Performance Evaluation of Main Memory Database Systems*

Dina Bitton
Carolyn Turbyfill
TR 86-731
January 1986

Department of Computer Science
Cornell University
Ithaca, NY 14853

* This research was partially supported by the National Science Foundation under grant DCR-8410889.

PERFORMANCE EVALUATION OF MAIN MEMORY DATABASE SYSTEMS

Dina Bitton Carolyn Turbyfill

Department of Computer Science, Cornell University

Ithaca, NY 14853

In this paper we present the results of a comprehensive benchmark of the relational **M**ain **M**emory **D**atabase **S**ystem (MMDBS), that is the foundation of the interactive office system, **O**ffice-**B**y-**E**xample (OBE). Based on this case study, we identify issues that must be considered in the design and implementation of MMDBS's. We determine relevant performance metrics and describe techniques for benchmarking MMDBS's.

TABLE OF CONTENTS

1. INTRODUCTION

The demand for fast response time and the availability of larger and inexpensive memories are motivating designers of database systems to reconsider the assumption that databases reside on disk during transaction processing. The **OBE**, **Office-By-Example**, **system**, developed at IBM Watson Research Center, follows that trend [ZI82, AHK85, WAB86]. The core of OBE is a **Main Memory Database System**, MMDBS, that exploits the memory residence assumption in its approach to database physical design, query processing algorithms and query optimization. In this paper, we describe and analyze a benchmark of the OBE MMDBS. Based on this case study, we determine appropriate performance metrics and systematic benchmarking techniques for main memory database systems.

The memory residence assumption can be viewed as a logical extension of the use of large buffers to enhance the performance of conventional disk database management systems (DDBS). In the design of conventional database systems, it is assumed that databases are orders of magnitude larger than the available memory capacity. Databases are stored on a mass-storage device, and this results in long access delays during query execution. Thus in DDBS's, file organization, access methods and buffer management are designed to limit these I/O delays. Performance is enhanced by providing a large buffer pool, where data records may be prefetched or frequently accessed data, such as indices, can be kept. The acquisition of large memories for this buffer pool can be evaluated in terms of performance gain per dollar-cost. In a recent study, Gray proposes a rule for determining a cost-effective size for a main memory buffer pool. Based on customers requirements and current prices of hardware, he establishes the "Five-Minute Rule": every database page that is referenced every five minutes should be memory resident [Gr85].

As the price of memory decreases, it is becoming feasible to keep the entire database, or the part of it referenced by a query, in main memory. Database systems that require fast response times and systems on workstations are good candidates for implementation as MMDBS's. The cost of acquiring large memories for a MMDBS can be justified when disk accesses, averaging 30 msec per block I/O, impose unacceptable delays on response time. For instance, meeting tight bounds on response time in real time systems and certain information systems may require a MMDBS. Office workstations involve a lower range of cost and performance where MMDBS's may become a viable alternative to DDBS's. The ratio of main memory to disk capacity is typically 1:10 in workstations, in contrast to at least 1:100 in mainframes. As the ratio of main memory to disk capacity decreases, the advantage of storing the database on disk diminishes. Furthermore, frequent accesses to slow disk storage incur delays unacceptable to the interactive user, who expects a database system to respond as quickly as an editor [AHK85].

A number of recent studies investigate the implication of the memory residency assumption on the design of databases and database management systems [KM84, DKO84, AHK85, Sh85, LC85]. In particular, analytical models and simulation are used in [DKO84, Sh85, LC85] to evaluate access methods and join algorithms under this assumption. However, experience with implementing MMDBS's is still lacking.

In benchmarking MMDBS's, many of the techniques previously used in benchmarking DDBS's [BDT83, BT84, BT85, BCH84, BD84] are inadequate because certain performance parameters are specifically related to the main memory residence assumption. In particular, space requirements, in the form of virtual and real memory, and memory management, by the operating system and the database system, are critical parameters in the design of a MMDBS benchmark. MMDBS's must pay careful attention to keeping both the code and the data storage structures compact. Thus, in the design and analysis of a MMDBS benchmark, we contend that performance must be evaluated in terms of the **space-time product** [Bu76,De80], rather than by time complexity alone.

The remainder of this paper is organized as follows. In Section 2, we present an overview of OBE and describe the hardware and software configurations of the two IBM systems on which we benchmarked OBE (an IBM 3081 and an IBM 4341 machine). In Section 3, we explain our benchmarking methodology. This methodology comprises the definition of appropriate performance metrics, timing techniques, and the design parameters for our test queries. In Section 4, we evaluate the performance of OBE on the test queries and describe our techniques for isolating the implementation features that affect the system's performance. In Section 5, we consider the space requirements of these test queries. In Section 6, we summarize our results and examine their implications for the design of future MMDBS's. Finally, we present our conclusions in Section 7.

2. DESCRIPTION OF THE SYSTEM

The OBE prototype was designed for an IBM 370 machine running VM/CMS operating system. In this section, we describe our test environment and give an overview of physical database design and query processing in OBE.

2.1. Hardware and Operating System

We performed our measurements on two machines. The first was an IBM 3081 running under VM Operating System Release HPO 3.2 at IBM T.J. Watson Research Center. The second was an IBM 4341, which led to slower execution of OBE queries, but might be more realistic for an office environment. In both cases, our test database was initially created and stored on an IBM 3380 disk. Comparing the performance of OBE on these two machines was interesting by itself, since we did not know how much adding CPU power would enhance the performance of the main memory database system. Of course, we expected added CPU power to provide more of an advantage to a CPU bound MMDBS than to an I/O bound DDBS. In addition, the 4341 times could be compared to times obtained in previous benchmarks of other database systems running on machines with comparable CPU's [BDT83, BT84, BD84, BT85].

Under VM, each user is given a virtual machine configured with a certain amount of virtual storage and virtual I/O devices. Typically, OBE runs in a virtual machine with 4 to 16 megabytes of virtual storage, depending on the size of the database. Pages are 4K bytes, paging is on demand and only referenced pages are kept in real storage [IBM83].

The VM **INDICATE** command [IBM83,Po85] allows the user to display the utilization of and contention for major system resources. The command provides information on processor and storage usage, the number of **start I/O** (SIO) instructions, and number of pages read and written. A variant of the **INDICATE** command, **INDICATE USER**, can be used before and after a program's execution to get summary information about resource usage of a single virtual machine.

VM also provides a **MONITOR** command [IBM83,Po85] for use by system analysts and operators. The Monitor collects data in two ways:

(1) Event driven: CP, the operating system that controls the real hardware and manages virtual machines, can be instructed to issue a **MONITOR CALL** instruction. This causes an interrupt when certain events occur, such as the addition of a user to the CPU scheduler's eligible queue.

(2) Timer driven: Some data can be collected at fixed intervals. The interval is a parameter that can be specified in the **MONITOR** command. On the 3081 we used, during regular working hours, a timer interrupt once every 60 seconds caused the collection of data from system counters concerning device usage.

FIGURE 1. The Two-Dimensional Interface

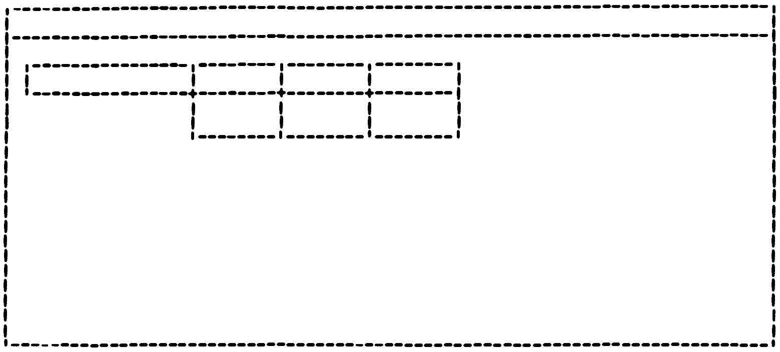

FIGURE 1.a Skeleton Table

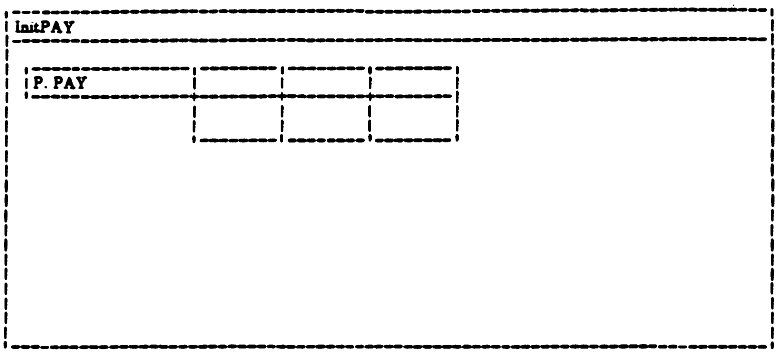

FIGURE 1.b Invoking a Relation Template

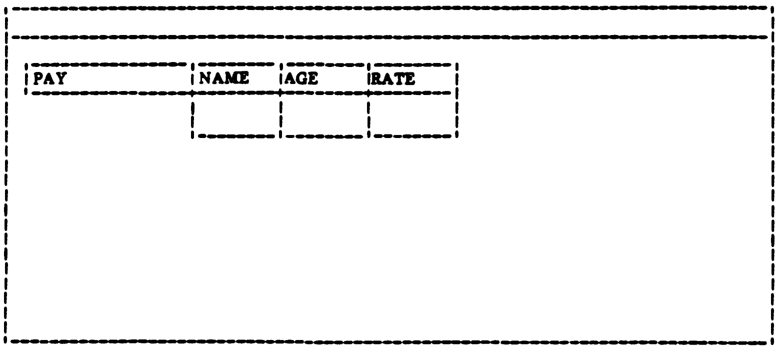

FIGURE 1.c A Relation Template

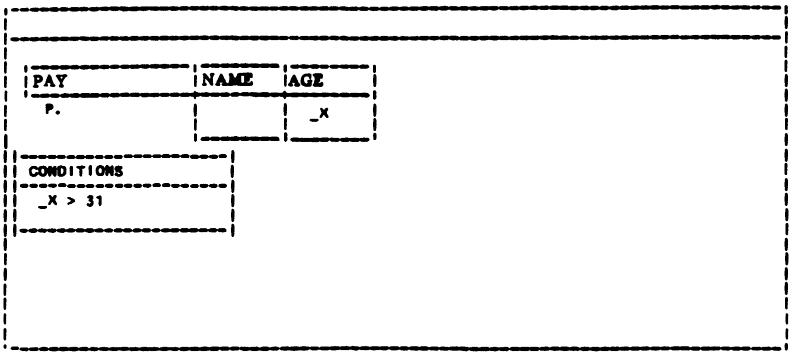

FIGURE 1.d A Select Project Query

2.2. The Database System in OBE

Detailed descriptions of the design and implementation of OBE can be found in [Zl82, AHK84, Wh85, WAB86, SK84, KMZ84, KH84]. In this section, we present an overview of the components that are relevant to this performance study: the target environment, the user interface, the physical database design and the access methods.

2.2.1. The target environment

A pivotal assumption in the design of the OBE prototype was that it would be targetted towards an environment where an unsophisticated user would create new database tables and formulate ad-hoc queries with ease. It was assumed that the user would expect response times similar to those of other interactive programs such as editors. The emphasis in the design was on ease of use and simple database design. Performance would be achieved by

(1) Keeping the data in main memory during query processing.

(2) Automatically inverting every relation on every attribute[1]

Although both approaches are unrealistic for a large, update intensive transaction management system, they are more feasible for a database in an office environment for four reasons. First, it is assumed that the database consists of a large number of relatively small tables and that most queries are retrieval queries involving multiple table joins. Second, it is assumed that the relations involved in any particular query will fit in memory. Third, if the relations queried are memory resident, then the inverted indices are instrumental in efficiently processing multiple table joins. Finally, if updates are infrequent, the overhead of maintaining the indices may be acceptable as long as the indices are memory resident.

2.2.2. The interface

OBE has kept the QBE [Zl75, Zl82, KMZ83] two-dimensional interface, which is an attractive implementation of the relational domain calculus. The underlying query language is relationally complete [Zl75, Zl82, KMZ83] and the syntax is very simple. It is based on making entries in skeleton tables instead of writing lengthy declarations and queries. Using a screen editor and pre-programmed function keys, the user may display skeleton tables (Figure 1.a), relation templates containing a relation name and attribute names (Figure 1.c), and condition boxes (where selection conditions are specified). The user does not have to remember the names of attributes to specify a query: these are automatically displayed. By positioning the cursor and using a function key, attributes that are not relevant to the current query can be erased. Table entries may consist of commands, **example elements**, literals or simple qualifiers: "P." for Print, "_X" for **example element** "X", "10" for the literal value 10, and "< 10" for a qualifier. The main steps in writing a select-project query are shown in Figure 1.

2.2.3. Database organization and data structures

In OBE, physical database design is automatically done by the system. All relations have the same inverted structure. The user only specifies the logical design including the option of declaring a key. This feature complies with the general design philosophy of the system: the database designer does not have to know about complex storage organizations, and does not have to decide whether an index should exist, much less choose between ISAM, hashed, or clustered/nonclustered B-Tree indices.

[1] Inverting the relations on every attribute is not hard-wired into the implementation. The optimizer does not assume that there is an index on every attribute and can handle other physical organizations of the relations [Wh85].

There are three basic data structures that we found to have significant impact on the performance of the system: **relation area**, **TD area**, and **pointer area**. These data structures have two features in common. First, when any one of these areas is created, they are created by a call to CMS that takes the size of the area as a parameter, and will only succeed if enough contiguous virtual memory is available to fill the request. Second, all pointers to an offset in an area requires 4 bytes.

(1) **Relation area:** All relations are stored in the same format, and all attributes are indexed. The storage scheme is intended to minimize the space required for a relation and its indices, while making both an index lookup and a sequential scan efficient. A relation is stored as an individual area[2] that constitutes a contiguous, relocatable unit of memory. A given value from a domain is stored only once, and a tuple descriptor, TD, is a sequence of pointers to domain values. Thus, within a single relation, the equality of two attribute values can be reduced to comparing pointers. Tuple descriptors form a double linked list, to facilitate sequential scans. An index is a linear array of pointers to tuple descriptors.[3] The implementors of OBE refer to a pointer to a tuple descriptor as a **TID**, short for tuple identifier. In Figure 2, the symbol ↑**TID** stands for a 4 byte pointer to a tuple descriptor. An **relation area** is shown in Figure 2.b that corresponds to the example relation "PAY" in Figure 2.a.

(2) **TD area:** A **TD area** has tuple descriptors and data exactly like a relation are, except that duplicate attribute values for the same domain are duplicated in the data. This means that comparisons of the same attribute in different tuples in the same **TD area** cannot be performed by comparing pointers in the TD's to the data. In addition, a **TD area** does not have any indices. Instead, a hash table used for duplicate elimination is at the bottom of a **TD area**. A hash table slot holds a pointer to the tuple descriptor, TD, of the tuple that hashed to that slot. A **TD area** cannot be queried like a relation. An example of a **TD area** containing all the tuples in the example relation "PAY" (Figure 2.a) is in Figure 2.c.

(3) **Pointer area:** A **pointer area** contains no tuple descriptors or data. It is always defined as corresponding to a single relation area. At the top of the segment is an array of pointers to TD's in a relation area. The pointer is the offset of a TD from the top of the **relation area** corresponding to the **pointer area**. At the bottom of the **pointer area** is a hash table used for duplicate elimination. A hash table slot contains pointers to the TD of the tuple in the **relation area** that hashed to that slot. A pointer area cannot be queried like a relation. An example of a **pointer area** defined relative to the **relation area** in Figure 2.b is in Figure 2.d.

In Section 4, we isolate the effects of these three data structures on performance by varying the output mode of queries. There are three options for the output mode of a query. An example of each output mode is in Figure 6, Section 4.2.3. Result tuples can be displayed by using the "P.", print, command in the desired attribute columns of the relation being queried. When this option is chosen, the result of the query is stored in a **pointer area**. The other two options are retrieval into user-created output or into a relation. User-created output, abbreviated UCO, is a table that is constructed interactively by entering literals and example elements in a skeleton table. UCO is used to format the result of a query. User-created output is not declared as, and cannot be queried like, a relation. When

[2] The designers of OBE refer to relations being stored in **relation segments** which should not be confused with segments in the underlying virtual memory system. A **relation segment** has the same layout as a **relation area**.

[3] Clearly any hierarchical index organization would have provided faster access and restructuring times, but this would have been at the expense of space. Also it should be noted that inserting an index entry can be performed with one **long move** instruction (Section 4.4), since the index is memory resident. Thus the benefit of main-

FIGURE 2. Data Structures

TUPLE #	ATTRIBUTES		
	NAME	AGE	RATE HOUR
1	Smith	25	$4.25
2	Hughs	50	$25.00
3	Michaels	31	$1.75
4	Jones	24	$53.47
5	Millman	76	$53.47

Figure 2.a The Relation "PAY"

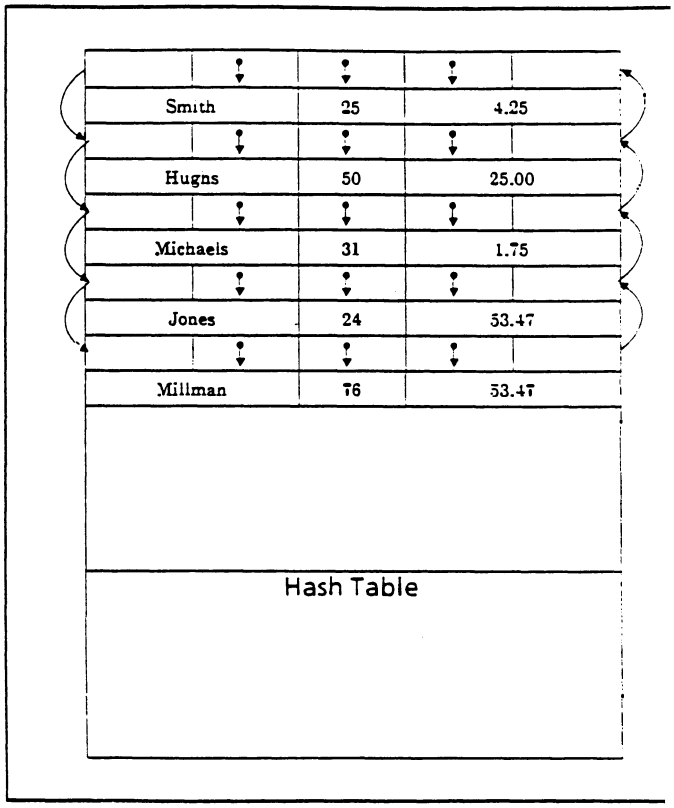

Figure 2.c TID Area Layout

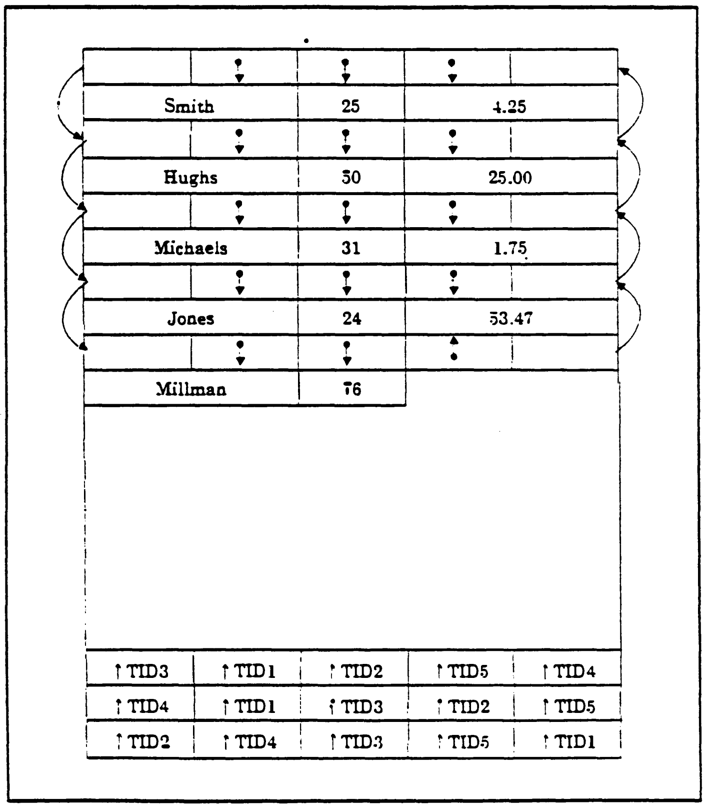

Figure 2.b Relation Area Layout

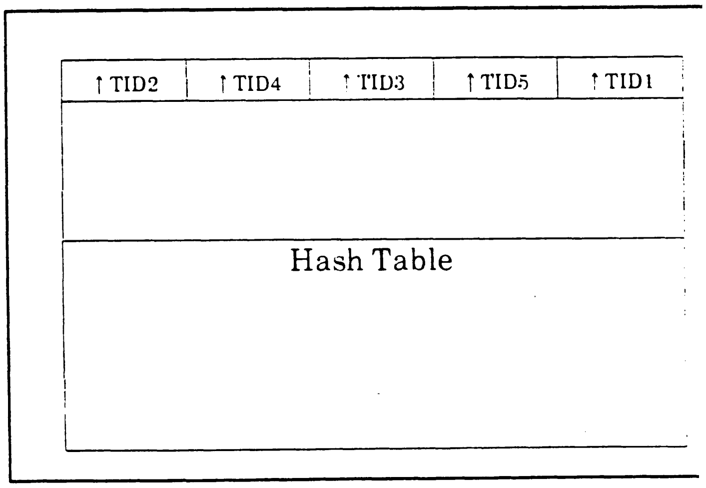

Figure 2.d Pointer Area Layout

the result of a query is retrieved into UCO, internally it is stored as a **TD area**. To retrieve the result of a query into the relation, the relation being retrieved into must be predeclared by filling in a relation definition table. Internally, the relation being retrieved into will be a **relation area**.

2.2.4. Access methods

Four basic access methods are implemented. The query optimizer [Wh85] uses a branch and bound algorithm to select a query processing plan that utilizes one or more of these access methods. The optimization phase precedes and is separate from the query processing phase.

(1) **Index-Lookup**: At the bottom of a relation area is an index for each attribute in the relation. Each index is a linear array of 4 byte pointers to tuple descriptors. The index is sorted on attribute value. It is a dense, nonclustered index. This index is compact. A 10,000 tuple relation would require 40,000 bytes for an index on one attribute and 640,000 bytes for an index on 16 attributes. Tuples can be accessed according to the value of 1 attribute through a very efficient binary search of the index. We call this search an equality lookup. Updating these indices is made more efficient through a **long move** instruction.

(2) **Simple-Scan**: Selections that involve a single relation can be performed by a very fast scan of the relation. A scan is accomplished by traversing the doubly linked list of TD's in the relation area. Pointers to the TD's that are selected are stored in an array which, if necessary for a join, can be sorted to become an **Temporary-Index**.

(3) **Temporary-Index**: A temporary index can be built on one or more attributes on the fly. First, the relation on which the index is being built is scanned using the **Simple-Scan** in order to apply all selection predicates to it. Pointers to all selected TD's (tuple descriptors) are collected in an array. The array is sorted on the join attribute(s) and is used as an index to perform joins.

(4) **Pipeline-Scan**: As part of OBE's pipeline approach to join processing [AHK85], a relation may be scanned a tuple at a time. As each tuple is processed, selections may be applied and the tuple is joined if possible, to qualifying tuples in other relations involved in the join. A tuple is eliminated from further consideration if it fails to join with at least 1 tuple from every other relation in the join.

There are two implementation decisions in this version of the prototype that should be mentioned. First, the **Index-Lookup** was never used to perform range selection. Second, single relation queries were treated as special cases of joins.

When analyzing our measurements in Section 4, we will discuss in detail these access methods and their respective costs and tradeoffs. In addition to the above access methods, sorting and hashing are employed to process lists of tuples formed during intermediate stages of query processing. However, the sort-merge join algorithm is never an option considered by the query optimizer. First, it would require too much space for storing temporary relations. Second, it may not compare favorably with other join algorithms when relations reside in main memory, as opposed to contiguous disk blocks. On the other hand, the query processor does utilize sorting as a building block in the processing of complex queries and for consistency checking in bulk updates. A heap sort [Kn73] is used to sort pointers in a **pointer area** for aggregate functions. The Power's sort [Po80] is an in-place merge sort that has very small storage overhead [Po80]. The Power's sort is used to sort TD's in **relation areas** and in **TD areas**, and to sort pointers for the **Temporary-Index**.

---

taining a B-tree like index is not as obvious as in the case of disk resident databases.

[4] The designers of OBE use a different terminology for their access methods and data structures [WAB86, Wh85].

Finally, when initially inserting tuples into a **TD area**, duplicate elimination on the tuple being stored is accomplished by hashing the entire tuple using the hash table at the bottom of the **TD area**. When initially inserting a pointer into a **pointer area**, duplicate elimination is accomplished by hashing on the attributes of the corresponding tuple that are projected by the query for which the **pointer area** is being created. OBE automatically performs duplicate elimination on the result of a query. Duplicate elimination is avoided where information about the key of the source relation allows the query processor to infer that no duplicates can possibly be introduced into the result.

3. BENCHMARKING METHODOLOGY

In this section, we justify the measures we use in evaluating OBE and describe how the selected measures were obtained. Section 3.1 delineates the measures of importance for MMDBS's. Section 3.2 details our timing techniques. Section 3.3 describes the database and queries we used to evaluate the performance of OBE.

3.1. Relevant Metrics

MMDBS's gain performance by using space, in the form of real and virtual memory, to save time. Thus, the performance of a MMDBS's must be quantified in terms of the **space-time product** [Bu76, De80]. The **space-time product** for a query is defined to be the average working set size during the execution of the query multiplied by the time required for the query. Without sophisticated tracing hardware and software, the **space-time product** for a query cannot be computed precisely [AEH84, AHK80, HL84]. However, there are more easily obtained measurements of space requirements that can be used in conjunction with precise timing of queries to provide a approximation to the **space-time product**.

The OBE prototype was designed under the assumption that all code and data necessary to carry out a query would fit in a user's virtual machine. That is, that a user would have enough virtual memory to accommodate the OBE module, program data space, the relations being queried and the result of a query. If this assumption was violated, OBE informed the user that insufficient virtual memory was available and aborted processing. Therefore, one relevant measure of the system's performance on any particular query is the minimum amount of virtual memory that the query can be processed in. For OBE, these measures are presented in Section 5.1.

Determination of the minimum amount of virtual memory required for any particular query is straightforward. Below the minimum, OBE aborts processing. However, determination of real memory requirements is not as straight forward. OBE assumes that enough real memory is available to allow the implementation to rely on the operating system in managing virtual memory. No assumptions are made concerning the algorithm used by the operating system for paging virtual memory. Therefore, once virtual memory requirements are determined, it is important to quantify the effects of paging on the performance of the system. This measure provides an indication of the amount of real memory required to provide acceptable performance. However, unlike virtual memory requirements, acceptable perfomance is no longer defined as the successful execution of a query. Rather, it is a value judgement that must be based based on cost and performance criteria [Gr85]. We present a controlled experiment for quantifying the effect of paging in Section 5.2.

When a MMDBS's space requirements are adequately met, the speed of the CPU becomes a dominant factor in the response time of a query. Therefore, measuring the effect of the CPU on the performance of a MMDBS is important. We present a comparison of the performance of OBE with two different CPU's, a 4341 and a 3081, in Section 4.1.

In addition, different data structures and algorithms used in query processing have different processing requirements. By systematically varying parameters of queries executed, we were able to design a benchmark that evaluated the processing requirements of the primary data structures and algorithms used in implementing OBE. These results are presented in Sections 4.2 through 4.6.

Finally, it must be acknowledged that there can be complex interactions between all of these measures: real and virtual memory requirements, paging, cpu speed, data structures and algorithms. As a first step, it is important to study these factors in isolation to provide a baseline for evaluating complex interactions between them. To this end, we conducted as many experiments as possible in single user, standalone mode. By single user, we mean that there was only one user of the DBMS active when we obtained our measures. This was also necessary because concurrency control was not implemented in OBE at the time we performed our experiments. By standalone, we mean that there were no other users on the machine when we performed our experiments.

Single user, standalone measures do not provide a realistic estimate of DBMS performance because they reflect the behavior of a DBMS in an underutilized resource environment. However, these baseline experiments are important for four reasons. First, measuring query execution time in single user mode is necessary in order to systematically evaluate schemes for physical database organization, or query optimization and execution algorithms. Second, the single user case constitutes a necessary baseline measure which must be used in the interpretation of multiuser experiments. Third, it is an indication of the best possible performance of a system with respect to a specific CPU and operating system. Finally, standalone measures provide an indication of the viability of an implementation in a workstation environment.

When it was not possible to obtain measures in single user, standalone experiments, we developed methods of approximating them. For instance, the standalone elapsed time for a query when no paging or I/O's occur during query execution, is well approximated by the cpu time required to execute a query. When we were unable to arrange standalone time to complete our experiments, we found that the **INDICATE USER** command [IBM83] provides a measure, **vtime**, which was a very good approximation to standalone elapsed time. Section 3.2 describes the measures we used.

3.2. Measurement Techniques

To measure the performance of the database functions of OBE, we wrote a benchmarking program that took the names of OBE query windows from an input file and obtained measures of interest before and after each query was executed. The query windows were two dimensional images of queries as they would be submitted to OBE from the screen. They were unparsed. The program executed as many queries as were in the input file by invoking the OBE query processor. We usually were only interested in measurements on the last query in the input file. At least two, and sometimes more than two, preparatory queries were always included in the input file to insure that the relations being queried and all OBE code were in virtual memory.

3.2.1. Query input

In order to respect the memory residence assumption, we wanted to have the relations queried memory resident before timing a query. In every experiment, the operand relations were read into virtual memory before a query was executed. This was easily accomplished with the implementation of OBE we tested.

In order for an interactive user to enter a query, the user has to press a function key to get a skeleton query table on the screen. An example of a skeleton query table is in Figure 1.a, (Section 2.2.2). The user then obtains a relation template for the relation to be queried. To display a template for the relation "PAY", a user would enter "P. PAY" in the skeleton query table, as shown in Figure 1.b (Section 2.2.2), and press <PROCESS>.

When a relation template is invoked in this manner, the entire relation is brought into virtual memory by OBE. An example of a relation template is shown in Figure 1.c (Section 2.2.2). Therefore, before timing a query, we preceded it in the input file with a query (or queries) that invoked a relation template thereby bringing the relation(s) to be queried into virtual memory.

In addition, we found that when the first query that actually required database processing was submitted, 3 **start I/O**'s always occurred. These **start I/O**'s apparently brought in some code or data that was needed to process queries. Since invoking relation templates did not evoke these 3 **start I/O**'s, we created a empty dummy relation with 1 numerical attribute and created a window, query **InitParse** shown in Appendix 3, that performed a range query on the dummy relation. Query **InitParse** did evoke the 3 **start I/O**'s.

By preceding the query to be timed with the proper intializing queries, the queries we timed never invoked **start I/O**'s. Thus, we conclude that all data required to execute the queries was in virtual memory. When run standalone with sufficient real memory, the queries also never caused paging.

We executed the benchmarking program once per input file. Each input file contained only one query of interest preceded by the necessary initializing queries. Our decision to execute only one query of interest per program is in contrast to our approach with DDBS's [BDT83] where we usually executed 10 queries at a time and then averaged the total to get the time for one query. For instance, to perform a selection we would choose 10 selection predicates using a table of random numbers and then execute the 10 different selections one after another. We did this to randomize the location of the tuples in the relation that would be accessed. This would reduce the probability that they would be contiguous on disk, thus reducing seek times, and to reduce the probablilty that the pages they resided on would already be in memory as an aftereffect of a previous query. Since, with OBE, the relation was guaranteed to be in memory, the location of tuples accessed was unimportant. The one exception to this case occurs in our paging experiment and it is dealt with in Section 5.2.

We also chose to execute only one query of interest per program to control the effects of fragmentation of virtual memory. In OBE, certain data structures could only be created if there was enough contiguous virtual memory to allocate to the structure. (See Section 2.2.3.) If, due to fragmentation, no contiguous blocks that are large enough exist, OBE would fail and tell the user to reinitialize, IPL, his virtual machine. While we did not quantify this effect, we found that the longer we used the system interactively in one continuous session, the more likely this kind of failure was. Furthermore, as fragmentation occurs, it is likely that memory allocation will take longer. By executing only 1 query of interest per program, we measured the performance of each query under similar conditions with respect to memory management.

3.2.2. System calls

At the beginning and end of the benchmarking program, were calls to the **INDICATE** command [IBM83] that provided us with general measures of the system's load such as CPU utilization. Just before each query was executed, 2 measures were obtained in the following order:

(1) **INDICATE USER** - a system call that provided monitor information about our virtual machine such as the amount of CPU time, machine, the number of paging reads and writes, and the number of **start I/O**'s charge to our virtual machine.

(2) A call to the TOD clock [IBM83] that we used to obtained elapsed times measured in microseconds.

Immediately after the execution of a query these two measures were obtained in reverse order. The difference between these cumulative measures taken just before and just after a query was executed provided us with a measure of the time and resources required by the query.

3.2.3. Approximating standalone elapsed time

One of our primary measures of OBE's performance was the elapsed time required for a query in the absence of paging, I/O, and competition with other users for cpu time. This differs from our approach in other experiments [BDT83, BT84, BT85] only in that the standalone elapsed time we use excludes paging and I/O's unless otherwise stated. We excluded paging and I/O in order to obtain a baseline measure of the best possible performance of a query with a particular CPU.

In a preliminary standalone test, we found that the measure **vtime**, obtained from the **INDICATE USER** command, for a query on a moderately loaded system was a very good approximation for the query's standalone elapsed time. In fact, over a crossection of 20 queries, the **vtime** always came within 4 percent of the standalone elapsed time. We repeated the experiment on the 3081, with one standalone run and one run when the system was loaded. During both runs, the VM monitor collected data. We used the monitor data from the loaded run to give us some quantification of system load for which this approximation of nonstandalone **vtime** to standalone elapsed time was reliable. In this paper, we refer to **vtime** as **CPUtime** because **vtime** is a measure of the CPU time a virtual machine uses minus any privileged instructions performed for that virtual machine by the operating system, CP. In particular, the CPU time used by CP to perform paging and I/O for the virtual machine is not included.

The experiment consisted of 20 queries, a mix of selections, projections, aggregates and aggregate functions, insertions and joins. Only three of these queries were identical to queries used in our preliminary test. Each different query was run 4 times in the standalone run and 6 times in the nonstandalone run. For each query we computed:

standalone elapsed time ≡ **average elapsed** time from the standalone run

and

loaded CPUtime ≡ **average CPUtime** from the nonstandalone run.

The results are contained in Table 1. The data confirmed our preliminary result. For each query we computed the absolute value of the difference between the **loaded CPUtime** and the **standalone elapsed time**, |ET - CPUtime| in Table 1. The average difference, over all queries was .04 seconds. The maximum difference for a single query was .180 seconds for query **MinFn-P**, an aggregate function that took 37.157 seconds **standalone elapsed time** and 36.977 seconds **loaded CPUtime**. For this query, the **loaded CPUtime** was within .5 percent of the **standalone elapsed time**.

We also computed the percent error relative to the **standalone elapsed time**:

% ERROR≡|standalone elapsed time - loaded CPUtime / standalone elapsed time| * 100

Averaging over all queries, the **loaded CPUtime** was within 1.2 percent of the **standalone elapsed time**. The greatest discrepancy came with query **JoinAB'**, a three way join that had a **standalone elapsed time** of .479 seconds and a **loaded CPUtime** of .461 seconds. Thus, the **loaded CPUtime** was within 3.9 percent of the **standalone elapsed time**.

It is probable that the execution of the VM monitor during our standalone run slightly increased the **standalone elapsed time** of our queries [Po85]. However, this effect could not serve to enhance our results as the **standalone elapsed** time for 17 out of the 20 queries was higher than the **loaded CPUtime**. Therefore, our claim that **loaded CPUtime**

Table 1: Relationship Between
Standalone Elapsed Time and Nonstandalone CPUtime

Where rounding made numbers seem inconsistent, we added 1 more decimal place. The original data was recorded to 7 significant digits.

QUERY	ELAPSED TIME	CPUtime	|ET - CPUtime|	% ERROR
Selectkey-Eq	0.147	0.145	0.002	1.384
Select1-Eq	0.147	0.145	0.002	1.409
Select10-Eq	0.168	0.167	0.002	0.927
JoinAB'	0.479	0.461	0.019	3.877
Select10-Rg	0.597	0.592	0.005	0.755
JoinCselAselB	0.6703	0.6701	0.0002	0.038
Select100-Rg+	0.9995	0.9884	0.0111	1.113
Join2	1.285	1.294	0.009	0.686
Join3	1.718	1.734	0.016	0.918
Join4	2.159	2.186	0.027	1.258
Select10-RgC	2.442	2.421	0.021	0.849
JoinAselB	2.854	2.823	0.031	1.091
Select10-Rg2N	2.917	2.878	0.039	1.349
Select10-Rg2D	2.956	2.892	0.064	2.166
Select10-Rg2	2.975	2.920	0.055	1.855
Insertkey	4.236	4.190	0.046	1.092
Insertall	4.241	4.204	0.037	0.861
Project20-P	10.357	10.290	0.067	0.646
MinKey-P	15.804	15.638	0.166	1.048
MinFn-P	37.157	36.977	0.180	0.485
AVG	4.716	4.681	0.040	1.190
MAX	37.157	36.997	0.180	3.887
MIN	0.147	0.145	0.0002	0.038

is within 4 percent of **standalone elapsed time** is, if anything, conservative.

According to VM monitor data, during the hour in which the data used to compute **loaded CPUtime** was collected, 222 users were logged on. However, only 22 users were ever active during the monitored interval. The average number of users in queue for either CPU[5] was 14. On the average, CPU utilization was 146%. During 53% of the total available CPU time (200% for 2 CPU's), one or both of the CPU's were idle but at least one user was waiting for I/O.

The amount of real memory available for paging was 56,764K. The average paging rate was 27 reads/writes per second. 1.7% of all pages chosen to be paged out of memory were taken from users in queue to use a CPU. The virtual I/O rate was 120 **start I/O's** per second. The diagnose I/O rate was 86 **start I/O's** per second.

Our experiment confirmed that the average **CPUtime** for a query on a moderately loaded system can be used as an approximation to the average standalone elapsed time for the query, when the query, run standalone, requires no paging or I/O. To be conservative, we assume that the **loaded CPUtime** is within 4 percent of the **standalone elapsed time**.

[5] A 3081 is a dual processor.

Throughout the remainder of this report, the measure labeled as **CPUtime** can be assumed to be the average **CPUtime** for a query on a moderately loaded system unless otherwise stated.

3.3. The Test Database and Queries

We used a test database consisting of relations that had been used in benchmarks of several DDBS's [BDT83, BT84, BT85, BD84]. In Section 3.3.1 we describe the database we used. Initially, we also used queries on these relations that we had benchmarked other systems with. While these queries were useful in making comparisons with DDBS's, they were inappropriate for an interactive system and for OBE's two-dimensional interface. In Section 3.3.2 we describe both the crossection of interactive queries we developed and some of the subtleties in query phrasing that became important due to the two-dimensional interface. (See Appendix 2).

3.3.1. The database

We used a test database consisting of 5 synthetic relations. Each relation has 16 attributes, 13 integer attributes and 3 string attributes. This relation is very wide. When all 16 attributes are printed, they will not fit on 1 line of a standard lineprinter. They also would not fit in a single screen using the OBE interface. We include 16 attributes in the synthetic relations to allow us to write a variety of queries on the same relation with the desired parameters, (e.g. result size, number of tuples that join). With each query, we are only interested in a subset of the available attributes, much like the view a user may have of a particular relation.

In OBE, all numerical values are stored as variable length floating point numbers. All strings are stored as variable length strings. The relations, including statistics on their instantiation in OBE, are summarized in Table 2.

The largest relation, **TenK**, had 10,000 tuples, 16 attributes per tuple. It occupied 3.5 megabytes but could be read in from disk in 2.47 seconds. It was stored on disk as a file with one 3,465,216 byte record. Storing the relation as one record is the optimization that allowed it to be read into virtual memory so quickly.

The relation **TenK** is the relation that we used for the majority of our queries. Of the 3.5 megabytes it occupied, the indices alone required 18 % of this space:

(10,000 tuples) * (16 attributes/tuple) * (4 bytes/attribute) = 640,000 bytes

Table 2: A Description of the Synthetic Relations
Used in Benchmarking OBE

Relation is stored as a Relation Area (including indices on every attribute)
Elapsed Time is Standalone Time Required to Read in Relation
From IBM 3380 Disk Using IBM 3880 Channel
(Elapsed Time is in Seconds, Blocks are 4096 Bytes)

RELATION NAME	NUMBER of TUPLES	NUMBER of BYTES	NUMBER of BLOCKS	ELAPSED TIME to READ from DISK
OneK	1,000	360,448	89	0.30
TwoK	2,000	712,704	175	0.58
FiveK	5,000	1,757,184	430	1.29
TenK	10,000	3,465,216	847	2.47

The integer attributes in the synthetic relations were all uniformly distributed within specified ranges so that controlling the size of the result of a query would be possible. The range of the integer attribute is implied by its name. For instance, the attribute two has two distinct values, 0 and 1. The attribute thousand has one thousand distinct values between 0 and 999 inclusive. The attributes in relation **TenK** are described in Table 3.

The nonunique values, whether strings or integers, can be used for partitioning in aggregate functions, varying the number of duplicate tuples in a projection and for modelling 1:N or M:N joins. Either unique or nonunique values can be used for controlling the selectivity of selections. Unique values can be used for modelling 1:1 joins. A unique value that is declared to be the key can be used to test selections, projections and updates on key attributes.

Each of the 3 string attributes is 52 characters long with 3 significant characters. The significant characters occur in positions 1,26 and 52 in the string. The remainder of the positions contain the same padding character. The attributes **stringu**1 and **stringu2** are candidate keys that had values ranging from:

Axxx . . . xxx**A**xxx . . . xxx**A** to **V**xxx . . . xxx**V**xxx . . . xxx**T**

With our unique string attributes, the leftmost significant character was varied most frequently, followed by the middle significant character. We did this to provide a mechanism for evaluating any short-circuit comparison algorithms or hardware.

The attribute **string**4 assumes only 4 distinct values:

Axxxxxxxxxxxxxxxxxxxxxxxx**A**xxxxxxxxxxxxxxxxxxxxxxxx**A**
Hxxxxxxxxxxxxxxxxxxxxxxxx**H**xxxxxxxxxxxxxxxxxxxxxxxx**H**
Oxxxxxxxxxxxxxxxxxxxxxxxx**O**xxxxxxxxxxxxxxxxxxxxxxxx**O**
Vxxxxxxxxxxxxxxxxxxxxxxxx**V**xxxxxxxxxxxxxxxxxxxxxxxx**V**

Table 3: Description of the Attributes in Relation TenK
10,000 Tuples in Relation
All Integer Attributes Uniformly Distributed
All Strings 52 Characters Long with 3 Significant Characters

NAME	TYPE	CARDINALITY	RANGE	ORDER	COMMENT
unique1	INT	10,000	0 - 99,999	random	candidate key
unique2	INT	10,000	0 - 99,999	random	declared key
two	INT	2	0-1	alternating	0,1,0,1,...
four	INT	4	0-3	alternating	0,1,2,3,0,1...
ten	INT	10	0-9	alternating	0,1,...,9,0,...
twenty	INT	20	0-19	alternating	0,1,...,19,0,...
hundred	INT	100	0-99	alternating	0,1,...,999,0,...
thousand	INT	1000	0-999	random	
twothous	INT	2000	0-1999	random	
fivethous	INT	5000	0-4999	random	
tenthous	INT	10,000	0-9999	random	candidate key
odd100	INT	50	1-99	alternating	1,3,5,...,99,1,...
even100	INT	50	2-100	alternating	2,4,6,...,100,2,...
stringu1	CHAR	10,000	A...A...A-V...V...T	random	candidate key
stringu2	CHAR	10,000	A...A...A-V...V...T	alternating	candidate key
string4	CHAR	4	A...A...A-V...V...V	alternating	

The smaller relations had the same attributes with identical ranges and cardinalities. When the number of tuples in a relation precluded an attribute from having all of the integer values within a the specified range, the attribute was given consecutive values starting at either the beginning or the end of the attribute's range. For instance, the relation **OneK** could not have 10,000 different values for the attribute **tenthous**. In this case, the attribute was given one thousand unique values in the range 0 to 999.

In creating a test database, we sometimes create more that one copy of the same relation. In the OBE test database, we had two copies of the relation **TenK** which we called **TenKe** and **TenKf**. We also had two copies of the relation **TwoK** which we called **TwoKb** and **TwoKc**. The attribute names in each relation are each appended with a letter to make all of the attribute names in the database unique. The letter appended to distinguish two copies of the same relation corresponds to the letter appended to distinguish their attribute names. For instance, the keys of relations **TenKe** and **TenKf** are **unique2e** and **unique2f**, respectively. In one-dimensional languages, this naming protocol prevents conflicts in attribute names when corresponding attributes from different relations are retrieved in a join. The OBE interface made this protocol unnecessary. For figures in this paper, we retained the protocol in examples of joins to allow easy translation of OBE queries to one-dimensional languages.

3.3.2. The test queries

The structure of our test database enables us to generate a comprehensive set of relational queries, with systematic control of the size of the result. In the evaluation of a DBMS, our first goal is to establish baseline measures for a relational instruction set. We achieve this by holding the size of the operand relation and the size of the result constant, and executing each of the 8 query types: Selection, Projection, Deletion, Insertion, Update, Simple Aggregate, Aggregate Function, Join. This is a nontrivial task as options in physical design, logical design, and query phrasing can multiplicatively expand the number of queries required to establish a relational instruction set. Reducing this number to less than a product of all of the possibilities is an art to which we address ourselves in this paper.

In benchmarking traditional DBMS's, we have always had a variety of physical designs to consider. In [BDT83], each type of query was performed on key and nonkey attributes with variations on the physical design: clustered index on the key, nonclustered indices on nonkey attributes, relations stored as heaps. For each type of query on each system, certain physical designs were optimal and others resulted in extremely poor performance.

In OBE, there is only one physical organization. All attributes have a nonclustered, dense index on them. The only choice the designer has is in the design of the relational schema. The designer may also specify a key attribute(s), although it is assumed that most users of the system will neither know, nor care, what a key is. Thus, the performance of the system on any particular query is both its best and its worst case performance.

We initially benchmarked OBE using queries which we had used in benchmarking other relational DDBS's. While these queries were useful in initially making comparisons between systems, we quickly came to the conclusion that they were inappropriate for an interactive system with a two-dimensional interface. (See Appendix 2.) We settled on a set of parameters for a **standard interactive query**, and varied them systematically to isolate parameters that affected the response time of queries. We consider the following to be reasonable parameters for a **standard interactive query**:

(1) Number of Tuples in the Result - 10. OBE actually only formats and presents the first 8 tuples in the result to the user, along with a message saying how many tuples are in the result. Using a function key, the user may expand the result table to view more of the result tuples.

FIGURE 3. A *Standard Interactive* Join

JoinAselB

TENKE	UNIQUE1E	UNIQUE2E
	_A	_Y

CONDITIONS
_X = _Y _X < 10

TENKF	UNIQUE2F	TENF	THOUSANDF	STRING4F
	_X	_B	_C	_D

UCO	UNIQUE1	UNIQUE2	TEN	THOUSAND	STRING4
P.	_A	_Y	_B	_C	_D

FIGURE 3.a JoinAselB: A *Standard Interactive* Join

UCO	UNIQUE1	UNIQUE2	TEN	THOUSAND	STRING4
	7095	0	6	267	OxxxxxxxxxxxxxxxxxxxxxxxxxOxxxxxxxxxxxxxxxxxxxxxx
	4470	1	6	790	AxxxxxxxxxxxxxxxxxxxxxxxxxAxxxxxxxxxxxxxxxxxxxxxx
	4008	2	0	816	AxxxxxxxxxxxxxxxxxxxxxxxxxAxxxxxxxxxxxxxxxxxxxxxx
	9033	3	7	545	HxxxxxxxxxxxxxxxxxxxxxxxxxHxxxxxxxxxxxxxxxxxxxxxx
	2873	4	9	61	VxxxxxxxxxxxxxxxxxxxxxxxxxVxxxxxxxxxxxxxxxxxxxxxx
	5674	5	0	338	OxxxxxxxxxxxxxxxxxxxxxxxxxOxxxxxxxxxxxxxxxxxxxxxx
	9388	6	0	701	AxxxxxxxxxxxxxxxxxxxxxxxxxAxxxxxxxxxxxxxxxxxxxxxx
	422	7	1	667	VxxxxxxxxxxxxxxxxxxxxxxxxxVxxxxxxxxxxxxxxxxxxxxxx
	425	8	0	684	OxxxxxxxxxxxxxxxxxxxxxxxxxOxxxxxxxxxxxxxxxxxxxxxx
	4343	9	2	810	OxxxxxxxxxxxxxxxxxxxxxxxxxOxxxxxxxxxxxxxxxxxxxxxx

FIGURE 3.b Result from JoinAselB in User-Created Output Table
(A *Standard Interactive* Result)

(2) Number of Attributes in the Result - 5. Four integers, including the key, and one string. The default attributes were: **unique1, unique2, ten, thousand** and **string4**.

(3) Method of Outputting the Result - "P." on a Query Table. This is the most convenient way of printing the desired tuples or attributes. In the case of joins, the default was retrieving into user-created output because there is no way to juxtapose tuples that joined from different relations using a "P.". Figure 3.a is an example of a two-way join that shows the use of user-created output to juxtapose attributes from different relations. Figure 3.b shows the result of the join which has the same attributes and same number of tuples as the default for a **standard interactive query**.

(4) Number of Relations in a Query - 1, when possible. We made this choice in order to isolate the effects of different data structures and algorithms on queries other than joins.

(5) The Size of the Relation Queried - 10,000 tuples, 3.5 megabytes. We used the relation **TenK** in all queries. Actually, **TenK** is larger than the typical relation for which the OBE prototype was designed. The designers expected most users to be able to use OBE with only 4 megabytes of virtual memory available to their virtual machine. However, our emphasis was on the evaluation of a prototype, not in providing example response times for a typical user. Large relations helped us isolate the factors that influenced performance and tested the robustness of the implementation.

(6) The Size of the Virtual Machine - 16 Megabytes of Virtual Memory. This was the maximum allowed to an ordinary user. We did this in order to run all queries in a similar virtual memory environment, thereby eliminating a possible confounding variable in comparisons between queries.

In Section 4, as we analyze the query processing algorithms used by OBE, we will explain how we varied query parameters to isolate the effects of the data structures and algorithms used by OBE.

3.3.3. Query phrasing

In the analysis of our initial experiments, we found that we could not understand the effects of some queries without analyzing the influence of the two dimensional interface. As with all other query languages, there are many alternate ways to phrase the same query. Where possible, we chose a presentation that did not confound the cost of processing the query with its method of input. There were two striking cases where the cost of query phrasing could not be ignored. One occurred in our analysis of updates where parsing time became important, the other occurred with our comparison of simple and complex selection predicates, where the condition box became important.

3.3.3.1. Parsing time

When preparing a query in OBE, the first action the user takes is to invoke a relation template as shown in Figure 4.a. The screen then displays a relation template that is used to input the query. The template for the relation **TenK** is shown in Figure 4.b. Depending on the query, some of the attributes in the template may be irrelevant. For example, query **Update1-key+** in Figure 4.c. has fifteen irrelevant attributes in the query template that must be parsed. **Update1-key+** updates the key of 1 tuple from the value 75 to the value 10075. **Update1-key**, in Figure 4.d, is the same query with the irrelevant attributes deleted from the screen. This is easily done with one of the function keys defined by OBE. Times for these two queries are summarized in Table 4.

The average CPUtime for executing **Update1-key+** was 4.19 seconds. The average CPUtime for executing **Update1-key** was 3.67 seconds. Once parsed, there would be no difference in the execution of the query, thus we concluded that the system took .51 seconds to parse the additional 15 attribute templates in the screen. Consequently, when comparing

FIGURE 4. Query Phrasing: Parsing Time

Init10K

P. TENK			

FIGURE 4.a Invoking Template For Relation TenK

Update1-Key

TENK	UNIQUE2
	75 u. 10075

FIGURE 4.d Update1-Key
(Superfluous Attributes Removed)

TENK	UNIQUE1	UNIQUE2	TWO	FOUR	TEN	TWENTY	HUNDRED	THOUSAND	TWOTHOUS	FIVETHOUS	TENTHOUS	ODD100	EVEN100	STRINGU1	STRINGU2	STRING4

FIGURE 4.b Template for Relation TenK

Update1-Key+

TENK	UNIQUE1	UNIQUE2	TWO	FOUR	TEN	TWENTY	HUNDRED	THOUSAND	TWOTHOUS	FIVETHOUS	TENTHOUS	ODD100	EVEN100	STRINGU1	STRINGU2	STRING4
		75 u. 10075														

Table 4: Impact of Parsing
Average CPU Time in Seconds on Loaded System

QUERY	CPUtime	COMMENT
Update1-key+	4.19	15 superfluous attributes in query table
Update1-key	3.68	only 1 attribute in query table

queries, we have to be aware that queries containing more or larger objects on the screen will take significantly longer to parse.

3.3.3.2. The condition box

In phrasing range selection queries, it was possible to express one condition, (i.e. **unique2** < 10), without using a condition box. The same query can be expressed using the condition box. Query **Select10-Rg**, which selects 10 tuples, has one condition on the key, **unique2**. When phrased without using the condition box, as shown in Figure 5.a, **Select10-Rg** required an average of .59 seconds of cpu time. Phrased with the condition box, as shown in Figure 5.b, the same query required 2.42 seconds of cpu time. The difference of 1.83 seconds for the same query must be due to additional processing done both in the parsing and query processing stages of query execution.

When a conjunction of two conditions on one attribute is expressed, the conditions must be expressed in the condition box. Query **Select10-Rg2**, shown in Figure 5.c, selects 10 tuples using the two conditions:

(**unique2** > 5838) and (**unique2** < 5849)

Select10-Rg2 required an average of 2.92 seconds of cpu time. The results are summarized in Table 5.

The difference in **CPUtime** required for queries **Select10-Rg** and **Select10-Rg2** should give us the difference between having to make one comparison to evaluate a tuple in a range query and having to make two comparisons. However, this difference is 2.33 seconds if **Select10-Rg** is phrased without the condition box and 0.40 seconds if **Select10-Rg** is phrased with the condition box. Consequently, when testing effects of parameters other than the condition box, we only compare queries that either both have or do not have the condition box.

Table 5: Impact of Condition Box
Average CPU Time in Seconds on Loaded System

QUERY	CPUtime	COMMENT
Select10-Rg	.59	one condition, no condition box
Select10-Rg	2.42	one condition, with condition box
Select10-Rg2	2.92	two conditions, with condition box

FIGURE 5. Query Phrasing: The Condition Box

Select10-Rg

TENK	UNIQUE1	UNIQUE2	TEN	THOUSAND	STRING4
P.		< 10			

FIGURE 5.a Select 10 Tuples on Key, 1 Condition, No Condition Box

Select10-Rg

TENK	UNIQUE1	UNIQUE2	TEN	THOUSAND	STRING4
P.		_X			

CONDITIONS
_X < 10

FIGURE 5.b Select 10 Tuples on Key, 1 Condition, With Condition Box

Select10-Rg2

TENK	UNIQUE1	TENTHOUS	TEN	THOUSAND	STRING4
P.		_X			

CONDITIONS
_X > 5838 _X < 5849

4. TIMING MEASURES

In this section, we analyze the time requirements of OBE, under the assumption that the space, virtual and physical main memory, requirements are met. In order to establish a preliminary measure of CPU utilization, we started with a small set of cross-section queries for which we had timing measurements on other DBMS's, and timed these queries in standalone mode on both the 3081 and the 4341. We then designed a comprehensive set of benchmark queries, and measured their **CPUtime** on a moderately loaded 3081 system. We divide this section into six subsections. In the first subsection, we describe and analyze our standalone measurements of the cross section queries on the two machines. Then, we devote one subsection to each of the relational operations (selection, projection, join), one for aggregates (simple aggregates and aggregate functions), and one for updates. For each type of query, we first describe the parameters that may have an impact on the performance of the query. These include parameters that define a relational query profile, such as complexity of the predicate for selection queries or the proportion of duplicate tuples created by a projection, and parameters that are derived from options provided by the OBE interface, such as the use of a condition box or the format of the output. Modelling these parameters required us to design a number of different test queries for each relational operation. In each of the five subsections, we describe these test queries, present our timings, and quantify the impact of each performance factor.

4.1. MIPS Rate

On a 3081 and a 4341, we measured and compared the standalone execution times for a cross section of queries including selections, projections, updates, joins, and aggregates. The same queries were previously benchmarked on a number of disk database systems ([BDT83, BT84, BD84, BT85]), which included both software systems running on a VAX host and database machines. See Appendix 2.

Table 6: Query Execution Time in Seconds on IBM 3081 and IBM 4341
Default Result Consists of All Attributes From Operand Relation(s)
Default Output Mode: "P." on Query Table

Query Name	Time on 3081	Time on 4341	Ratio 4341/3081	Query Description
Select1-Eq+	0.5	3.1	6.2	Select 1 tuple
Select100-Rg+	1.3	7.6	5.8	Select 1% of the tuples
Select1000-Rg+	1.6	10.3	6.4	Select 10% of the tuples
Select1000-Rg+	4.7	34.7	7.4	Select 10%, with condition box
Project100	14.3	106.8	7.5	Project 100 tuples, 6 attrs.
ProjectAll	1.0	6.9	6.9	Project all of **OneK**
JoinAB'+	2.6	18.4	7.1	2-way join
JoinAselB+	5.6	37.8	6.8	1 selection, 2-way join
JoinCselAselB+	4.8	30.2	6.3	2 selections, 3-way join
Update1+	4.7	26.2	5.6	Update one tuple
Insert1	2.4	24.1	10.0	Insert one tuple
Delete-Eq+	0.4	2.5	6.3	Delete one tuple
MINIMUM	0.5	3.1	5.8	
MAXIMUM	14.3	106.8	10.0	
AVERAGE	3.6	25.7	6.9	

For this experiment, we made available to OBE the amount of physical main memory required to keep both the database system's code and the operand relations in core. A subset of our measurements is presented in Table 6. A brief description of the queries appears to the right of the table. The 10,000 tuple relation, **TenK**, was included in every query except **ProjectAll**, which was a projection of the relation **OneK** performed without duplicate elimination.

Our goal was to verify whether the ratio between the elapsed time of the same queries on the 4341 and the 3081 would be different from the known ratio between the MIPS rates of the two machines. For the numbers in Table 6, the average ratio was 6.9. The ratio that is usually quoted between MIPS rates for the 4341 and 3081 is 7. This result is an indication that a general CPU performance metric is a strong basis for predicting the effect of CPU speed on the performance of OBE. This distinguishes OBE from conventional, disk database management systems.

4.2. Selections

A number of factors determine the speed at which a selection query can be processed by a database system. In a conventional DBMS, the storage structure of the operand relation is one of them, and performance will widely vary depending on the structure chosen by the database designer. Since in OBE all relations have the same storage structure, the performance of selections only depends on the query profile. We examined the impact of three parameters that model this profile:

(1) Query Input Format: with or without condition box

(2) Selection Predicate:
(a) Selectivity: number of tuples selected
(b) Range versus equality selection
(c) Simple (**attr <op> const**) versus composite predicate (boolean combination of simple predicates)
(d) Key versus Nonkey Attributes specified in selection predicate, or in result

(3) Query Output Mode: "P." on query table, retrieving into user-created output, or retrieving into a predeclared relation

4.2.1. Query input format

With a simple selection predicate, the OBE interface provides two options to express the selection: within the table skeleton, or separately, in a condition box. In order to establish a baseline measurement with these two options, we started with timing a **standard interactive query** for which the output fits well on the screen, query **Select10-Rg**. Query **Select10-Rg**, phrased with and without the condition box, is shown in Figure 5, Section 3.3.3.2. This test query has a **standard interactive result**: 10 tuples, 5 attributes per tuple. (See Figure 3.b, Section 3.3.2.) Timings for this query expressed with and without a condition box are presented in Table 7. Without a condition box, the selection required 0.59

Table 7: Times for the Baseline Selection

Average CPU time in Seconds on Loaded System
(10 tuples in result, Output Mode: "P." on query table)

QUERY	TYPE	CPUtime	PREDICATE
Select10-Rg	Range Selection	.59	Key < 10, no condition box
Select10-Rg	Range Selection	2.42	Key < 10, with condition box

seconds, which is very fast. By comparison, a disk database system using a secondary B-tree index can perform this query in between 1 to 2 seconds [BDT83]. On the other hand, the selection phrased with a condition box took 2.42 seconds, almost 5 times longer. The difference of 1.81 seconds is accounted for by two factors. First, the time to parse the query was slightly increased by the presence of the condition box. Second, when the query was expressed with a condition box, the optimizer chose a different access method, which was not efficient for this simple range selection. We investigated this anomaly further, by checking precisely how both queries were executed.

In both cases, the optimizer chooses to scan the entire relation, rather than using the index, because the query is a range selection. However the first case, when the query was phrased without the condition box, the **Simple-Scan** access method was used. In the second case, the **Pipeline-Scan** was chosen. The second choice was made because the query optimizer treats a single relation query as a special case of a join. Clearly, the **Pipeline-Scan** code incurred an unecessary overhead in this particular case.

4.2.2. Selection predicate

In order to test the impact of selectivity on selections, we timed equality selections that retrieved 1 and 10 tuples, and range selections that retrieved 10 and 100 tuples (out of 10,000 tuples in the operand relation). Where possible, we also varied whether the selection was on a key attribute. Clearly, we could not have an equality selection on the key that retrieved more than one tuple. The key, unique2, was included in the result, and therefore, duplicate elimination was not done on the result. Selections that did not use the condition box are presented in Table 8.

The first three queries in Table 5 were equality selections that retrieved one, one and ten tuples respectively. Only **Select1-key** was a selection on the key. Comparing **Select1-key** and **Select1**, both required an average of 0.15 seconds of CPU time, indicating that the selecting on the key had no effect. The next query, **Select10-Eq**, which retrieved 9 more tuples, required an average of 0.16 seconds of CPU time, 0.01 seconds more than **Select1-Eq**. This difference is not remarkable. Query Select10-Eq can be found in Appendix 3. Queries **Select10-Rg** and **Select100-Rg** were range selections that retrieved 10 and 100 tuples respectively. Comparing **Select10-Eq**, an equality selection that retrieved 10 tuples, and **Select10-Rg**, a range query that retrieved 10 tuples, we get the cost of scanning the relation to be approximately about .4 seconds. Comparing the range queries **Select10-Rg** and **Select100-Rg**, we get the cost of increasing the selectivity of the query from 10 to 100 tuples is only .03 seconds.

The lack of effect of selectivity on the times for these queries is due to their method of output. When the result is output through a "P." on the query table, the qualifying tuples

Table 8: Times for Range and Equality Selections with Selectivity Varied

No Condition Box, Output Mode: "P." on Query Table

QUERY	TYPE	CPUtime	RESULT SIZE	PREDICATE
Select1-key	Equality	0.15	1	Key=838
Select1-Eq	Equality	0.15	1	candidate key=838
Select10-Eq	Equality	0.16	10	thousand=838
Select10-Rg	Range	0.59	10	Key < 10
Select100-Rg	Range	0.62	100	Key < 100

are not materialized in a result relation. Rather, the result is stored in a **pointer area**, which means that the result consists of an array of pointers to qualifying tuples in the operand relation. In addition, because the key attribute was included in the result tuple, no duplicate elimination was done. Finally, regardless of the method of output, only 8 tuples at a time are formatted for display to the interactive user. At the bottom of the result table that displays the first 8 result tuples, is a message informing the user of the total number of tuples in the result. The user may view additional result tuples by using a function key to expand the result table. Thus, as the selectivity of a query increases, the time to format the result tuples does not - unless the user explicitely asks to see more output. It is interesting to note that the trick of formatting only a few tuples at a time not only decreases the interactive response time, it also saves an interactive user from being bombarded with screens full of unwanted information when the result is much larger than expected.

Using range selections that included the condition box, we tested the impact of the predicate complexity, by adding an inequality to define the range. We tested the effect of selecting on the attribute explicitely declared as a key versus selecting on a candidate key. We also tested the effect of duplicate elimination when the result had ten tuples. These experiments are summarized in Table 9.

In Table 9, the most striking result is the difference between **Select10-Rg**, a range query on the key requiring 1 comparison and **Select10-Rg2**, a range query on the key requiring 2 comparisons per tuple. Range queries are executed by scanning the 10,000 tuples in the operand relation. The additional comparison required .5 additional seconds, or .0005 additional seconds per tuple. While the cost of scanning the relation and doing two comparisons on the key is not prohibitive on a 3081, on a 4341, the cost of an additional comparison would work out to be about 3.5 seconds. Thus with a slower CPU, it would be worthwhile have the optimizer take more time and space, in the amount of code in the optimizer, to decide whether or not to use the index in performing a range query. If the size of the result is expected to be small, the index should be used.

Furthermore, it is clear that on all range queries using the condition box, the **Pipeline-Scan** as opposed to the **Simple-Scan** is being used. While this accounts for an increase of 1.83 seconds in query execution time on a 3081, it could cost 12.81 seconds on a 4341. Again, with a slower CPU, it would be desirable to have the optimizer take more time and space to figure out when the **Simple-Scan** can be used. It also brings into question the strategy of treating single relation queries as special cases of joins.

As with equality selections, we tested whether range selection on the key was any different from selection on a nonkey attribute. There was no remarkable difference. Query Select10-Rg2 on key is in Figure 5.c, Section 3.3.3.2. Query Select10-Rg2 on candidate key is in Appendix 3. Finally, we forced duplicate elimination by repeating the same selection and replacing the key with another candidate key, in the result. In this case, (there were no

Table 9: Times for Range Selections with Condition Box

Average CPU Time in Seconds on Loaded System
"P." on Query Table, 10 tuples in Result, unique1 is a Candidate Key

QUERY	CPUtime	COMPARISONS	DUP. ELIM.
Select10-Rg	2.42	Key<10	No
Select10-Rg2	2.92	5838<Key<5849	No
Select10-Rg2	2.87	5838<unique1< 5849	No
Select10-Rg2	2.89	5838 < unique1 < 5849	Yes

FIGURE 6. Query Output Mode

```
|------------------------------------------------------------------------------|
| Select100-Rg                                                                 |
|------------------------------------------------------------------------------|
|                                                                              |
| |----------|----------|----------|------|----------|----------|              |
| | TENK     | UNIQUE1  | UNIQUE2  | TEN  | THOUSAND | STRING4  |              |
| |----------+----------+----------+------+----------+----------|              |
|   P.       |          | _X       |      |          |          |              |
|            |          |          |      |          |          |              |
|            |----------|----------|------|----------|----------|              |
|                                                                              |
|                                                                              |
|                     ----------------------|                                  |
|                     | CONDITIONS          |                                  |
|                     |---------------------|                                  |
|                     | _X < 100            |                                  |
|                     |                     |                                  |
|                     |---------------------|                                  |
|                                                                              |
|------------------------------------------------------------------------------|
```

FIGURE 6.a Result Retrieved Through a "P." on a Query Table

```
|------------------------------------------------------------------------------|
| Select100-Rg                                                                 |
|------------------------------------------------------------------------------|
| |----------|----------|----------|------|----------|----------|              |
| | TENK     | UNIQUE1  | UNIQUE2  | TEN  | THOUSAND | STRING4  |              |
| |----------+----------+----------+------+----------+----------|              |
|            | _A       | _B       | _C   | _D       | _E       |              |
|            |          |          |      |          |          |              |
|            |----------|----------|------|----------|----------|              |
|                                                                              |
|   |----------|--------|--------|--------|--------|--------|                  |
|   | &ABC     | A      | B      | C      | D      | E      |                  |
|   |----------+--------+--------+--------+--------+--------|                  |
|     P.       | _A     | _B     | _C     | _D     | _E     |                  |
|              |        |        |        |        |        |                  |
|              |--------|--------|--------|--------|--------|                  |
|                                |---------------------|                       |
|                                | CONDITIONS          |                       |
|                                |---------------------|                       |
|                                | _B < 100            |                       |
|                                |                     |                       |
|                                |---------------------|                       |
|                                                                              |
|------------------------------------------------------------------------------|
```

FIGURE 6.b Result Retrieved Into User-Created Output Table

```
|----------------------------------------------------------------------------------------
| Select100-Rg
|----------------------------------------------------------------------------------------
||----------|----------|----------|------|----------|----------|
|| TENK     | UNIQUE1  | UNIQUE2  | TEN  | THOUSAND | STRING4  |
||----------+----------+----------+------+----------+----------|
|           | _A       | _B       | _C   | _D       | _E       |
|           |          |          |      |          |          |
|           |----------|----------|------|----------|----------|
|
|  |--------|------------|------------|------------|------------|----------------------------|
|  | PREDEF | A          | B          | C          | D          | E                          |
|  |--------+------------+------------+------------+------------+----------------------------|
|    I.     | _A         | _B         | _C         | _D         | _E                         |
|           |            |            |            |            |                            |
|           |------------|------------|------------|------------|----------------------------|
|
|                               |----------------------|
|                               | CONDITIONS           |
|                               |----------------------|
|                               | _B < 100             |
|                               |                      |
|                               |----------------------|
|----------------------------------------------------------------------------------------
```

duplicates in the 10 tuple result, but the query processor could not predict it), the cost of duplicate elimination was negligible.

4.2.3. Query output mode

In Table 10, we consider two range queries, **Select10-Rg** and **Select100-Rg**, that retrieve 10 and 100 tuples, respectively. With each query, we tried all three methods of outputting the result: "P." on query table; into user-created output; into a predeclared, empty relation. Figure 6 shows query **Select100-Rg**, phrased with each output mode.

When retrieving into an existing relation, an intermediate **TD Area** is used to initially form the tuples to be inserted. In our test, the result relation was initially empty. The major difference between retrieving into user-created output and retrieving into a predeclared, but empty, relation is the cost of actually creating the inversion when the result tuples are inserted into the predeclared relation.

It is interesting to note that whether the size of the result was 10 or 100 tuples, the difference between a "P." and retrieving into user-created output was greater than the difference between retrieving into user-created output and retrieving into a relation. Switching from storing the result as a list of tuple addresses (in a **pointer area** for a "P.") to creating a relation without inversion (in a **TD area** for user-created output) cost approximately .5 seconds whether the size of the result was 10 or 100 tuples. Compared to the time for retrieving into user-created output, retrieving into a predeclared, but empty relation required .1 second more when the size of the result was 10 tuples, and .3 seconds more when the size of the result was 100 tuples.

While selecting 100 tuples always cost more than retrieving 10 tuples, the difference is most noticeable when retrieving into a relation. When retrieving into a relation, it cost almost .3 seconds more to select 100 tuples than it did to select 10. Here, we note that for appropriate interactive queries, the cost of retrieving into a relation, even with the cost of creating complete inversion, was not particularly high. However, it was very inconvenient to have to predeclare a result relation if we wanted to be able to treat the result as a relation, (i.e. execute a query on a result relation).

4.2.4. Summary

In benchmarking selection queries, we found that the OBE prototype was very sensitive to a number of parameters. A summary of the factors that affected the performance of selections at low selectivities follows:

(1) The use of the condition box increased the cost of a range query by a factor of 5 due to the unnecessary use of the **Pipeline-Scan**.

(2) Equality selections were 4 times faster than range selections at low, comparable selectivities.

Table 10: Effect of Output Mode

"P." on Query Table, User Created Output, Into Predeclared Relation
Average CPU Time in Seconds on Loaded System
Range Selection on Key, With Condition Box

QUERY	Result Size	P.	UCO	Relation
Select10-Rg	10 tuples	2.42	2.98	3.08
Select100-Rg	100 tuples	2.55	3.04	3.35

(3) Range selections with one a simple qualifier, (1 inequality), were 15 to 20 percent faster than those with a composite qualifier, (2 inequalities).

(4) Retrieving tuples by a "P." on the query table was 15 to 25 percent faster than retrieving into user-created output which, in turn, was 3 to 10 percent faster than retrieving into a relation.

(5) Selectivity had a definite impact when retrieving into user-created output or into a predeclared relation.

Certain factors had a negligible impact on the performance of selections:

(1) Selecting on the key attribute, was no different from selecting on a candidate key.

(2) When there were no duplicates in the result, and the result was output through a "P." on the query table, the cost of duplicate elimination was negligible .

(3) The size of the result relation only slightly increased the cost of the query when the result was output through a "P." on the query table.

4.3. Projections

When projecting a relation on non-key attributes, duplicate tuples must be eliminated. Duplicate elimination is a costly operation that is usually performed by sorting or hashing. In OBE, although hashing is always used for a first phase in duplicate elimination, the actual method used for eliminating duplicates and storing result tuples depends on the output mode chosen for the query. Thus we tested the effect of query output mode on the performance of projections. With one query output mode, "P." on a query table, we found performance problems that seemed to be related to the number of duplicates in the result. Thus, we performed a special experiment to test the effect of the number of duplicate with this query output mode. All projections used in Section 4.3 are shown in Appendix 3.

4.3.1. Query output mode

Our baseline projection query, **Project20**, is a projection of a 10,000 tuple relation on 4 attributes: 3 integers and one 52 character string. The query produces 20 distinct tuples: thus 99.8% projected tuples are duplicates. In Table 11, timings of this query with the three possible output modes are presented: "P." on the columns of the projected attributes, user-created output (UCO), and into a relation. There are two differences between these queries. Retrieving into a result relation requires that the result, once formed in an intermediate segment, be inserted into the target relation, with the indices being built. When the relation being inserted into is empty to start with, this does not appear to be expensive. On the other hand, when retrieving into user-created output, there is the cost of formatting 8 result tuples for display to the screen. Apparently, these two costs offset each other. We observe that retrieving the tuples through a "P." command took 3.84 seconds (37%) more

Table 11: Effect of Method of Output
Projections on 10,000 Tuple Relation, 20 result tuples

TD = "P." on Query Table, UCO = User-created Output, Relation = Into Relation
Average CPU Time in Seconds on Loaded System

Query	Output	CPUtime
Project20-P	P.	10.29
Project20-U	UCO	6.46
Project20-R	Relation	6.46

time than retrieving into user-created output, but there was no difference between retrieving into a relation or into user-created output. This result is in striking contrast to our results with selections in Section 4.2, where for comparable queries, retrieving through a "P." command on a query table was 21% faster than retrieving into an empty relation and retrieving into user-created output was 30% faster. The explanation for this apparent contradiction is that the cost of duplicate elimination, incurred in the projection but not in a selection on a key attribute, outweighs the benefit of using a faster, space-conserving method of storing the result tuples.

Since OBE employs hashing to eliminate duplicate tuples, the effect of method of output when the number of duplicates is high, as was the case for the query in Table 11, can be explained by the way collisions are handled. When the result is in user-created output, hashing collisions are resolved by comparing tuples that are in the same **TD area** as the hash table. When the result is output through a "P." on a query table, pointers to result tuples are stored in a **pointer area** which also contains the hash table used for duplicate elimination. Hash table collisions must be resolved by comparing tuples in the operand relation area. Through the mechanisms provided by the high level language in which OBE was written, the cost of referencing the operand relation turns out to be quite high. It is possible that the code to resolve collisions when switching between the **pointer area** and the relation segment could be optimized.

4.3.2. Number of duplicates

The increased cost of a projection when the output mode is a "P." on the query table appears to be a function of the number of duplicates in the result. To quantify the cost of resolving collisions in the hash table in a **pointer area** we tested two more projections, with the "P." output mode, and fewer duplicates to be eliminated. The results in Table 12 demonstrate that as the number of duplicates is decreased, from 9980 to 2007, the time to do the projection decreased, from 10.29 seconds to 7.93 seconds, a factor of 23%. Surprisingly, the query that took the least time retrieved 7,973 more tuples. Thus, the **pointer area** is both space and time efficient when the result is large and there are few duplicates, but performance is impaired, regardless of the size of the result, when there are many duplicates.

4.3.3. Summary

Overall, projections in OBE are fast. By way of comparison with DDBS's, we will consider the cost of scanning our test relation of 10,000 tuples on a DDBS. To execute a projection, exclusive of the cost of duplicate elimination, the relation must be scanned. As stored in OBE, it requires 3.5 megabytes of storage. Assuming 30 milliseconds to read 1 4K block, scanning the entire relation would require 27 seconds, almost 3 times longer than OBE

Table 12: Effect of Number of Duplicates
Projections on 10,000 Tuple Relation

Output is by a "P." on Query Table Output
Average CPU Time in Seconds on Loaded System

Query	Result	# of Duplicates	CPUtime
Project20-P	20 tuples	9980	10.29
Project100	100 tuples	9900	9.95
Project8K	7993 tuples	2007	7.93

takes regardless of the method of output or the number of duplicates eliminated.[6]

We tested the combined effect of two factors on projections: the method of outputting the result and the effect of the number of duplicates eliminated. We found that the cost of retrieving the result through a "P." on the query table required up to 60 percent more time than retrieving either into a relation or into user-created output. Finally, we observed that the cost of a projection when the output mode was through a "P." could vary by 20% depending on the number of duplicates, and that the effect of a large number of duplicates far outweighed the effect of a large result with this output mode.

4.4. Deletions, Insertions, and Updates

OBE does not optimize its data structures or algorithms for updates. Clearly, maintaining complete inversion of every relation increases the cost of tuple insertion, deletion and modification. In a DDBS, this overhead would not be acceptable. However, because of the memory residency, OBE can exploit features such as a **long move** machine instruction in order to substantially reduce the cost of restructuring the indices (each index being a linear. array). Unlike in airline reservation or banking system transactions, in OBE, the designers assumed that other than an occasional update, updates are executed in bulk when the relations involved are not being used by multiple interactive users. Every deletion, insertion or update would be treated as a bulk operation, that is, one in which many more than one tuple may be involved. Thus, maintaining consistency of the database with each update was a major concern in the design of the update algorithms. For instance, deletions are done in two phases. First the tuples to be deleted are located and counted, then the user is prompted to verify whether the deletion should be carried through. With insertions, the entire relation is searched to make sure that the new tuples will not duplicate existing tuples.

We modelled test queries for updates so that these special features would be accounted for. In our measurements, we isolated the cost of the different phases in processing an update. With deletions, we isolated the cost of locating the tuples to be deleted from the cost of updating indices. Comparing insertions with deletions, we estimated the cost of the Power's sort [Po80] that is used to prevent the insertion of duplicates into a relation. With updates, which are actually deletions followed by insertions, we measured the efficiency with which overhead common to insertions and deletions is combined.

4.4.1. Deletions

In the first phase of a deletion, the candidate tuples are located by whatever method the optimizer deems appropriate, as if the query was a retrieval query. In our test queries, the tuples to be deleted are identified by an equality selection on one attribute, thus an **Index-Lookup** is used. The actual deletion is performed in the second phase which begins after the user has been informed of how many tuples were selected for deletion, and has validated the operation by pressing a function key. Thus we were able to isolate the cost of the consistency checking (i.e. locating and counting the target tuples) by aborting the test queries before the actual deletion. Times for deletion queries are presented in Table 13. We varied the number of tuples deleted between 1 and 1000. We accomplished this by picking one value of an attribute with a particular uniform distribution and deleting all tuples with that value. For instance, to delete 10 tuples in one query on relation tenktupe, we deleted all tuples where the attribute **thousand** was equal to 500. This attribute has a uniform distribution of the integers between and including 0 to 999. To delete 1000 tuples, we deleted all tuples where the attribute **twenty**=0. Queries **Delete1** and **Delete10** are

---

[6] However, if we consider OBE running on a 4341, we would find that the fastest projection, **Project20-U**, which required 6.46 seconds on a 3081, would require about 45 seconds on a 4341. Thus, for this implementation to perform well on a 4341, smaller relations would be required.

shown in Appendix 3.

For each query in Table 13, we have decomposed the total CPUtime into the time for locating the tuples to be deleted, labeled "Checking", and the time for the actual deletion. The latter approximately represents the time for updating the indices. The cost of checking gives us a very good estimation of the cost of doing equality lookups on an index. For deleting 1 and 10 tuples, it is almost certainly dominated by the time parse the query. To locate 1000 tuples requires only .19 seconds more than locating 1 or 10, or .00019 seconds per tuple. On a ten MIP machine, this is approximately 190 machine instructions per tuple located.

Since **our** test relation has 16 attributes, 16 indices are updated in the second phase of the deletions (last column in Table 13). The "Restructuring Cost" entry for the query **Delete**1, .14 seconds, includes the time for 16 **long move** instructions that must be executed to recompact the indices. We observe that the cost of updating the index is linear in the number of tuples deleted:

Cost to Update Index = 0.14 * (**number of tuples deleted**)

Because the cost of locating the tuples to be deleted is so low relative to the cost of updating the indices, the total time to delete tuples is also close to linear in the number of tuples deleted. When 1000 tuples are deleted, the cost goes slightly under

0.14 * (**number of tuples deleted**).

In this case, with each deletion from the relation, the indices get smaller and searching them for the appropriate tuple to delete takes less time.

Although the time for deletions includes restructuring of the indices, and removing the tuple addresses from the doubly linked list of TD's, it does not account for restructuring the relation completely. Unlike the space for the indices, the space for TD's and data is not reclaimed. It can be reclaimed by running a compacting program when the relation is not being used.

4.4.2. Insertions

Insertions are executed by first forming the tuples to be inserted in an intermediate **TD area**. The segment is identical to one used for user-created output, including a hash table at the bottom of the segment that checks for duplicate tuples. Extensive checking is done to insure that the user does not make mistakes with keys, such as trying to insert two tuples with identical keys but different values for nonkey attributes. Enough information,

Table 13: Deletions on 10,000 Tuple Relation

Average CPU Time in Seconds on Loaded System
Tuples Selected for Deletion through Simple Equality Selection on Integer Attribute.

Query	Number Deleted	With Actual Deletion	Consistency Checking	Restructuring Cost*
Delete1	1 tuple	.23	.09	.14
Delete10	10 tuples	1.55	.09	1.46
Delete100	100 tuples	14.97	.11	14.86
Delete500	500 tuples	72.55	.18	72.37
Delete1000	1000 tuples	139.56	.28	138.98

* Restructuring Cost ≡ (Time With Actual Deletion) - (Time With Only Checking)

such as the inconsistent tuples, is saved in a different segment to provide meaningful error messages to the user. The number of tuples that will actually be inserted is determined and the user is asked, as in deletions, to validate the insertion before any changes are made in the relation. The tuples are inserted one at a time, in the first free space below the tuples already in the relation. As each tuple is inserted, the indices for each attribute are updated. As with deletions, if 10 tuples in a 16 attribute relation are inserted, at least 160 move long instructions must be executed.

In contrast to deletions, updating an index for insertions takes considerably less time but the consistency checking required for an insertion took considerably more time. To isolate these costs, we designed two insertions, **Insertkey** and **Insertall**, that both insert one tuple. However, **Insertkey** inserts a tuple where only the key inserts a new value in its attribute domain, while **Insertall** inserts a tuple that contains a new value in all attribute domains. Thus for the first query, only one new value was actually inserted in the database. To store the other 15 attributes, an equality lookup was performed on the index for each attribute, and a pointer to the attribute value already stored was copied into the TD for the tuple being inserted. For the second query, 16 new attribute values had to be stored in the database. Queries **Insertkey** and **Insertall** are shown in Appendix 3. Times for these two queries are presented in Table 14. As with deletions, we have isolated the time for consistency checking from the time for restructuring the relation. The difference between the costs of the two queries was small (0.02 sec), indicating that storing attribute values does not account for much of the insertion time. On the other hand, for both queries, the contrast between the time to check an insert and the time to restructure the relation is quite remarkable. The time to parse the query and check the insertion was 4.16 seconds in both queries. Of this time, at least 0.51 seconds was parse time, (Section 3.3.3.1). This leaves 3.65 seconds for consistency checking when only 1 tuple is inserted. Although apparently high, this time is actually very fast considering that it includes the time to sort a 3.5 megabyte relation on its key, the integer attribute **unique2** using Power's sort [Po80].

While for deletions, we could assume that the time to actually delete a tuple was insignificant in comparison to the time to delete the pointer to the tuple from all the indices, with insertions, this is clearly not the case. When all the attributes being stored are unique, it appears that is took more time to actually store the tuple than it did to update the indices. This accounts for the restructuring cost to store 16 unique attributes being twice the restructuring cost of storing only 1 unique attribute and 15 nonunique attributes, 0.04 seconds as opposed to 0.02 seconds. If we take 0.02 seconds as an upper bound for the time to update 16 indices, this means each index update required about 1.25 milliseconds. This includes the time for performing the binary search of the index, the long move instruction and the insertion of the new tuple address in the space created by the long move instruction.

Table 14: One-tuple Insertions in 10,000 Tuple Relation

Average CPU Time in Seconds on Loaded System
Number Unique Attribute Values ≡ Number of Attributes Values Inserted

Query	Unique Attribute Values	With Actual Deletion	Consistency Checking	Restructuring Cost*
Insertkey	1	4.18	4.16	.02
Insertall	16	4.20	4.16	.04

* Restructuring Cost ≡ (Time with Insertion) - (Consistency Checking)

4.4.3. Updates

Updates consist of a deletion followed by an insertion. Before the update is actually performed, the tuples to be updated are copied into an intermediate segment. If the key is updated, checking is performed to insure that no duplicates or inconsistencies in the key will be caused by the updates and to detect various subtle anomalies that can occur when more than one tuple is updated at the same time. In the process of this checking, both the intermediate and target relation are sorted using Power's sort [Po80]. If the key is not updated, no checking is done. As with deletions and insertions, after all checking has been done and the number "X" of tuples that will be updated has been determined, the user is asked to "Press <Process> to Update X Tuples". If the user presses the function key, then X tuples are deleted, with all indices being updated after each deletion, and then the X new tuples are inserted, with all indices being updated after each updated tuple is inserted. Updates are summarized in Table 15.

The test queries tried to update 1, 5, 10, 100, 500 and 1000 tuples. However, for 100, 500 and 1000 tuples, we could only complete the consistency checking phase. These updates failed because we ran out of memory. This highlights a potential problem for MMDBS designers who rely on a host operating system for memory management. OBE required that all of the code and program data space, all operand relations, all intermediate segments, and all result segments fit in the virtual memory allocated to a user's virtual machine. The OBE benchmarking program, code and data space took about 4 megabytes of virtual memory. The **TenK** relation required 3.5 megabytes of memory. This put a lower limit on the amount of virtual memory required to execute a query at 8 megabytes. The maximum amount of virtual memory allowed a user was 16 megabytes. When we tried to update 100 tuples, the source relation segment was not large enough to fit 100 new tuples. Therefore, OBE, using a call to CMS, requested a new segment of virtual memory of over 3.5 megabytes so that it could copy the relation into a larger segment and then insert the 100 updated tuples. Although there was enough unallocated virtual memory remaining to fill this request, the largest unallocated block of virtual memory was too small to fill the request due to fragmentation of virtual memory, so the update failed.

The first three queries in Table 15 all update one tuple. Only the first two queries, **Update1-key** and **Update1-key+**, update a key attribute. **Update1-key+** actually performs the same update as **Update1-key** but contain 15 superfluous attributes in the query

Table 15: Updates on 10,000 Tuple Relation

Average CPU Time in Seconds on Loaded System
Tuples to be Updated Selected through Simple Equality Selection

Query	Attribute Updated	Number of Tuples Updated	Complete Update	Consistency Checking	Restructuring Cost*
Update1-key	Key	1	3.68	3.51	.17
Update1-key+	Key	1	4.19	4.03	.16
Update1	Nonkey	1	.28	.11	.17
Update5	Nonkey	5	.94	-	-
Update10	Nonkey	10	1.79	.12	1.67
Update10	Nonkey	10	1.79	.12	1.67
Update100	Nonkey	100	-	.25	-
Update500	Nonkey	500	-	1.02	-
Update1000	Nonkey	1000	-	2.48	-

*Restructuring Cost ≡ (Time With Actual Update) - (Consistency Checking) window. Queries **Update**1**-key** and **Update**1**-key+** are shown in Figure 4, Section 3.3.3.1. The difference in time between these to queries, 0.51 seconds, is accounted for by the additional parse time required for query **Update**1**-key+**. Updating the key incurs a very high cost for consistency checking, a minimum of 3.51 seconds for the Power's sort [Po80] for the target relation (Section 4.4.2), while restructuring the relation is very fast, .16 to .17 sec. When updating a key attribute, as with insertions, the target relation is sorted on its key to facilitate duplicate elimination.

Update1 and all of the remaining queries shown in Table 15 update one non-key attribute. **Update**1 requires only .28 seconds total, which is 92 percent faster than the same update on a key attribute. We also note that this is only while an update does consist of both an insertion and a deletion, a great deal of unnecessary overhead is eliminated where possible. In the case of updating a nonkey attribute, the overhead that is eliminated includes the Power's sort of the target relation.

This is even more clearly seen in the checking cost for **Update1000**, which is an update of 1000 nonkey attributes. Because the target relation was not sorted, the time to check for duplicates is limited to the tuples in the intermediate segment that the tuples to be updated have been copied to. Thus, the cost of checking for updating a nonkey attribute in 1000 tuples is even less than the cost of checking when a key attribute in 1 tuple is updated. We should also point out that if the update of the key was a bulk update, the target relation would still only be sorted once and the cost of the sort would be amortized over the number of tuples updated.

4.4.4. Summary

In spite of having completely inverted relations and doing a great deal of consistency checking for the user, OBE handles insertions, deletions and updates well. Updating nonkey attributes is fast: .28 seconds for one attribute in one tuple. Updating the key attribute, and inserting are much slower due to extensive consistency checking. Bulk deletions performed well, with a cost increasing linearly with the number of tuples deleted. But we were unable to thoroughly test bulk updates, because in preparing for the insertion of updated tuples, OBE required more contiguous virtual memory than could be allocated. This problem is partially due to our benchmarking technique. We used a relation much larger than this prototype was designed to handle.

The **long** move instruction is the feature that makes complete inversion at all feasible in a MMDBS. By way of contrast, for a DDBS, if there were 16 indices on a relation, the disk resident indices could not be updated in under 16 disk accesses. Assuming 30 milliseconds to read one index page from disk, updating the indices alone would require .48 seconds. In fact, OBE updates 16 indices in under 20 milliseconds, less time than the cost of reading 1 index page from disk.

4.5. Aggregates

The strategy for perfoming simple aggregates and aggregate functions is similar in OBE. Two different execution strategies are employed, depending on whether the result is retrieved: 1) through a "P.", 2) into user-created output or into a relation. We tested the effect of query output mode on the performance of aggregates.

4.5.1. Query output mode

If the result output mode is through a "P." on the query table, all selections and joins are performed and pointers to tuples in the result are written into an intermediate **pointer area**. Then, if necessary for an aggregate or aggregate function, the pointers in the intermediate segment are sorted using a heap sort [Kn73]. Finally, the aggregate or aggregate function is computed on the tuples in the intermediate segment and the result is written to a **TD area**.

If the result output mode is into user-created output or into a relation, all selections and joins are performed and the tuples in the result are written into an intermediate **TD area**. Then, if necessary for an aggregate or aggregate function, the TD's intermediate **TD area** are sorted using Power's sort [Po80]. Finally, the aggregate or aggregate function is computed on the tuples in the intermediate segment and the result is written to a result **TD area**. If the result is retrieved into a relation, the tuples in the result **TD area** are inserted into the relation.

We tested several aggregates and aggregate functions and we found that two were representative of the performance of these queries. Query **MinKey** was a simple aggregate that asked for the minimum value of a key attribute. The result of **MinKey** was one tuple with one attribute. Query **MinFn** was an aggregate function that asked for the minimum key value grouped by the attribute **tene**. Here the result was 10 tuples, with two attributes in each tuple. Both queries, and their results, are shown in Appendix 3.

Initially, we tested aggregates with a "P." on the query table as the method of outputting the result. After analyzing projections, we realized that storing the intermediate result in a **pointer area** might be profoundly degrading the performance of aggregates. Then, we ran both queries retrieving the result into user-created output. The times for these four queries are in Table 16.

When the result output mode was through a "P." on the query table, the simple minimum aggregate, query **MinKey**, required 15.64 seconds. The aggregate function, query **MinFn-P**, required 36.98 seconds. When we retrieved the result into user-created output, the time for both queries was reduced over 70 percent. This difference is clearly a result of the Power's sort [Po85] being used in queries **MinFn-U** and **MinKey-U**. In the Power's sort on a **TD area**, all comparisons are made within the intermediate **TD area**. However, for queries **MinFn-P** and **MinKey**, when the heap sort [Kn73] is performed on a **pointer** area, the relation corresponding to the **pointer area** must be referenced to actually perform comparisons between attributes.

4.5.2. Summary

The poor performance of aggregate functions when the query output mode is through a "P." reinforces our opinion that the **pointer area** is a source of performance problems. However, the **pointer area** handles large results very well when comparisons do not have to be made between tuples in the result, and it is also very space efficient. Therefore, it would be worthwhile to attempt optimization of the code for performing comparisons between tuples referenced in the **pointer area**.

In contrast, when the **pointer area** is not used in an aggregate, OBE performs well. The minimum aggregate function when the query output mode was into user-created output

Table 16: Times for Aggregates and Aggregate Functions

Average CPU Time in Seconds on Loaded System

QUERY	CPUtime	RESULT	COMMENT
MinKey-P	15.64	P.	Minimum on Key
MinKey-U	4.53	UCO	Minimum on Key
MinFn-P	36.98	P.	Min. Function on Key, 10 Partitions
MinFn-U	8.01	UCO	Min. Function on Key, 10 Partitions

required 8.01 seconds. Previously, we computed that just scanning relation **TenK** in a DDBS would require approximately 27 seconds if **TenK** required 3.5 megabytes of storage. Using fixed sizes for attributes, **TenK** can be stored compactly in 1.8 megabytes. Scanning 1.8 megabytes from disk at 30 milliseconds to read one 4K page would require 13 seconds. OBE can perform the aggregate function in 8.01 seconds.

4.6. Joins

A pivotal assumption in the implementation of the OBE prototype, was that its DBMS functions would frequently be used for ad hoc complex queries involving multiple relations. Our test join queries were designed to measure the effect of the following factors:

(1) Query Complexity. We varied this factor by considering 3 test queries: a simple 2-way join, a join combined with a selection, and 2 joins combined with 2 selections.

(2) Number of Operand Relations. We varied this number from 2 to 4 (thus considering 2 to 4-way joins).

(3) Query Output Mode, combined with the above.

All join queries referred to in this section are shown in Appendix 3.

4.6.1. Join algorithm

Since the relational join is a basis for most complex queries, special attention was paid to optimizing join queries. OBE uses a pipelined strategy [AHK85] that avoids the creation of temporary, intermediate relations in the process of query execution. This element is crucial with regards to the main memory residence assumption, since temporary relations incur a large storage overhead. For instance, a sort-merge join would not be a viable option, mainly because it requires temporary storage as large as the operand relations to store the sorted relations before they are merged and joined.

Essentially, the pipelined strategy is a nested-loops join, processed in main memory and optimized by a depth-first search. When joining a number of relations, the query optimizer determines an optimal ordering to nest the loops. One optimization is to choose the relations predicted to have the smallest number of qualifying tuples as the outer relations. However, the ordering is part of a global query optimization process [Wh85]. For each tuple of the outer relation, the second relation is scanned for matching tuples. For each matching tuple of the second relation, the third relation is scanned, etc. Within each loop, all relevant selection qualifications are checked before proceeding with the join. The pipelined join algorithm is summarized in Figure 7.

4.6.2. Query complexity

For this experiment, we used two 10,000 tuple test relations, **A** and **B**, and a temporary relation **B'** containing 10 tuples selected from **B**. Relation **A** is actually relation **TenKe** and relation **B** is actually relation **TenKf**. Three test queries were designed to measure the efficiency of the query optimizer, when a number of relational operations are combined. In order to isolate the effect of query complexity from other confounding factors, in the three queries, we used the same building block: a 2-way join between a 10,000 tuple relation and a 10 tuple relation, on an integer attribute. We also retrieved the result into user-created output, from the same columns. Unlike in our other test queries, retrieving by using the "P." command did not seem appropriate. does not juxtapose tuples that are joined. The three queries were:

(1) **JoinAselB**: a selection on **B** producing a 10 tuple relation **B'**, followed by a join with **A**

(2) **JoinAB'**: join of a 10,000 tuple relation **A** with the 10 tuple relation **B'**

Figure 7: A Summary of the Pipelined Join Algorithm [AHK85]

```
{ Join R1, R2, ..., Rk. Join-Selection Predicate is Qual }
    for each tuple t1 in R1
        if (t1,**...) satisfies Qual then
            for each tuple t2 in R2
                if (t1,t2,**...) satisfies Qual then
                .
                .
                .
                    if (t1,...,tk) satisfies Qual then
                        form join tuple
```

(3) **JoinCselAselB**: query **JoinAselB**, followed by a selection, followed by a join with **C**. Relation **C** is identical to relation **B'**. This query is the same as a join between **B'**, which contains 10 tuples, and relations **A** and **B**, combined with selections on **A** and **B** that produce 10 tuples.

We used queries very similar to these in previous benchmarks [BDT83]. See Appendix 2. Timings for these three queries are presented in Table 17.

Our basic join was query **JoinAselB**, shown in Figure 3, Section 3.3.2. It was a join of two 10,000 tuple relations tuple relations with a selection on one of the relations. The selection was used to limit the size of the result to 10 tuples. The result consisted of 5 attributes, (4 integers and 1 string), exactly the same result attributes used in testing selections. Whether we joined two or three relations, we always retrieved the result from 2 relations so that our results would not be confounded by an additional variable - the number of relations that the result was assembled from.

Our first obervation is that the joins **JoinAB'** and **JoinCselAselB** are very fast compared to selection queries from Section 4.2. On the other hand, the join **JoinAselB**, which was less complex than **JoinCselAselB**, took 2.82 seconds. **Select10** has the same selection predicate on Tenktupe that is used in **JoinAselB** and required 2.98 seconds when the result output mode was into user-created output. (See Table 10 in Section 4.2.3). A simple selection took 5 percent longer than the equivalent selection performed in conjunction with a join! If performed optimally, however, we think **JoinAselB** could have taken even less

Table 17: Join Complexity for Interactive Joins

Average CPU Time in Seconds on Loaded System
Result Tuple: 4 integers and 1 string, Assembled from 2 Relations
Equijoins on key, UCO output mode, with condition box, 10 tuples in result

Query	CPUtime	Operand Relations	Size of Each Relation
JoinAselB	2.82	2	10K,10K tuples
JoinAB'	0.46	2	10K,10 tuples
JoinCselAselB	0.67	3	10,10K,10K tuples

time. Using the **Simple-Scan** to perform the selection on relation B should take no more than .59 seconds, the time required for query **Select10-Rg**, (Table 8, Section 4.1.3). After the selection is performed, the join should take no longer than query **JoinAB'**. **JoinAB'** is the join of relation A with the relation B'. The relation B' contains the result the same selection used in **JoinAselB**, 10 tuples, sixteen attributes per tuple. **JoinAB'** required .46 seconds CPUtime. Adding the cost of query **Select10-Rg** and the cost of query **JoinAB'** we get: .59 + .46 = 1.45 seconds. **JoinAselB** required 2.82 seconds however, implying that this query was not executed optimally.

JoinCselAselB is a join between relation **B'**, which contains 10 tuples, and relations **A** and **B**, both of which contain 10,000 tuples. The selections on relations **A** and **B** are identical to the selection used to create **B'**: unique2 < 10. **JoinCselAselB** required .67 seconds, only .21 seconds more than query **JoinAB'**.

OBE correctly optimized **JoinAB'** and **JoinCselAselB**. That is, it put relations that would limit the size of the result relation at the head of the pipeline and avoided scans of relations through proper use of indices. However, with both queries **JoinAB'** and **JoinCselAselB**, the choice for the optimizer was not very difficult as there was one 10 tuple relation in both joins.

4.6.3. Number and size of operand relations

Our second experiment measured the impact of increasing the number of relations in a join. We joined 2, 3 and 4 relations, varying the size of the operand relations so that the total number of tuples in all of the relations was between 9K and 13K. We did not want to limit the selectivity of the join through a selection, so we always included the relation **OneK**, which had a thousand tuples, and executed an equijoin on the key of each relation. This resulted in the size of the result always being 1000 tuples. In addition, we were not testing the optimizer, so we chose queries that appeared to be correctly optimized. The optimizer should have put the smallest relations at the head of the pipeline, and judging by the performance of these queries, did so.

We were also not particularly interested in measuring the effect of having a very large result, so we offset the cost of a large result relation by retrieving only tuples from relation **OneK** through a "P.". This resulted in the result relation being stored in a **pointer area** which we found to be insensitive to the size of the result with selections and projections. Furthermore, retrieving join results through a "P." on one relation, allows the pipeline evaluation to be short-circuited. Once it is determined that a tuple in **OneK** will successfully join with at least one tuple from every other relation, the pointer to that tuple can be written to the result **pointer area** and the next tuple in the scan of **OneK** can be considered. Finally, for these joins, we did not use the condition box. Times for these joins are in Table 18.

The times for executing a join was clearly more sensitive to the number of relations in the join that to the total size of the relations involved in the join. This can be seen in Table 18 where the query execution time is strictly increasing with the number of relations involved in the join, in spite of the fact that the 4 way join, which required the most time, 2.18 seconds, had the fewest total number of tuples in its operand relations, 10,000 tuples.

By way of comparison with DDBS's, we will consider the cost of one common component of these three joins on a DDBS. To form the result relation on a DDBS, the relation **OneK** from which all 1000 tuples are placed in the result, must be scanned. We consider two cases:

(1) Assume **OneK** is very compact. That is, each of its 13 integer attributes require 2 bytes of storage and each of its string attributes require 52 bytes of storage. Then, one tuple of **OneK** would be 182 bytes wide and the entire relation could be stored in 182,000 bytes. To read OneK from disk, assuming 30 milliseconds to read one 4K

Table 18: Effect of Number of Operand Relations on Joins

Average CPU Time in Seconds on Loaded System
1000, 16-attribute Tuples in Result
Result Retrieved by "P." on Query Table for Relation OneK
Equijoins on Key Attributes, No Condition Box, No Duplicate Elimination

Query	CPUtime	Operand Relations	Size of Each Relation	Operands Total Size
Join2	1.29	2	1K,10K tuples	11K tuples
Join3	1.73	3	1K,2K,10K tuples	13K tuples
Join4	2.18	4	1K,2K,2K,5K tuples	10K tuples

block, would require 1.35 seconds.

(2) As it is stored in OBE, OneK requires 360,000 bytes of storage. Assuming 30 milliseconds to read one 4K block, scanning **OneK** would require 2.64 seconds.

Thus, each of these joins requires under 1 second more than part of the cost of forming the result on a DDBS, (2.18 - 1.35 = 0.83).

The difference between the two way join, **Join2**, and the three way join, **Join3**, was .44 seconds. The difference between the **Join3** and the **Join4** was .45 seconds. Dividing the time for the join by the number of relations involved in the join we get .65 seconds per relation for the two way join, .58 seconds per relation for the three way join and .55 seconds per relation for the four way join. Thus, the increase in time as the number of relations joined was increased was less than linear. This effect also holds when queries **JoinAB'** and **JoinCselAselB** are added to the picture as is illustrated in Table 19.

Again, dividing the time for the join by the number of relations involved we get .23 seconds for **JoinAB'** and .22 seconds for **JoinCselAselB**. What all of these queries have in common, is that the size of the result is limited by the smallest relation involved in the join, and all of the joins are 1 to 1, (equijoins on key attributes). Query **JoinAselB**, has the size of the result limited by a range selection on relation **B**, which adds at least the cost of a comparable range selection on **B**, was not executed optimally.

Table 19: Summary of Joins

Average CPU Time in Seconds on Loaded System
Operand and Result Size in Tuples
Result Composition = # Attributes / # Relations Composed from

Join Query	Result Size	Result Composition	Output Mode	# Relations Joined	Total Operand Size	CPUtime
AB'	10	5/2	UCO	2	10K + 10	.46
CselAselB	10	5/2	UCO	3	20K + 10	.67
Join2	1000	16/1	P.	2	11K	1.29
Join3	1000	16/1	P.	3	13K	1.73
Join4	1000	16/1	P.	4	10K	2.18
AselB	10	5/2	UCO	2	20K	2.82

4.6.4. Summary

We joined two to four relations with either 10 or 1000 tuples in the result. The total number of tuples in all relations ranged from 10,010 to 20,010. The total size of all relations involved in a single query ranged from 4 megabytes to 8 megabytes. With our particular subset of 6 joins we found that:

(1) All 6 joins executed in under 3 seconds.

(2) For 1 to 1 joins with the size of the result held constant, the time for the joins increased linearly with the number of relations in the join.

(3) A two way join that included a range selection executed 5 percent faster than the comparable simple range selection.

5. SPACE/TIME MEASURES

Memory management is a complex aspect in OBE. A design assumption was that all code and data would have to fit in the virtual memory allocated to the user. Then, for the system to achieve good performance, the physical memory available should be almost as large as the virtual memory. In this section, we analyze the memory requirements of OBE and investigate the **Space/Time** tradeoffs in query optimization. In particular, we quantify the degradation caused by paging when the physical memory requirements are not met.

5.1. Memory Requirements

OBE requires a certain amount of virtual memory, for the code and data segments of the database management system. In addition, for each query, it requires another allocation of virtual memory for storing the operand relation segments.

5.1.1. Size of the code

In the design of a main memory database system, it is critical to keep the size of the code small in order to leave as much memory as possible for database relations. Furthermore, a system with a large amount of code is vulnerable to performance degradation due to code being paged out.

The OBE module occupies 509 CMS blocks, that is 2,084,864 bytes. When it is structured with a shared segment, it requires slightly more memory. In this case, the reentrant, shared portion requires two megabytes. The unshared portion of the code then requires 278,776 bytes of the user's virtual memory. In addition, CMS occupies 131,072 bytes of a user's virtual memory. Finally, the program stack and heap occupy over 1 megabyte of memory. Thus, with shared segment, the code, operating system, program stack and heap occupy under 2 megabytes of memory. Without shared segment, the code, operating system, program stack and heap occupy slightly under 4 megabytes.

We created a special benchmarking module in order to include measurement probes in the code and simulate user response where it was required to complete a query. This special module occupied 2,162,655 bytes. In addition, the benchmark module was built with a larger heap and certain data structures were enlarged to handle a few of our queries that did not fit in one screen. Including the program stack and heap and space for the operating system, the benchmarking version of the OBE module required approximately 4 megabytes of virtual memory.

5.1.2. Virtual memory requirements

We determined the virtual memory requirements for a cross section of the queries described in Section 4, by submitting the special benchmarking program with one query of interest plus the necessary initializing queries to the virtual batch machine, on the 3081.

When a program is submitted in batch, the user specifies the amount of virtual memory to be given to the batch machine. If the batch machine runs out of virtual memory, the OBE program aborted. By submitting only one query of interest per program, we were able to measure all queries in an similar, favorable environment with respect to the state of the virtual machine's memory. Each batch job is given its own virtual machine just as an interactive user is when the user logs on.

We tested a cross section of our interactive queries with 7 to 16 megabytes of virtual memory allocated to the virtual machine. We found that none of our queries of interest ran in under 8 megabytes of virtual memory although some of our initializing queries ran in 7 megabytes, or less. Since the size of virtual memory can only be specified in megabytes, if a query required 8 megabytes of virtual memory, it actually used more than 7 and up to 8 megabytes.

The minimum amount of virtual memory required for the queries tested is in Table 20. As can be seen, most of the interactive queries ran in 8 megabytes of virtual memory. This is not surprising since the benchmarking program required 4 megabytes of virtual memory and the relation **TenK**, which was an operand relation in every query, occupied 3.5 megabytes of memory. However, we must also point out that by reading in a relation when it is first referenced, we end up with a selection that selects one tuple on the key requiring 8 megabytes of virtual memory, even though the equality lookup used accessed very few pages of the relation. This results in a reasonable **space-time product** only if the weight of space, relative to time, is extremely low.

A problem for the OBE optimizer was estimating the size of the result segment. The result segment was always allocated before a query was executed. If OBE requested a result segment that was too large, and CMS did not have a contiguous block of memory large enough, OBE had to abort processing. On the other hand, if the result segment was too small, the query would have to be restarted with a larger result segment being allocated. Since the relation **TenK** could not even be read into memory in under 8 megabytes of virtual memory, the reasonable response times of the interactive queries that ran in 8 megabytes imply that the optimizer successfully balanced the tradeoffs involved when appropriate interactive queries were executed.

The aggregate function **MinFn**, provides an example of the space/time tradeoffs involved in the use of the **pointer area** used with the "P." result output mode. When the result was retrieved through a "P." on the query table, query **MinFn-P** executed in 36.98 seconds and required a minimum of 8 megabytes of virtual memory. When the result was retrieved into user-created output, **MinFn-U** executed in 8.01 seconds but required 9 megabytes of virtual memory.

The maximum amount of memory required by any query was 11 megabytes. Except for the aggregate function **MinFn-U**, all of the queries that required more than 8 megabytes were joins. However, half of the joins we tested did run in 8 megabytes of memory. Virtual memory requirements for joins are summarized in Table 21.

Looking at the difference between the joins that ran in 8 megabytes of memory and the join that required 9, we see that operand relations occupying 3.8 megabytes of virtual memory could be accommodated in 8 megabytes but operand relations occupying 4.5 megabytes could not. The breaking point for queries that can be executed in 8 megabytes of virtual memory is somewhere in between.

The initializing query that read in relation **TenKf** when relation **TenKe** was already in memory could not be executed in under 11 megabytes of virtual memory. Thus, it is logical that the two joins that involved both 10,000 tuple relations required at least 11 megabytes of virtual memory.

The fact that two copies of the 10,000 tuple relation easily fit into 11 megabytes of virtual memory is in contrast to the failure of bulk updates when 100 or more tuples were

Table 20: Virtual Memory Requirements to the Nearest Megabyte

Average CPU Time in Seconds on Loaded System
Default Query Parameters are as defined for Standard Interactive Query

Query	Memory	CPUtime	Comment
InitB'	7	.04	read in B'
Init1K	7	.04	read in OneK
Init2Kb	7	.04	read in TwoKb.OneK already in
Init2Kc	7	.04	read in TwoKc;1K,2Kb already in
InitParse	7	.07	dummy query to cause start I/O's
Init5K	8	.05	read in FiveK;1K,2Kb,2Kc in
Init10Ke	8	.05	read in TenKe
Select100-Rg+	8	.99	entire tuple retrieved
Select10-Rg	8	.59	range, no condition box
Select10-Rg	8	2.42	range, with condition box
Select10-Rg2	8	2.92	2 qualifications
Select10-Rg2	8	2.89	with duplicate elimination
Select10-Eq	8	.17	equality, thousand = 838
Select100-P	8	2.55	select 100 tuples
Select100-U	8	3.04	into UCO
Select100-R	8	3.35	into relation
Selectkey-Eq	8	.14	select 1 tuple on key
Project20-U	8	6.46	20/10,000 tuples, into UCO
Project20-R	8	6.46	20/10,000 tuples, into Relation
Project20-P	8	10.29	20/10,000 tuples
Delete1	8	.23	where key=838
Delete10	8	1.55	where thousand=500
Delete1000	8	139.56	where ten=0
Insertkey	8	4.19	1 new value inserted in relation
Insertall	8	4.20	16 new values inserted in relation
Update1-key+	8	4.19	all attributes in query window
Update1-key	8	3.68	only key in query window
Update1	8	.29	update candidate key
Update5	8	.95	update 5 tuples
Update10	8	1.79	update 10 tuples
MinKey	8	15.64	minimum on key
MinFn-P	8	36.98	minimum on key, ten partitions
JoinAB'	8	.46	into UCO
Join2	8	1.29	Join 10K and 1K
Join4	8	2.19	Join 1K,2K,2K,5K
Join3	9	1.73	Join 1K,2K,10K
MinFn-U	9	8.01	min on key,10 partitions,into UCO
Init10Kf	11	.05	read in TenKf, 10Ke already in
JoinCselAselB	11	.67	into UCO
JoinAselB	11	2.82	into UCO

Table 21: Virtual Memory Requirements for Joins

Minimum Memory to the Nearest Megabyte
Average CPU Time in Seconds on Loaded System
Total Size of Operand Relations in Bytes

Query	Min. Memory	CPUtime	# Relations	Relations Total Size
JoinAB'	8	.46	2	3,649,280
Join2	8	1.29	2	3,825,664
Join4	8	2.19	4	3,542,040
Join3	9	1.73	3	4,538,368
JoinCselAselB	11	.67	3	7,294,496
JoinAselB	11	2.82	2	6,930,432

updated. These updates failed because there was no contiguous block of virtual memory available to expand the **TenK** relation into. This is surprising since a bulk update of 10 tuples only required 8 megabytes of virtual memory. We tested the bulk update of 100 tuples in 16 megabytes of virtual memory. It would seem that an 8 megabyte increase should accommodate the difference between updating 10 and updating 100 tuples, but it did not. Our conclusion is that fragmentation of virtual memory is a serious problem for the designers of MMDBS's to consider.

5.2. Effect of paging

When its physical memory requirements are not met, the MMDBS relies on the operating system for virtual memory management. A ubiquitous criticism of the concept of MMDBS's is that no matter how well they perform when their physical memory requirements are met, paging will unacceptably degrade their performance. Thus it is important to measure the effect of paging on the performance of the MMDBS, and establish a ratio of physical to virtual memory size that will provide acceptable performance. In designing a paging experiment, our goal was to measure how badly paging affected the performance of the MMDBS, in absolute terms and as compared to buffered I/O in a conventional database system. We use a technique that can be used to test locality of reference in algorithms without tracing memory references. Depending on the resource management and scheduling algorithms of the underlying operating system, this paging experiment can also be viewed as a way to emulate the operation of the MMDBS in a multi-user environment, under controlled test conditions.

In order to force paging, a simple approach is to time queries executed by the MMDBS when the system is loaded. We rejected this approach because it is too difficult to determine how much of the degradation in elapsed time is caused by waiting in CPU and I/O queues versus how much degradation is caused by waiting for a page to be read or written. This task is made even more difficult if the system involved uses a cache for paging, as the cost of a page read or write depends on both the state of the cache and the state of the disk used for paging. The other approach, which is consistent with our general benchmarking methodology, is a controlled paging experiment that eliminates confounding variables.

We measured the effect of paging in a single user, standalone experiment on a 4341 that used a 3350 disk for paging. The 4341 did not have a paging cache. It had 16 megabytes of real memory available, consisting of four 4-megabyte memory modules. To force paging, we could bring the machine up in a configuration that used 1,2 3, or 4 of the 4-megabyte memory modules. Then, by systematically varying both the size of the query operand relations, from .82 to 9.32 megabytes, and the amount of physical memory made available to OBE, from 4 to 16 megabytes, we forced paging in a controlled manner.

A number of parameters may determine the amount of paging activity: the number and the size of the operand relations, the size of the result relation and the query type. As our access to the 4341 was limited, we did not have the time to perform experiments fully investigating all of these factors. Therefore, we restricted ourselves to one query type and kept the size of the result relation fixed. We chose to focus on the pipeline join algorithm, which is the basis of all complex queries in OBE and which had performed most impressively in our single-user, standalone experiments on both the 4341 and the 3081. In order for the pipeline approach to query processing to be considered viable for MMDBS's, it must hold up under paging.

We tested 8 joins, that join between 2 and 5 relations. The total number of tuples in the relations range from 3,000 to 30,000, and the total size of the relations range from 828,000 bytes to 9.32 megabytes. The minimum amount of virtual memory required to execute the joins ranged between 8 and 16 megabytes. As with the joins in Section 4.6, all of the joins were one-to-one joins on the key that retrieved all 1,000 tuples from the 1,000 tuple relation. Thus, we limited the size of the result through the size of the smallest relation, without using a selection. In addition, all 16 attributes, 13 integers and 3 strings, in the 1,000 tuple relation were retrieved. In order to compensate for an unrealistically large result, the result was retrieved through a "P." that resulted in the result being stored in a space-efficient pointer area. In addition, retrieving the result through a "P." allowed the pipeline algorithm to be short-circuited once it was determined that a tuple in the 1,000 tuple relation would join with at least 1 tuple from every other relation in the join. Finally, from previous experiments, we knew that these joins were correctly optimized (the optimizer put the smallest relations at the head of the query processing pipeline).

Table 22[7] shows the single user, standalone elapsed times of our test join queries on the 4341 only. Due to limited access to the machine, we ran each query once with each of the 4 different memory allocations. We logged off and logged back on between each query in order to insure that each query was run with virtual memory in a similar state.

Looking at the numbers reported in Table 22, we make the following observations:

Table 22: Execution Time of Join Queries under Paging

Time as function of amount of Real Memory available
Memory Required ≡ Virtual memory in megabytes (4 meg granularity)
Time is in Seconds, Operand sizes in thousand tuples

Sizes of Join Relations	Memory Required	Time with 16 Meg	Time with 12 Meg	Time with 8 Meg	Time with 4 Meg
1K,2K	8	9.80	9.76	10.00	10.62
1K,5K	8	10.01	10.02	10.30	17.88
1K,10K	12	10.07	10.19	10.82	30.83
1K,2K,5K	8	12.92	12.92	13.03	29.46
1K,2K,10K	12	13.03	13.00	16.45	41.42
1K,2K,2K,5K	8	15.83	16.04	16.15	30.34
1K,2K,5K,10K	12	15.98	16.42	30.53	91.90
1K,2K,5K,10K,10K	16	19.97	36.43	75.68	241.08

[7] Some of the queries in Table 22 are shown in Appendix 3 under the following names: (Join2 = 1K,10K), (Join3 = 1K,2K,10K), (Join4 = 1K,2K,2K,5K), (Join5 = 1K,2K,5K,10K,10K).

(1) When 16 megabytes of real memory was available, no paging occurred with any of the joins.

(2) With 16 megabytes of real memory available, execution time was strictly increasing according to the number of relations involved in the join, with the number of tuples in the relations having a secondary effect. This is most clearly seen contrasting the 1K,10K join with the 1K,2K,5K join. With 16 megabytes of real memory, the 1K,10K join took almost 3 seconds less than the than the 1K,2K,5K join. However, this effect is reversed when the amount of real memory available is far less than the virtual memory required to execute the queries.

(3) When only 4 megabytes of real memory were available, the execution time for the joins was strictly increasing with the total size of the relations joined, and the number of relations involved in the join had a secondary effect. In this case, the 1K,10K join required 1.37 seconds more than the 1K,2K,5K join.

(4) Except when the virtual memory requirement of the join exceeds by far the amount of main memory available, our measurements show that the performance of the pipeline joins degrades gracefully as the amount of paging increases.

A number of interesting problems are raised by this initial experiment but are beyond the scope of this paper. First, a study of locality in database referencing would provide a more general understanding of the effect of paging on a MMDBS. Second, one may investigate the need for new paging policies, tuned to database management in MMDBS's, in analogy with the need for buffering strategies in operating systems for DDBS's [St81].

6. SUMMARY: A CASE STUDY FOR MMDBS's

We have presented the results of a comprehensive benchmark of the relational MMDBS that is the foundation of the interactive office system, **Office-By-Example**. In Section **6**.1 we summarize our conclusions about the performance of OBE. In the course of evaluating OBE, we have identified issues that must be considered in the design and implementation of MMDBS's. In Section 6.2, we discuss these issues.

6.1. The Performance of OBE

We evaluated OBE with respect to the 2 goals that directed its design:

(1) The system must be user-friendly and truly interactive [Gr83].

(2) The system must have an interactive response time on simple and complex retrieval queries comparable to an interactive full screen editor. With relations occupying up to a total of 2 megabytes of memory, an acceptable response time would be preferably under 1 second, but up to 5 seconds, depending on the query.

In comparison to other systems we have benchmarked, we found that OBE supported a very user-friendly DBMS with full relational capabilities. The two dimensional interface was both easy to learn and to use. Update queries were interactive in that the system provided extensive consistency checking with constructive intermediate feedback, and the option to safely abort. The method of displaying the result was also interactive. The first 8 tuples in the result were displayed and the user was given the option of viewing as much of the remainder as desired. In the prototype that we evaluated, the user and, in particular the benchmarker, was relieved of all responsibility for physical design. One drawback we found was that result tables could not be queried as relations unless the result was specifically retrieved into a predeclared relation.

On a cross section of simple and complex interactive retrieval queries, all queries except 3 executed in .15 to 8 seconds. This is a good level of performance, considering that we used a 10,000 tuple relation that occupied 3.5 megabytes of memory, the same very large relation we previously used to benchmark commercial disk database systems [BDT83]. Furthermore, no alternative data structures, such as clustered relations or hashed indices, were explicitly built to achieve acceptable performance for any query. Since there was only one physical design option, automatically implemented by the system, these numbers represent both the best and worst case performance of the test queries.

We identified two performance problems. The first accounted for the 3 interactive queries, a projection, an aggregate and an aggregate function, that had response times between 10 and 37 seconds. These queries only performed unacceptably with one of 3 result output modes, a "P." on the query table. This result output mode involved the use of the space efficient data structure, a **pointer area**, that resulted in poor performance when comparisons between tuples referenced in pointer area were made. It is not clear whether the high cost of comparisons is inherent to this data structure, or whether its desirable space conserving properties can be retained through careful optimization of the code.

The second performance problem was identified through the failure of a bulk update of 100 tuples that required a 3.5 megabyte relation to be copied to a larger segment of contiguous virtual memory. While there was enough total virtual memory to accomodate the enlarged relation, the update failed due to fragmentation of virtual memory.

We also identified the tradeoffs involved in the design alternatives chosen for OBE. Every desirable feature of the system had a related cost. The user-friendly two-dimensional interface could result in significant parsing time. It took .51 seconds to parse a relation template with 16 attributes. Consistency checking for updates cost time, a minimum of 3.51 seconds for a bulk update involving the key of our 10,000 tuple test relation. Variable length fields increase the space required to store tuples and increase the cost of comparisons, particularly between numbers. OBE reads a relation into memory when its relation template is invoked in the process of writing a query. This results in improved query response time, when the relation is actually queried, at the cost of space in the form of virtual and real memory. To relieve the user of the problem of physical design, all relations were given the same fully inverted storage structure. This approach costs space and time in that for every specific query type there is almost certainly a specialized physical design that would require less space or would result in better response times.

6.2. Design and Implementation of MMDBS's

In the course of benchmarking OBE we have identified issues that must be considered in the design and implementation of MMDBS's:

(1) Physical memory must be very efficiently used, in order to keep the database memory resident.

(2) Memory Management is a critical problem that general purpose operating systems may not adequately address. The problem appears to be even more critical to MMDBS's than buffer management is to conventional DDBS's.

(3) The speed of the CPU on the host machine directly affects the relative weight of space and time in design tradeoffs.

We have observed four approaches to dealing with limited physical memory. The first is to keep the database small. OBE assumes that a typical user will have a virtual machine with 4 megabytes of virtual memory [AHK85]. Since the operating system and the OBE program require about 2 megabytes of the user's virtual memory, most relations are expected to occupy under 2 megabytes. Because database relations are expected to be small, OBE reads all operand relations into virtual memory before a query is executed.

The second method is to keep relations compact. The linear index used by OBE is a compact index that is feasible only because memory residence allows the **long move** instruction to be used in restructuring it. In a 10,000 tuple relation, 16 adjacent linear indices, occupying a total of 640,000 bytes, could be restructured in 0.02 seconds when a tuple was inserted and 0.14 seconds when a tuple was deleted.

A third approach is to use space efficient algorithms such as the pipeline join algorithm that eliminates intermediate relations. The compact **pointer area** was part of a space efficient algorithm for handling the result of a query.

Finally, the database code can be kept compact. The OBE query processing strategy uses relatively few data structures and algorithms, and avoids adding extra code to optimize special cases. Furthermore, with the use of a shared segment for most code, OBE could limit the amount of code that occupied a user's virtual memory space to 0.3 megabytes.

Memory Management strategies are a critical design parameter for MMDBS's. For systems such as OBE that use virtual memory management as supported by the host machine, **memory allocation**, **paging** and **fragmentation** are important issues. In OBE, the optimizer was responsible for estimating the amount of virtual memory that had to be allocated for intermediate results and output of a query. Correct estimation was crucial to obtaining acceptable performance. Since the amount of virtual memory available almost always exceeds real memory, the effect of paging must be considered. In particular, the locality of reference of algorithms must be evaluated. OBE's algorithms were not optimized with respect to any page replacement algorithm. While the pipeline join strategy performed well with paging, OBE's other query processing algorithms require evaluation. Fragmentation of virtual memory can degrade performance by increasing the cost of memory allocation or by causing failure of the system. The latter effect was observed with a bulk update of a 3.5 megabyte relation in OBE.

We found that MIPS rate was a good predictor of relative performance on different machines when the real memory requirements of the MMDBS were met. OBE was developed on a machine that has a fast CPU, the IBM 3081. Certain strategies which incurred an acceptable overhead on the 3081, such as always scanning a relation to perform range queries, may not be feasible on a machine with a slower CPU.

7. CONCLUSIONS and FUTURE RESEARCH

In this paper we have presented the results of a comprehensive benchmark of the relational MMDBS that is the foundation of the interactive office system OBE. In the course of evaluating OBE, we have identified issues that must be considered in the design and implementation of MMDBS's. We have determined the relevant metrics and developed techniques for benchmarking MMDBS's.

In the performance evaluation of MMDBS's, it is important to consider the combined space and time requirements for the execution of queries, as opposed to simply evaluating response times. Our first step was to determine the memory requirements for different queries. In OBE, all of our test queries executed in between 8 and 11 megabytes of virtual memory.

To evaluate query processing algorithms, a necessary baseline measure is the response time of a query when its memory requirements are met and when all relations involved in a query are already memory resident. These are the primary measures we obtained with OBE. In addition, the cost of reading data into virtual memory must be quantified. In OBE, a 3.5 megabyte relation could be read in from a 3380 disk in 2.47 seconds. When memory requirements are not met, the effect of paging must be measured. We have presented a controlled experiment to measure the effect of paging.

Finally, the impact of memory management techniques must be quantified. In addition to measuring the degradation caused by paging, the cost of memory allocation and the effects of fragmentation must be isolated. We designed our experiments to minimize the effect of fragmentation when that was not the aspect of the system under consideration. Determining efficient memory management techniques for MMDBS's is an important direction for future research.

APPENDIX 1

One of the hazards of benchmarking a prototype during its development is that changes in the system can quickly render performance measures obtained obsolete. The experiments in Sections 4.1 and 5.2 were performed several months before the remaining experiments in this paper. In the intervening period, some changes were made in OBE that improved its performance. We wanted to repeat these experiments with the interactive queries we developed on the most current version of OBE. Unfortunately, both experiments required access to a 4341 in standalone mode and the machine we had used was no longer available.

It is very unlikely that our conclusions concerning MIPS rates in Section 4.1 would be changed by the slightly different version of OBE. However, the performance of OBE in the paging experiment in Section 5.2 would probably be improved as one of the changes made the relations more compact. To quantify the change in OBE's performance, we repeated a few of the queries that were used in Sections 4.1 and 5.2. The results are in Table 23 below. For these queries, on the average, the response time after the system was changed was 15.8% faster. The query that improved the most was **Select100-Rg+** with a 23.3 % improvement. The query that improved the least was **Update1-Key+** with an 11% improvement.

Table 23: Change in Performance of OBE
After Measurements for Sections 4.1 and 5.2 were Taken

Average CPU Time in Seconds on Loaded System

QUERY	OLD CPUtime	NEW CPUtime	COMMENT
Update1-key+	4.71	4.19	Update 1 tuple on key
Select100-Rg+	1.29	.99	Select 100 on key
Join2	1.58	1.29	Join 1K,10K tuple relations on key
Join3	2.01	1.73	Join 1K,2K,10K tuple relations on key
Join4	2.49	2.18	Join 1K,2K,2K,10K tuple relations on key

APPENDIX 2

In our initial experiments with OBE, we used queries that we had used in benchmarks of several DDBS's [BDT83,BT84,BT85] in order to get a comparative evaluation of OBE's performance. The queries shown in Section 4.1 are the some of the queries we used in [BDT83]. We soon realized, however, that there were two problems with this particular cross section of relational queries: 1. We had omitted some query types that were important in the environment for which OBE was designed. 2. The size of the result in most of the queries in [BDT83] was too large for an interactive OBE user who would view the result on a terminal screen.

In [BDT83], we omitted two important query types:
(1) **Complex Joins** - The designers of OBE assumed that the typical OBE database would consist of numerous small relations. They also assumed that a large percentage of a typical user's queries would be complex retrieval queries involving a join of 2 or more relations. In our tests of OBE, we tested joins of 2 to 5 relations.
(2) **Bulk Updates** - It was also assumed that except for the occasional interactive update, most updates of the database would be done in bulk when there was only one user of the relation(s) being updated. Therefore, we tested bulk updates of 1, 5, 10, 100, 500 and 1000 tuples.

The result size of most queries in [BDT83] was inappropriate for an interactive benchmark for two reasons:
(1) The number of tuples in the result was too large. For selections, the size of the result varied between 1, 100 and 1000. Projections retrieved either 100 or 1000 tuples. The aggregate functions had 100 partitions. The joins all had 1,000 tuples in the result.
(2) The number of attributes in the result was too large. For selections, all 16 attributes from the relation **TenK** were retrieved. For joins, all attributes from all relations involved in the join were retrieved. The two-way joins had 32 attributes in the result, 26 integers and 6 strings. The three way join had 48 attributes in the result, 39 integers and 9 strings. Each string contained 52 characters. For the benchmark of OBE, we developed a **standard interactive result** consisting of 10 tuples, with 5 attributes in each tuple. (See Section 3.3.2).

Two join queries reflect the difference between the queries in [BDT83] and the ones we used in the benchmark of OBE. The first, **JoinAselB+** is from [BDT83] and corresponds to the Quel query:

```
range of t is TenKf
range of w is TenKe
retrieve into tempsell(t.all,w.all) where
(t.unique2f = w.unique2e) and w.unique2e < 1000
```

The second, **JoinAselB**, is a join more appropriate to an interactive environment. It corresponds to the Quel query:

```
range of t is TenKf
range of w is TenKe
retrieve into tempsell(w.unique1e,w.unique2e,t.tenf,t.thousandf,t.string4f)
where (t.unique2f = w.unique2e) and w.unique2e < 10
```

Both queries are shown in Figure 8. Compared with a strict translation into the OBE interface, shown in Figure 8.a, **JoinAselB+** is easier to write in Quel. In contrast, **JoinAselB+**, as phrased in Figure 8.b, is very easy to write because user-created output is not used. However, the result is presented in two tables: the tuples from each relation that joined were not juxtaposed. Compared to its Quel version, **JoinAselB**, in Figure 8.c, is clearly easier to write in the OBE interface, as the names of the attributes being joined or retrieved do not have to be remembered by the user.

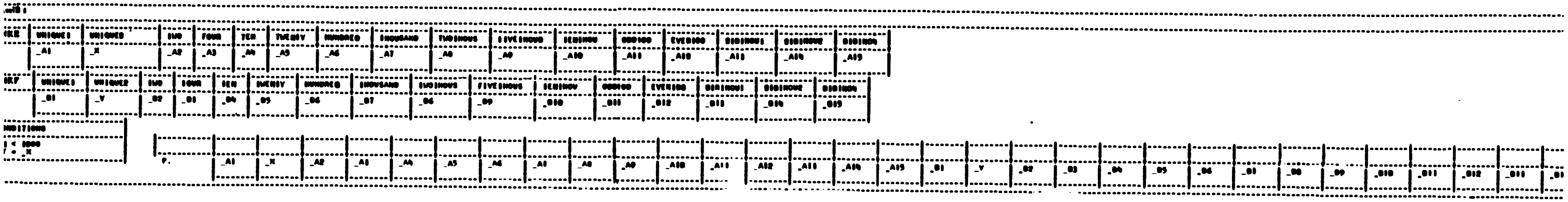

FIGURE 8.a *JoinAselB +* strictly translated from [BDT83].

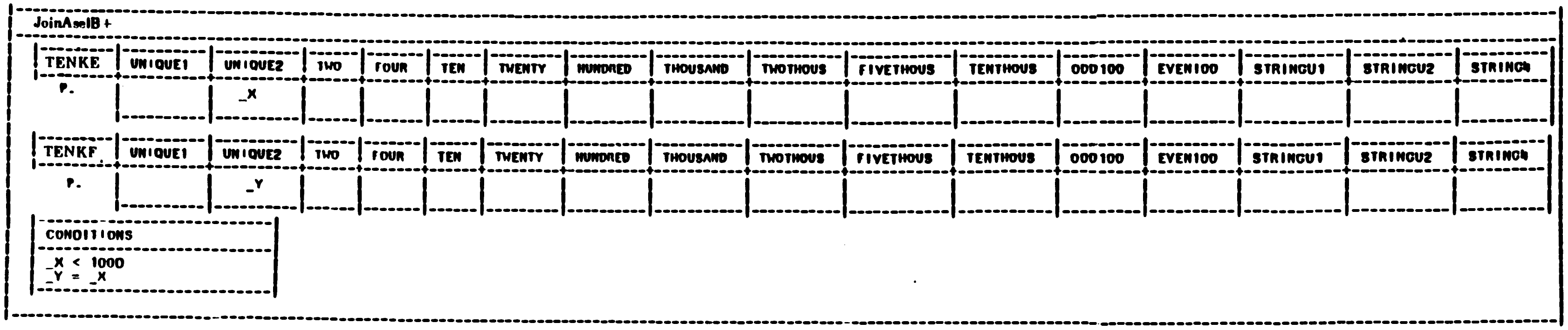

FIGURE 8.b *JoinAselB +* rephrased.
Time for this query given in Section 4.1.

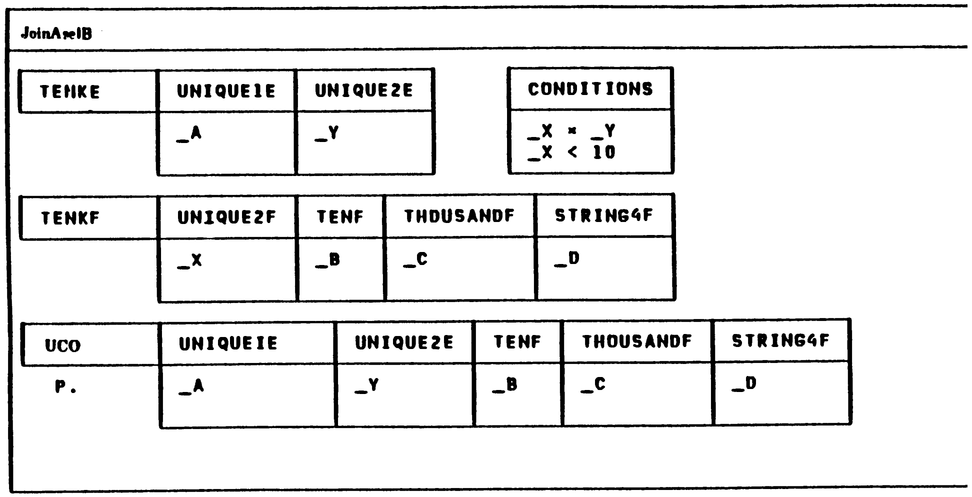

FIGURE 8.c *JoinAselB* rephrased as a *Standard Interactive Query*.
Time for this query given in Section 4.6.

APPENDIX 3

Delete1

TENK	UNIQUE2
D.	=8075

Delete10

TENKTUPE	THOUSANDE
D.	500

InitParse

DUNNY	OO
P.	< 10

InsertaH

TENK	UNIQUE1E	UNIQUE2E	. . .	STRING4E
I.	7825.25	8075.25		NNxxxxxxxxxxxxxxxxxxxxxxxxNxxxxxxxxxxxxxxxxxxxxxxxxH

Insertkey

TENK	UNIQUE1	UNIQUE2	. . .	STRING4
I.	7825	8075.25		NxxxxxxxxxxxxxxxxxxxxxxxxNxxxxxxxxxxxxxxxxxxxxxxxxN

Join2

ONEK	UNIQUE1A	UNIQUE2A	TWOA	FOURA	TENA	TWENTYA	. . .	EVEN100A	STRINGU1A	STRINGU2A	STRING4A
P.		_X									

TENKF	UNIQUE2F
	_X

Join3

ONEK	UNIQUE1A	UNIQUE2A	TWOA	FOURA	TENA	TWENTYA	. . .	EVEN100A	STRINGU1A	STRINGU2A	STRING4A
P.		_X									

TENKF	UNIQUE2F
	_X

TWOKB	UNIQUE2B
	_X

Join4

ONEK	UNIQUE1A	UNIQUE2A	TWOA	FOURA	TENA	TWENTYA	. . .	EVEN100A	STRINGU1A	STRINGU2A	STRING4A
P.		_X									

TWOKB	UNIQUE2B
	_X

FIVEK	UNIQUE2D
	_X

TWOKC	UNIQUE2C
	_X

Join5

ONEK	UNIQUE1A	UNIQUE2A	TWOA	FOURA	TENA	TWENTYA	. . .	EVEN100A	STRINGU1A	STRINGU2A	STRING4A
P.		_X									

TENKF	UNIQUE2F
	_X

TWOKB	UNIQUE2B
	_X

FIVEK	UNIQUE2D
	_X

TENKE	UNIQUE2E
	_X

JoinAB'

TENKE	UNIQUE1E	UNIQUE2E
	_A	_Y

CONDITIONS
_X = _Y

BPRIME	UNIQUE2Z	TENZ	THOUSANDZ	STRING4Z
	_X	_B	_C	_D

UCO	UNIQUE1E	UNIQUE2E	TENZ	THOUSANDZ	STRING4Z
P.	_A	_Y	_B	_C	_D

JoinCondAandB

TENKE	UNIQUE1E	UNIQUE2E
	_A	_Y

CONDITIONS
_X = _Y _Y < 10 _Z < 10 _Y = _Z

TENKF	UNIQUE2F
	_Z

BPRIME	UNIQUE2Z	TENZ	THOUSANDZ	STRING4Z
	_X	_B	_C	_D

UCO	UNIQUE1E	UNIQUE2E	TENZ	THOUSANDZ	STRING4Z
P.	_A	_Y	_B	_C	_D

MinKey-P (with result)

TENK	UNIQUE2
P.	MIN._X

TENK	UNIQUE2 MIN
	0

MinFn-P (with result)

TENK	UNIQUE2E	TENE
P.	MIN._X	g.

	TENE
4	9
15	8
3	7
0	6
13	5
12	4
31	3
9	2

Project20-P

TENK	TWO	FOUR	TEN	STRING4
P.				

Project20-R

TENK	TWO	FOUR	TEN	STRING4
	_A	_B	_C	_D

REL	A	B	C	D
I.	_A	_B	_C	_D

Project20-U

TENK	TWO	FOUR	TEN	STRING4
	_A	_B	_C	_D

UCO	A	B	C	D
P.	_A	_B	_C	_D

Project100

TENK	HUNDRED	FOUR	TEN	STRING4
P.				

Project8K

TENK	THOUSAND	FOUR	TEN	STRING4
P.				

Select10-Eq

TENKTUPE	UNIQUE1E	UNIQUE2E	TENE	THOUSANDE	STRING4E
P.				= 838	

Select10-Rg2 (on candidate key)

TENK	UNIQUE1	UNIQUE2	TEN	THOUSAND	STRING4
P.	_X				

CONDITIONS
_X > 5838 _X < 5849

Select100-Rg+

ONEK	UNIQUE1A	UNIQUE2A	TWOA	FOURA	TENA	TWENTYA	. . .	EVEN100A	STRINGU1A	STRINGU2A	STRING4A
P.		< 100									

REFERENCES

[AEH84] Alanko, T.O., Erkio, H.H.A., and Haikala, I.J., "Virtual Memory Behavior of Some Sorting Algorithms", *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 4, July 1984.

[AHK80] Alanko, T.O., Haikala, I.J. and Kutvonen, P.H., "Methodology and Results of Program Behavior Measurements", *ACM Sigmetrics, 7th IFIP W.G.7.3.*, 1980

[AHK85] Ammann, A., Hanrahan, M., and Krishnamurthy, R., "Design of a Memory Rendent DBMS", *Proceedings of IEEE COMPCON*, 1985.

[BDT83] Bitton, D., DeWitt, D. and Turbyfill, C., "Benchmarking Database Systems - A Systematic Approach", *Proceedings of VLDB*, 1983.

[BT84] Bitton, D. and Turbyfill, C., "Design and Analysis of Multiuser Benchmarks for Database Systems", *Technical Report 84-589*, Cornell University.

[BT85] Bitton, D. and Turbyfill, C., "Evaluation of a Backend Database Machine", *Proceedings of HICSS*, January 1985.

[BCH84] Bogdanowicz R., Crocker M., Hsiao D., Ryder C., Stone V., Strawser P., "Experiments in Benchmarking Relational Database Machines," *Database Machines*, Springer Verlag 1983.

[BD84] Boral, H. and DeWitt, D., "A Methodology for Database System Performance Evaluation", *Proceedings of Sigmod*, 1984.

[Bu76] Buzen, J., "Fundamental Operational Laws of Computer System Performance", *Acta Informatica 7*, 1976.

[De80] Denning, P.J., "Working Sets Past and Present", *IEEE Transactions on Software Engineering*, Vol. SE-6, No. 1, January, 1980.

[DKO84] DeWitt, D., Katz, R., Olken, F., Shapiro, L., Stonebraker, M., and Wood, D., "Implementation Techniques for Main Memory Database Systems", *Proceedings of the 1984 SIGMOD Conference on Management of Data*, June 1984.

[Gr83] Gray, J., "Practical Problems in Data Management", A position paper, *Proceedings of SIGMOD*, 1983.

[Gr85] Gray, J., "The 5 Minute Rule", *Technical Note*, Tandem Computers, May 1985.

[HL84] Heidelberger, P. and Lavenberg, S.S., "Computer Performance Evaluation Methodology", *IEEE Transactions on Computers*, Volume C-33, Number 12, 1984.

[IBM83] IBM, "VM/SP System Programmer's Guide", Release 3, 1983.

[KH84] Krishnamurthy, R. and Hochgesang, G.T., "Architecture for an Universal Office System", *JCIT*, 1984.

[KM84] Krishnamurthy, R. and Morgan, S.P., "A Pragmatic Approach to Query Processing", *VLDB*, 1984.

[Kn73] Knuth, D.E., "The Art of Computer Programming", Volume 3, pp. 145-147, 1973.

[KMZ84] Krishnamurthy, R., Morgan, S.P. and Zloof, M.M., "Query-By-Example: OperationsOn Piecewise Continuous Data", *IBM Research Report*, 1983.

[LC85] Lehman, T.J., and Carey, M.J., "A Study of Index Structures for Main Memory Database Systems", *Technical Report 605*, University of Wisconsin, July 1985.

[Po80] Power, L.R., "Internal Sorting Using a Minimal Tree Merge Strategy" *ACM Trans. on Math. Soft.*, 6:1, March 1980.

[Po85] Potter, D.H., Verbal communications, IBM T.J. Watson Research Center, 1985.

[Sh85] Shapiro, L.D., "Join Processing in Database Systems with Large Memories", *Technical Report*, North Dakota State University, December 1985.

[SK84] Sockut, G. and Krishnamurthy, R., "Concurrency Control in Office-By-Example", *RC10545*, 1984.

[St81] Stonebraker M., "Operating System Support for Database Management", *CACM*, 24:7, July 1981.

[Wh85] Whang, K.Y., "Query Optimization in Office-By-Example", *IBM RC 11571*, December 1985.

[WAB86] Whang, K.Y., Ammann, A., Bolmarcich, T., et al, "Office-By-Example, An Integrated Office System and Database Manager", *IBM Research Report*, 1986.

[Zl75] Zloof, M.M., "Query-By-Example", *AFIPS Conference Proceedings*, National Computer Conference 44, 1975.

[Zl82] Zloof, M.M., "Office-By-Example: A Business Language that Unifies Data and Word Processing and Electronic Mail", *IBM Systems Journal*, vol. 21, No. 3, 1982.

ACKNOWLEDGEMENTS

We are grateful to the members of the OBE group for making this research possible. Moshe Zloof and Ravi Krishnamurthy initiated this cooperative venture, and Ravi enthusiastically described the OBE prototype to us. Art Ammann initiated the main-memory residency idea for OBE, and gave us valuable insight on the significance of this design decision. Art Ammann, Tony Bolmarcich, Maria Hanrahan, Richard Hansen, Kevin Hoffman, Ravi Krishnamurthy, Vance Waddle, and Kyu Young Whang all deserve thanks for their helpful and competent assistance. Many other staff members at IBM Yorktown were very helpful: Susan Betz, Flo McGuire, Tom Pecoraro, Linda Rubin, and Olympia Supple. We thank Patricia Goldberg, and Art Ammann for authorizing publication of this work.

Special thanks are due to Maria Hanrahan who helped us in every stage of the benchmarking: refining the benchmarking module and the test queries, running standalone runs during weekends, explaining query processing, and correcting many very rough drafts. Dave Potter spent a great deal of time obtaining data from the monitor and interpreting it with particular reference to both reliability and validity. Kevin Hoffman wrote the original benchmarking program that we modified to suit our purposes. Tony Bolmarcich wrote the routines to access the TOD clock and to handle terminal I/O, and was the last word on any bug we were trying to track down. Kyu Young Whang explained the query optimizer and gave us detailed feedback on drafts of this paper. At Cornell: George Boyce solved technical difficulties in the unloading and transfer of the raw data. Jeffrey Millman and Hal Perkins carefully edited several drafts of the paper. Jeffrey Millman also constructed the figures.