzzled by what changes were made to the aggregate function algorithms in commercial INGRES that caused it to run slower than university INGRES (especially considering that the page size used by commercial INGRES is twice that of university INGRES). As for the use of indices, it appears that for both university INGRES and IDM the query optimizer chose to ignore the index in all cases. While this decision leaves both systems with a slow scalar aggregate operation, it is a better alternative for the execution of aggregate functions.

Table 8
Aggregate Queries Without Indices
Total elapsed time in seconds

System	Query Type: MIN Scalar Aggregate	MIN Aggregate Function 100 Partitions	SUM Aggregate Function 100 Partitions
U-INGRES	40.2	176.7	174.2
C-INGRES	34.0	495.0	484.8
IDMnodac	32.0	65.0	67.5
IDMdac	21.2	38.2	38.2

Table 9
Aggregate Queries With Indices
Total elapsed time in seconds

System	Query Type: MIN Scalar Aggregate	MIN Aggregate Function 100 Partitions	SUM Aggregate Function 100 Partitions
U-INGRES	41.2	186.5	182.2
C-INGRES	37.2	242.2	254.0
IDMnodac	27.0	65.0	66.8
IDMdac	21.2	38.0	38.0

Finally, while the DAC reduces the time for the scalar aggregate in a proportion similar to the selection queries (the speedup observed is 1.27), it improves more significantly the performance on aggregate functions (speedup of 1.7).

4.5. Update Queries

The numbers presented in the tables below were obtained for stand-alone updates (delete, append, and modify). The principal objective of these queries was to look at the impact of the presence of both clustered and non clustered indices on the cost of updating, appending or deleting a tuple. A more realistic evaluation of update queries would require running these benchmarks in a multiprogramming environment, so that the effects of concurrency control and deadlocks could be measured.

These results are basically what we expected to see. First, for all systems, the advantage of having an index to help locate the tuple to be modified overshadows the cost of updating the index. The numbers obtained for the "delete 1 tuple" and "modify 1 tuple" queries (in Tables 10 and 11) support this claim very strongly. However, it should be noted that not enough updates were performed to cause a significant reorganization of the index pages. Also the reader should be aware of the fact that three indices had been built

Table 10
Update Queries Without Indices
Total elapsed time in seconds

System	Query Type: Append 1 Tuple	Delete 1 Tuple	Modify 1 Tuple (Key Attr)
U-INGRES	5.9	37.6	37.7
C-INGRES	1.4	32.3	32.8
IDMnodac	0.9	22.8	29.5
IDMdac	0.7	20.8	20.9

Table 11
Update Queries With Indices
Total elapsed time in seconds

System	Query Type: Append 1 Tuple	Delete 1 Tuple	Modify 1 Tuple (Key Attr)	Modify 1 Tuple (Non-Key Attr)
U-INGRES	9.4	6.8	7.2	9.1
C-INGRES	2.1	0.5	1.6	1.6
IDMnodac	0.9	0.4	0.6	0.5
IDMdac	0.8	0.4	0.5	0.5

on the updated relation (one clustered index and two secondary indices), in order to account for the cost of updating indices in a significant way.

Another observation, that surprised us at first, is the low cost of the append compared to the cost of the delete, in the no-index case. The explanation for this discrepancy is that all the systems append new tuples without checking if they were not already present in the relation. Thus, appending a tuple only involves writing a new tuple, while deleting a tuple requires scanning the entire relation first. On the other hand, when a clustered index is available, deleting is faster than appending a tuple, apparently because the index is modified but the tuple is not physically deleted. Finally, the performance of all systems on the "modify non-key" (that is modify a tuple identified by a qualification on a non-key attribute) demonstrates a very efficient use of a secondary index to locate the tuple. However, one could again argue that the right algorithm for this query would require verifying that the modified tuple does not introduce an inconsistency by duplicating an existing tuple.

5. Conclusions and Future Research

In this paper, we have presented and interpreted a set of measurements performed on several database management systems. Originally, we had intended to compare the relative performance of database machines that use special purpose hardware and conventional database management systems that run on general purpose computers. However, in the early stages of our benchmark design, we realized that we had to limit the scope of our measurements in order to reach any valid conclusions. The main limitation of the present study is that it addresses only the *single user* case. At this point, we must therefore admit that our benchmark is neither an exhaustive comparison of different systems, nor a realistic approximation of what measurements in a multiuser environment will be like.

However, we have found that limiting our experiments to stand-alone queries was the only systematic way to isolate the effects of specific hardware configurations, operating system features, or query execution algorithms. For this reason, the single user case constitutes a necessary baseline measure which we will use in the interpretation of multiuser benchmark results.

Finally, we would like to emphasize that designing the WISC database, and the set of queries that go with it, represents a first attempt at introducing a scientific approach to database benchmarking. We will continue refining the single user benchmark while we also start work on multiuser benchmarks.

6. Acknowledgments

A large number of people deserve thanks for making this paper possible. First, Rakesh Agrawal helped in the design of the relations and queries used in our benchmark. Second we would like to thank Britton-Lee Incorporated and Relational Technology Incorporated for their support in the benchmarking process. Although only a handful of database accelerators were running when we began the benchmarking process, Britton-Lee generously made a DAC available for us. We especially wish to thank Mike Ubell of Britton-Lee for helping us run our benchmarks remotely. We also wish to thank Derek Frankforth, Bob Kooi, Trudi Quinn, and Larry Rowe at RTI for their help in bringing up the benchmark on VMS. We also wish to thank Haran Boral for his suggestions on the earlier drafts of this paper.

Finally we would like to acknowledge the support for this research provided by the National Science Foundation under grant MCS82-01870 and the Department of Energy under contract DE-AC02-81ER10920.

7. References

[BABB79] Babb, E. "Implementing a Relational Database by Means of Specialized Hardware," ACM TODS, Vol. 4, No. 1, March 1979.

[BANE78] Banerjee J., R.I. Baum, and D.K. Hsiao, "Concepts and Capabilities of a Database Computer," ACM TODS, Vol. 3, No. 4, Dec. 1978.

[BITT82] Bitton, D. and D.J. DeWitt, "Duplicate Record Elimination in Large Datafiles," to appear ACM Transactions on Database Systems.

[BORA82] Boral, H., DeWitt, D.J., Friedland, D., Jarrell, N., and W. K. Wilkinson, "Implementation of the Database Machine DIRECT," IEEE Transactions on Software Engineering, November, 1982.

[BORA83] Boral H. and D. J. DeWitt, "Database Machines: An Idea Whose Time is Past? A Critique of the Future of Database Machines," Technical Report, Computer Sciences Department, University of Wisconsin, April 1983.

[DEWI79] DeWitt, D.J., "DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems," IEEE Transactions on Computers, June 1979, pp. 395-406.

[DEWI81] DeWitt, D. J., and P. Hawthorn, "Performance Evaluation of Database Machine Architectures," Invited Paper, 1981 Very Large Database Conference, September, 1981.

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

[HAWT82] Hawthorn P. and D.J. DeWitt, "Performance Evaluation of Database Machines," IEEE Transactions on Software Engineering, March 1982.

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

[MCGR76] McGregor, D.R., Thomson, R.G., and W.N. Dawson, "High Performance Hardware for Database Systems," in *Systems for Large Databases*, North Holland, 1976.

[OZKA75] Ozkarahan, E.A., S.A. Schuster, and K.C. Smith, "RAP - Associative Processor for Database Management," AFIPS Conference Proceedings, Vol. 44, 1975, pp. 379 - 388.

[OZKA77] Ozkarahan, E.A., Schuster, S.A. and Sevcik, K.C., "Performance Evaluation of a Relational Associative Processor," ACM Transactions on Database Systems, Vol. 2, No.2, June 1977.

[STON76] Stonebraker, M.R. , E. Wong, and P. Kreps, "The Design and Implementation of INGRES," ACM TODS 1, 3, (September 1976).

[STON80] Stonebraker, M. R, "Retrospection on a Database System," ACM TODS 5, 2, (June 1980).

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

[SU75] Su, Stanley Y. W., and G. Jack Lipovski, "CASSM: A Cellular System for Very Large Data Bases", Proceedings of the VLDB Conference, 1975, pages 456 - 472.

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

Appendix I

List of Join Queries in INGRES Format

joinAselB

```
range of t is tenthoustup
range of w is tenthoustup2
retrieve into tempsel1(t.all,w.all) where
(t.unique2 = w.twounique2) and w.twounique2 < 1000
```

sjoinAselB

```
range of t is tenthoustup
range of w is tenthoustup2
retrieve into tempsel1(t.all,w.all) where
(t.unique1 = w.twounique1) and w.twounique1 < 1000
```

joinABprime

```
range of t is tenthoustup
range of b is tempBprime
retrieve into tempjoinABpr (t.all,b.all)
where t.unique2 = b.twounique2
```

sjoinABprime

```
range of t is tenthoustup
range of b is tempBprime
retrieve into tempjoinABpr (t.all,b.all)
where t.unique1 = b.twounique1
```

joinCselAselB

```
range of o is thoustup
range of t is tenthoustup
range of w is tenthoustup2
retrieve into tempsel3(t.all,w.all)
where (t.unique2 = w.twounique2) and
(w.twounique2 < 1000) and (t.unique2 < 1000)
and (t.unique2 = o.oneunique2)
```

sjoinCselAselB

```
range of o is thoustup
range of t is tenthoustup
range of w is tenthoustup2
retrieve into ntmpsel3(t.all,w.all)
where (t.unique1 = w.twounique1)
and (w.twounique1 < 1000) and (t.unique1 < 1000)
and (t.unique1 = o.oneunique1)
```