EVALUATION AND ENHANCEMENT

OF THE

PERFORMANCE OF RELATIONAL DATABASE MANAGEMENT SYSTEMS

by

Paula Birdwell Hawthorn

Memorandum No. UCB/ERL M79/70

7 November 1979

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Evaluation and Enhancement of the Performance of Relational Database Management Systems

Paula Birdwell Hawthorn

Sponsor: National Science Foundation MCS-75-03839-A01

ABSTRACT

The concurrent advances in the understanding of data management systems and in the improvements in computer architecture technology have led to interest in special-purpose computers designed to enhance the performance of data management systems. In order to judge the relative usefulness of these machines, their performances can be compared to standard systems. However, there is little data available to indicate the actual performance characteristics of standard data management systems. Also, it is not clearly understood what types of applications are best suited to database machines.

In order to determine the performance characteristics of a standard database system, the performance of the relational data management system INGRES is analyzed. A benchmark analysis technique is used for the performance evaluation. The INGRES system and the UNIX operating system were instrumented and the benchmarks were run in a carefully controlled test environment with the instrumented systems.

There are three sets of benchmarks, corresponding to three

different types of user database management commands ("queries"). These are overhead-intensive queries, which reference little data; data-intensive queries, which reference a large quantity of data; and multi-relation queries, which are found only in relational data management systems.

Since database machines are back-end machines, the functions of setting up the commands and communicating with the user are performed in the front-end system. If the proportion of time spent in such overhead functions is very large, little performance improvement can be gained by the use of a database machine. The proportion of time spent in processing the data (as contrasted with overhead functions) was measured in order to determine the overall effectiveness of database machines.

The amount of sequentiality and locality of I/O references was also measured. This was done to determine the usefulness of large buffers for pre-fetching data pages, and for retaining those already referenced. The measurement of locality also gives an indication of the effectiveness of those database machines that rely on caching data.

The results show that the performance characteristics of two query types: data-intensive queries and overhead-intensive queries, are so different that it may be difficult to design a single architecture to optimize the performance of both types. Significant sequentiality of reference was found in

the data-intensive queries. It is shown that back-end data management machines that distribute processing toward the data may be cost effective only for data-intensive queries. It is proposed that the best method of distributing the processing of the overhead-intensive query is through the use of intelligent terminals. It is shown that there is significant locality of reference in the multi-relation queries.

The performance of the INGRES system is compared to estimated performances of several database machines. A numerical analysis is performed for each of five database machines, simulating the machine's performance in executing each of four queries. The results are compared to the performance of a standard system. It is found that for significant classes of queries a standard system provides the most cost-effective processing of the queries.

Chapter 1

Introduction

1. Chapter 1

1.1. Introduction

The concurrent development of data management systems and computer architecture technology has led to interesting conjectures about the use of the advanced technology to enhance the performance of the data management systems [HSIA79]. The focus of this thesis is the projected performance enhancement gained by the use of two architectural revisions. The particular revisions are discussed in Section 1.2. Section 1.3 is the explanation of the methods used in this thesis to predict the performance changes which would result from the revised architectures. Section 1.4 contains the conclusion.

1.2. Proposed Architecture Revisions

The following two computer system architecture designs are of particular interest in the performance enhancement of data management systems.

1.2.1. Multiple Processors

One of the effects of the rapid decline of the cost of processing units is that it is feasible to use multiple proces-

sors in a single computer system [SU78]. In systems designed for data management, a simple method of scheduling those processors is to dedicate different processors to different functions: some for data manipulation, some for terminal monitors, and so on. This is the multi-processing of the logical functions of the data management system, using one or more processors per function. Although no previous published data management system architectural design has embodied the use of multiple processors for logically separate functions, the principle of the use of separate processors for separate functions is well known. For instance, the CDC 6000 machines were designed with multiple, special purpose processors [THOR70].

Another method of using multiple processors is to use them as array processors, many performing the same logical function at once. In most designs, the action the processors perform is to process the data as it is read from the mass memory. This design has been proposed in several systems [OZKA75, SLOT75, BAUM76, DEW78A] and is the basis for most special-purpose architectures for data management (database machines).

1.2.2. Extended Storage Devices

An extended storage device is any storage system with access times between those of moving-head disks and very fast random access memory. This includes fixed head disks, bubbles,

CCDs, and slower, cheaper semiconductor random access memory [HSI76B]. Extended storage devices are used in data management systems to cache the database. If a reference is made to data residing in an extended storage cache the access time will be faster than if the data is in the slower mass memory.

1.3. Performance Improvement Analysis

In order to predict the effect that a different architecture will have on the performance of a data management system, particular performance characteristics of the system must be examined. In chapters three and four the performance analysis of the data management system INGRES [STON76] is described. Chapter three contains the description of INGRES, the operating system it runs on, and the software trace facility used to measure the performance of INGRES. Chapter four contains the results of the performance analysis of INGRES.

The method of analysis used was to construct benchmark query streams, and measure the performance of the system as it executed the query streams. Three sets of benchmarks were developed from user query streams. Each benchmark set was constructed to correspond to a particular query type. One set was composed of queries that are defined as "overhead-intensive" because the majority of the time spent in executing them is spent in the overhead functions. The second set

were the data-intensive queries. These were queries that spent the majority of the execution time actually manipulating the accessed data. The third set were multi-relation queries. A complete explanation of the benchmark queries is contained in chapter four.

The performance of the system was analyzed with respect to the improvement possible through the use of multiple processors and extended storage devices. An explanation of the measurements made, and their significance, is contained in the following paragraphs.

1.3.1. Determining the effect of multiple processors

To determine the utility of functionally separate multiple processors the logical functions of the data management system must be identified. The amount of time spent in each function must be determined. Therefore, the major functions of the INGRES data management system were identified and the time associated with each function measured. It is shown in chapter four that the overhead-intensive queries spend as much as 90% of their execution time performing functions that can easily be delegated to a special purpose processor (an intelligent terminal). It is shown that the data-intensive and multi-relation queries spend a smaller percentage of time in the overhead functions. It is also shown that systems which allow multiple processing of the data manipulation functions can result in the faster processing

of the multi-relation and data-intensive queries.

1.3.2. Extended Storage Devices

Extended storage devices are used as large buffers. The problem of determining the effectiveness of the use of large buffers is essentially the problem of determining the data reference patterns. If the data is referenced randomly across the database, the buffer size must approach the size of the database in order increase performance through obtaining a large percentage of hits in the buffer [RODR76]. There are two data access patterns for which large buffers result in reduced data access time. These are discussed in the following paragraphs.

1.3.2.1. Locality of access

A process exhibits locality of reference to its data pages if, once a page has been referenced, there is a high probability that it will be referenced again. It has been shown that most processes do exhibit locality of reference to their program pages [DENN68, CHU72, LEWI73]. There remains the question of the reference behavior of a process with respect to file references [RODR75]. This thesis is especially concerned with one type of process, data management systems. The problem of characterizing data management system I/O references has been extensively discussed in the literature. Chapter two contains a summary of the relevant

research in that area. In chapter two it is shown that previous analyses of the performance patterns of data management systems resulted in the conclusion that there was very little locality of data reference [RODR76]. In chapter four it is shown that for overhead-intensive queries in INGRES there is a great deal of locality of data references. The differences in the results are because of a difference in data management systems and a difference in analysis techniques. The studies summarized in chapter two were trace analysis studies which apparently did not include a large percentage of overhead-intensive queries.

1.3.2.2. Sequentiality of reference

A data management system exhibits sequentiality of reference to its data pages if, for a given file, the logical page reference n + 1 immediately follows the reference n. If the I/O references are known to be sequential, the data can be stored sequentially on the disk and read into an extended storage device an entire track or cylinder at a time, thus dramatically reducing access time. Therefore the sequentiality of data references was measured for the INGRES system, and the results reported in chapter four. It is shown that, for data-intensive and multi-relation queries, there is a great deal of sequential referencing of data. This is in agreement with the studies reported in chapter 2, where sequentiality of access was found in other data management systems [SMIT76]. Therefore the use of extended storage

devices is indicated for data-intensive and multi-relation queries.

1.3.3. Database Machines

Several special-purpose data management machines have been proposed [LANG78, LIN76, SLOT75, SCHU78, SU75, BAN78A, DEW78A, BABB79]. These machines combine the use of extended storage devices and parallel processing to enhance the performance of data management systems. Chapter five contains an analysis of the performance of the five best known systems. The predicted performance of the systems is compared to the performance of INGRES running on a general purpose computer. It is found that for significant classes of queries the database machines are not cost-effective.

1.4. Conclusion

The organization of the remaining chapters is as follows. Chapter two contains the results of the major performance analyses appearing in the literature. Chapter three is a general description of INGRES and the UNIX operating system. Chapter four contains the results of the benchmark analyses. In chapter five the performance of the major data base machines is analyzed and compared. Chapter six contains the conclusions.

2. Chapter 2

2.1. Introduction

INGRES is a relational data management system. There exist no previous published performance analyses of relational data management systems. This chapter contains summaries of the performance analyses that do exist, which were of hierarchical data management systems. There are two additional major differences between the previously published performance analyses and this one:

1) They were trace analyses, rather than the benchmark analysis used in this thesis.

2) The intent of the analyses was not to discover the possible advantage of using advanced technological designs, which is the intent of this thesis.

It is instructive to examine the performance analyses of other data management systems to determine the generality of the results of this study, and to determine the best method of analysis.

Section 2.2 contains an explanation of the traces which were analyzed, and an explanation of the data management systems that produced the traces. Section 2.3 contains the results

of the trace analyses. It is shown that the method of analysis used, which was simply tracing the execution of "typical user queries" for several days, does not lead to insight into the functioning of the data management system. In section 2.4 a separate area of performance analyses is discussed. These are the analyses concerned with double paging studies. Although not directly related to the performance issues discussed in this thesis, these studies are interesting in that they reveal the difficulties in the operating system/ data management system interface.

2.2. The IMS and AAS Traces The following section contains an examination of the reports of various researchers in the search for and analysis of data reference patterns. Most of the major work in this area consists of analyses of two page reference traces. In Section 2.2 the traces are examined, drawing from the papers that describe them. Then in the Section 2.3 the conclusions that the researchers drew from the traces are described.

2.2.1. The IMS trace

IMS (Information Management System, [IMS360]) is a hierarchical data management system developed and marketed by IBM. The following description of IMS is in part due to Date [DATE75] and in part to Lavenberg and Shedler [LAVE75].

In IMS, data is organized into "segments" in a tree struc-

ture. A segment is composed of related "fields". The root segment, its child segments, their child segments, etc., form a database record. There are several possible physical organizations of the database; in the one used for the trace the organization was HIDAM. In this organization, the access to the root segments is via an index (using an indexed sequential technique). In general, the access to child segments is sequential in top-down-left-to-right hierarchical sequence order (TDLR) except that there are pointers between segments that allow some data to be bypassed. The data is first loaded in TDLR sequence. Additions to the database are handled by attempting to place the added segment physically where it belongs in TDLR order. If there is no room, pointers are placed which chain it to its logical siblings and allow it to be stored in a physical block separate from its place in the logical database.

The trace was made of an interactive installation. In IMS, an interactive user enters a transaction from a terminal. The IMS section that handles the terminal interface translates the transaction to calls to the data language interface ("Data Language/I", DL/I). The program that does this translation is most often a user-written program. The DL/I calls are at a very low level, and navigational in nature (e.g. "get the first department record with department_number = 123"; "get within department the next employee record")

Using the HIDAM storage structure as an example, to obtain a list of all employees in the shoe department first the user program issues a DL/I call to get the department where name = shoe. Then it issues a call to get the first employee within the shoe department. It prints the name, and subsequently issues the call to get the next employee. If the database was previously set up with the definition that employee segments should be linked, the effect of the "get next" is that the next employee segment is immediately accessed via a pointer in the first employee segment. If it is not linked, all the intermediate dependent segments must be scanned until the next employee segment is found. This sequential scan results in a physically sequential scan of the database pages only if there are no added segments chained to other pages.

Obviously, the most efficient access to a segment is a direct access to the root segment. A measure of the efficiency of the database organization, then, is the number of segments that must be searched until the user query is satisfied. This is the segment path length and is the subject of several analyses of this trace.

According to Tuel and Rodriguez-Rosell [TUEL75] the trace consisted of probes inserted into the DL/I program, which, at the completion of the DL/I call, recorded the call type, transaction, terminal, time of day, segment search argument (the list of segments and qualifications of them to be

searched for the call), status code, and returned key. This information was written to the IMS log tape.

The database was copied so the trace could be used to simulate the system activity on a separate test machine. It is stated in [TUEL75]:

> "Shortly before initiation of the trace, the keys of the relevant data bases were dumped onto tape. The keys were dumped in hierarchical order (top-down, left to right) for easy reloading." p.11

By using an ordered dump of the databases, rather than simply copying them as they lay on the disks, all information was lost regarding the disorder of the database that would arise from update activity. Since added segments are chained with pointers and stored away from where they would fit in the logical database, a logically sequential scan of the segments is a physically sequential scan of the data blocks only if the data is close to the condition it was in shortly after loading. Whether or not the database was in relatively clean condition was completely lost by this dumping technique.

The system itself is described as an on-line manufacturing control system, and had an average of 80 terminals active at once. There were five databases, ranging in size from 0.77 MB to 113.65 MB, with a total of 175.77 MB. The system was traced for seven consecutive days of operation. The trace itself can be analyzed to obtain such information as the number of DL/I calls per second, the number of DL/I calls

per transaction, and so on.

2.2.2. The AAS Trace

AAS is an IBM in-house data management system that is similar in design to IMS. The trace of the interactive system was reported by Easton in [EAST75]. The database was 2*10**6 pages; there were an average of 400 terminals active at once. Easton noted that Zipf's law [KNUT73] held for the data references: in the three days traced, only 4 * 10**5 pages were referenced.

Comparing the two traces, it is stated in [EAST77]:

> "...the number of page faults for a particular buffer size was examined over successive intervals of 5*10**5 references. In the case of AAS the fluctuations from interval to interval were small. In the case of IMS, however, the number of faults in one such interval was less than 700, while in another it exceeded 5600....Therefore one must be cautious about assuming that behavior measured or modeled in one time period will be maintained in another time period. A detailed intuitive understanding of the nature of the workload at the time of a measurement can aid in attributing any generality to the results obtained"

However, the technique of tracing a random workload leads to great difficulty in obtaining a detailed intuitive description of the workload for either application.

2.3. The Trace Analysis

In the analysis of the IMS trace done by Tuel and Rodriguez-Rosell, the database was loaded from the dump tape onto a test machine. In order to obtain performance meas-

ures the trace was used as a batch input stream to a specially written interface to an extremely well instrumented version of IMS. It is reported in [TUEL75]:

> Two alternatives exist for the test system: a batch region DL/I system, or an IMS control region with several message processing regions. Initially, the batch implementation has been chosen, because of the relative simplicity of setting up the system. The corresponding on-line environment is now being prepared. The main difference between batch and on-line systems is that in the batch, DL/I calls are serialized, whereas in the on-line system, the DL/I calls are multiprogrammed and contend for system resources. However, since there is little data sharing, the CPU and I/O resources for a set of transactions are considered to be similar in both cases, aside from the CPU overhead in supporting multiprogramming."[p.12].

So what was analyzed in [TUEL75] concerning the I/O performance of the IMS trace is not actually the seven-day trace itself, but a secondary trace taken from serializing the DL/I calls of a working system, and using them as batch input to operate on a newly-loaded database. Serializing the calls results in a major perturbation of the results. It completely changes the validity of the trace as a trace of an interactive system. In particular, the sequential referencing of data in a batch system does not appear as strongly in a multi-process interactive system.

In [TUEL75] the major emphasis was analysis of the data reference patterns for locality and sequentiality. It was discovered that there is almost no locality of reference to segments - that is, there was almost no re-referencing of

segments. It was found that when the segments were grouped into blocks, a high hit ratio was achieved because there was a large amount of sequential referencing of the segments, and grouping them into blocks effectively performs a read-ahead function. The natural conclusion from this discovery is that the databases should be written on the disk in very large blocks. There are two problems with this conclusion. First, it depends on the fact that logically sequential segments will be written physically sequentially on the disk, which is only true in a database that has experienced little update activity. Second, this sequentiality of reference, Tuel and Rodriguez-Rosell point out, is almost bi-modal in nature:

> "...a small number of very long (greater than 200) access path lengths dramatically affect the sample characteristics -- in fact, the 0.1% of calls having long path lengths account for 27.4% of all the segments touched. Although these fractions are undoubtedly characteristic of this particular system, it is believed typical. Thus models of behavior may have to consider at least two classes of calls to characterize relationships between response time and utilization." (p.20)

Where [TUEL75] is an overview of the IMS trace and the serialized trace, [RAGA76] is an in-depth analysis of the serialized trace. The references to segments and to blocks were analyzed for sequentiality and locality. In this paper it is again shown that there is no locality of reference to segments. Also, it is shown that pre-fetching of segments is a better technique than demand fetching the segments.

Storing the segments into disk pages, and doing block reads is also shown to be an advantageous technique. When data is block read, the pages are stored contiguously on the storage device (assumed to be a moving head disk). Then, when one page is read, n others are also read. This is called "block reading" and n is the "blocking factor". The advantage of reading data in blocks is that disk arm motion is saved if the data will eventually be used; the disadvantage is that extra information may not be used and the channel and memory have been impacted by having to handle the extra data. In an IMS system, since the DL/I call is at such a low level, it is not possible to know if the extra data fetched will be used. In [RAGA76] block reading with a fixed n is explored, and various values chosen for n, with the result that reading about four blocks at a time was optimal.

Algorithms for dynamically changing the blocking factor were explored in two studies, [SMIT76 and FRAN76]. The proposal in [SMIT76] is to assume that past history in some way predicts future usage. At each request for data, the system uses information about how many of the past references were sequential references, and pre-fetches several blocks at a time dependent on that knowledge. This is shown to be optimal over a fixed blocking factor, and over demand fetching of blocks. Smith apparently worked with the same serialized trace as in [RAGA76].

A similar analysis is contained in [FRAN76]. In this

analysis, the original IMS trace was converted to a sequence of segment references by use of an algorithm devised by Mommens. The sequence was then used as input to a simulation that mapped the segment references to block references. In [FRAN76] the blocking factor n would be determined from a "learning period" of dynamically keeping track of sequential references to blocks, then fixed for another period of time. This is shown to be optimal compared to demand fetching and compared to having a fixed blocking factor.

In [RAGA74, SMIT76 and FRAN76] the results of experimentation with using a variation of the LRU strategy for buffer replacement by handling un-referenced data pages separately from referenced ones are reported. In [FRAN76] the results showed that the miss ratios for the AAS and the IMS secondary trace derived from the Mommens' sequence were consistently better when the read-ahead blocks were maintained separately than when a standard LRU buffer replacement algorithm was applied. The result in [SMIT76] was exactly counter to that of [FRAN76], although the experimental design was almost identical, and the IMS trace was used in both cases. [RAGA74] contains results similar to [SMIT76]. It is intuitively clear that, in a multi-programmed situation, where one buffer is shared among several independently executing processes, read-ahead blocks will suffer in a LRU replacement scheme, and probably be over-written by the time the process requesting them gets around to using them. This

is entirely dependent on the number of processes concurrently sharing the buffer space, however, since if only a single process is using the buffer, and it is processing the data sequentially, the read-ahead blocks will never be over-written. We therefore can understand why the analyses of the secondary trace of the IMS system that was made by serializing the DL/I calls yield the result that treating the read-ahead blocks differently makes no difference: in a uni-programmed system, it wouldn't. What is puzzling, then, is why the result is different for [FRAN76]. It may be that the Mommens technique successfully simulates multi-programming of the DL/I calls; or, since all these analyses use only a portion of the trace, it may be that in the portion used in [FRAN76] the sequentiality present was a result of inter-query sequentiality (successive DL/I calls referenced successive blocks, a natural outcome of the "get next" call). In [SMIT76] it is reported that the sequentiality present was due to intra-query sequentiality (the result of one DL/I call was that many blocks were read sequentially).

In the [FRAN76] it is also noted that there was one index block reference for every five data block references, a surprisingly high number. It may be these index references that yield the amount of locality apparent from miss ratio curves displayed in [FRAN76, SMITH76 and RAGA74]. Although it is explicitly stated that there is no locality in [RODR76]:

> "Comparison of the two plots [miss ratio functions for IMS and AAS] shows that, in both cases, over range of several orders of magnitude for window size values, the average working set size is almost a linear function of the window size and it does not show the rapid decrease in slope beyond a window size value, so characteristic of VMPS [virtual memory program systems] programs. This plot gives clear indication of the lack of locality in DBS [Data Base Systems]." p10.

the miss ratio curves all show a decrease in miss ratio as the buffer size increases, even for buffer sizes much smaller than the databases. This can be explained by the frequency of referencing the indices.

In an analysis of the AAS trace, Easton found that the terminals referenced data at random within the data base, but that for each terminal there was a high probability of re-referencing the same data page within one transaction, and that there was also some re-referencing of the same data page across transactions from the same terminal. Further analysis revealed that the re-referencing within a transaction was because of sequential reads to the same data page (an average of 40 reads per transaction) and that the inter-transaction re-referencing was due to subsequent transactions referencing the same data. He developed a markov model which expressed the probability of re-referencing a page as a two-phase probability dependent on whether it had been recently referenced, and validated the model to the AAS trace. In [EAST78] a generalization of the model was applied to a trace of IMS logical record references. The model itself is not of primary importance in this

discussion; we are concerned with the data reference patterns exhibited in the traces.

Of the IMS trace Easton remarks:

> "The second reference string analyzed, also of length about 2*10**6 references, is from a measurement of an IMS system that is used for engineering analysis in a manufacturing environment. The original data consisted of a map of the database at the start of the measurement and a sequence of DL/I calls that was subsequently executed. This sequence was constructed by J. Mommens, who developed a technique for expanding each DL/I call. Mommens' sequence was then converted to a sequence of references to 4096-byte pages for this study."

In both the IMS and AAS reference traces, the miss ratio curves plotted by Easton show that, as the buffer size increases, the miss ratio decreases. In the AAS trace, this drop is plotted for buffer sizes extending from 10**3 to 10**5 pages. For the IMS trace, the miss ratio plotted drops from 0.005 at 200 pages in the buffer, to .0012 at 2000 pages in the buffer.

2.4. Double Paging Studies

There have been several studies of the "double paging anomaly", which is defined in this section. These studies have nothing to do with the performance analysis of INGRES since INGRES was implemented on a non-virtual memory machine. This discussion is included for completeness, and to show the problems of buffering data on a virtual memory machine.

From the above studies it can be concluded that buffering the data pages is optimal, and that large buffer sizes are

best. In fact, IMS does buffer the database pages. These buffers are maintained by IMS; a logical decision since the cost of communicating with the operating system per segment would be very high. This implementation of one virtual memory system (IMS) on top of another virtual memory system (the IBM virtual memory operating systems) leads to problems. Since the system maintaining the buffer does not know if a virtual memory page is actually in memory, it can easily assign a page that currently resides on the paging device to a new database page being read in. The operating system, seeing the reference to a page on the paging device, will read that page into memory first, before the data management system can write the new database page into it. This is the "double paging anomaly" identified by Goldberg and Hassinger [GOLD74].

IMS had an even worse problem: the algorithm for obtaining information was to search the entire buffer first, then to go to the disk. If the buffer itself is paged, the buffer searches cause many page faults. This problem was studied by Tuel in [TUEL74], and his conclusion was that the entire buffer should be assigned to real memory, and never be paged.

Sherman and Brice in [SHER76] report that the later IMS systems used a table to find out what was in the buffer, and so had only the double paging problem to deal with. It was shown in [SHER76] that the potential benefits of paging the

buffer outweigh the disadvantage of double paging, and in [BRIC77], an extension of their earlier work, they showed that it is better to manage the page allocation for buffers in a partition separate from the program pages.

In [LANG77] the advantage of using virtual buffers is quantified by using user-supplied probabilities of re-referencing the data. It is shown that buffering to fast paging devices is optimal when the probability of re-referencing the data is high. In [FERN78] the interaction between the two page replacement algorithms (one for the allocation of memory pages, the other for allocation of virtual buffer pages) is explored. An algorithm that takes into account whether or not the buffer page is actually in memory is found to be optimal.

It is also pointed out in [FERN78] that one solution to the double-paging problem is to have better communication between the operating system and the data management system.

2.5. Conclusion

The following points are clear from the above discussion:
1) The use of trace data for performance evaluation must be accompanied by a clear, detailed explanation of the work load and specific programs that created the trace. Otherwise, insight gained by the trace analysis is highly lmited. This is apparent from the sometimes conflicting results of

the AAS and IMS traces. It is for this reason that a benchmark analysis was chosen for the following performance measurements of INGRES. Only in the controlled environment afforded by the benchmark analysis can the precise phenomena measured be understood.

2) Data management systems that support only low-level queries are forced to guess at optimization strategies. The emphasis on predictive buffering strategies in the above literature is only necessary if the system can not determine from the query exactly what data will be needed.

3) When a data management system is implemented with a general purpose operating system, there can be performance problems due to the destructive interference of the two systems. The problem of buffering in a virtual memory system is a case of this destructive interference.

3. Chapter 3

3.1. Introduction

It is impossible to assess the generality of the results of any performance evaluation of a real system unless it is known precisely why the stated results occur. This chapter presents an overview of INGRES and the operating system UNIX [RITC74] so the results of the performance analysis of INGRES that are reported in the next chapter can be properly evaluated.

The first section of the chapter defines some of the terms that will be used in the following three chapters. Section 3.3 is a brief summary of INGRES; section 3.4 is an explanation of UNIX I/O handling. The final section contains the conclusions.

3.2. Definitions

3.2.1. Query

A query is a directive to INGRES to display or modify data or data structures. INGRES queries are written in QUEL, the INGRES query language, and are single-statement commands. There is no notion of a transaction that is made up of several user queries; each query is executed independently.

A query stream is a group of QUEL statements, executed sequentially.

3.2.2. Overhead-intensive and Data-intensive Queries

In this section two query types are defined: overhead-intensive queries, and data-intensive queries. A more complete definition of both types is presented in the following paragraphs. Intuitively, overhead-intensive queries are those that reference little data, which means that the data management system overhead becomes a large component of the query response time. A data-intensive query is one that references a large amount of data, so that the overhead functions are a small portion of its response time. Clearly there is a continuum between the query types. They are presented as two separate types in this analysis because the performance patterns are markedly different on opposite ends of the continuum. The differentiation is similar in concept to the that between "simple" and "batch" queries defined in [GRAY78].

We will define an overhead-intensive query as one for which data processing time is less than system (operating and data management) overhead to process the query. The overhead is the time to communicate with the user, parse and validity check the query, and issue the command to fetch the data. The data processing time is the time to actually fetch and manipulate required data. Therefore, the overhead-intensive

query is a query which references little data. This case arises when the query inherently references little data, and the database has been previously optimized to support the query. For instance,

retrieve (EMPLOYEE.name) where EMPLOYEE.empnum = 1234

will be a overhead-intensive query if there are few employees with EMPLOYEE.empnum = 1234 and if a useful storage structure involving empnum is available. Such a structure exists if the EMPLOYEE relation is hashed on empnum, ISAM on empnum, or has a secondary index on empnum.

Overhead-intensive queries are common in such applications as data entry, banking, airline reservation, inventory control, and customer information systems.

A data-intensive query is defined as a query for which the time to process the data is much greater than the overhead. It references a large quantity of data, and is the other end of the continuum from overhead-intensive to data-intensive queries. A data-intensive query arises from two causes:

1) the query is inherently data-intensive.

If, in the above example, there were two million employees with EMPLOYEE.empnum = 1234, the query would be a long, data-intensive query.

Inherently data-intensive queries are produced any time

there is a complete scan of a large portion of the data, as in the production of periodic reports, billing of large sections of customer accounts, and statistical analyses of large amounts of data for such applications as management information systems.

2) queries for which the database is not well-structured.

In the above example, if the EMPLOYEE relation is not structured on EMPLOYEE.empnum or if it does not have a secondary index on EMPLOYEE.empnum, the entire EMPLOYEE relation will be read.

3.2.3. Multi-relation queries

Multi-relation queries are specific to relational systems. They are queries which reference more than one relation and may be overhead or data intensive depending on the size and the storage structures of the relations referenced.

3.2.4. Self-overlapped CPU and I/O Time

If the I/O and CPU time for a single process is overlapped: that is, if the process is processing data from one disk page while another of its pages is being read, then it has self-overlapped CPU and I/O time. Often processes arrange to do this for themselves, through double-buffering and read-ahead techniques, or the operating system does it for them, through read-ahead or block reading. The advantage to

such a scheme is that the response time for the process is greatly increased on a lightly loaded system.

3.2.5. CPU or I/O bound

A set of queries is I/O bound if a major decrease in the CPU time of the queries has little effect on the speed with which the query set is executed. Similarly, a query set is CPU bound if a major decrease in I/O time has little effect on the execution time.

Whether a system is CPU or I/O bound at a given instant depends on the transaction mix concurrently executing at that point in time. If several queries, each of which is I/O bound when running stand-alone (which is possible due to the self-overlap of CPU and I/O times explained above) are accessing data on different devices the mixture can be CPU-bound or I/O bound. It can be CPU bound if the I/O times of the queries can be overlapped because there are several devices capable of operating concurrently, but their CPU times cannot because there is only one CPU. On the other hand, that same set will be I/O bound if all transactions reference data on the same disk.

Likewise, it is possible that a set of queries, measured to be CPU-bound on a stand-alone system, when run concurrently may become I/O bound because they are all referencing the same device and increasing each others access time due to

the destructive interference of each others I/O references. This destructive interference occurs on moving-head disks when single queries are referencing data physically within the same area on the disk, thus causing the disk arm to move little, but several queries run at the same time are referencing data in physically separate areas, thus resulting in more movement of the disk arm.

The following is an example:
query A has 10 units cpu time, 15 units I/O time;
query B has 12 units cpu time, 5 units I/O time.
When run stand-alone, due to self-overlap features, A is I/O bound, B is CPU bound. When a mixture of two queries of type A is run, there are two possibilities:
1) The I/O is to two different devices, in which case the total time to execute both queries is 20 units of CPU time and 15 units of I/O time (since when one is accessing one disk, the other can access another disk). So the transaction mix is CPU bound.

2) The two queries reference data on the same device, so the time to execute the transaction mix is 20 units CPU time and at least 30 units of I/O time: the mix remains I/O bound.

In case of two queries of type B run concurrently, if the queries reference data on separate devices the transaction mix remains CPU bound. However, if the data for both queries resides on the same device, the destructive

interference can more than triple the I/O time of each query, making the mixture I/O bound. This can occur if, for instance, all the data necessary for one query resides on the same disk cylinder at one edge of a moving head disk, and the data for for the other query resides on another cylinder, on the opposite edge of the disk. When either query is run stand-alone it requires an average access time of half a disk rotation to read each page of data, but when run with together each requires an average access time of the time to move the disk head across the full width of the disk. On most disks, that time is more than triple the latency time. So to run the mixture, it would take 24 units of CPU time and 30 units of I/O time, thus making the mixture CPU bound.

We conclude, then, that although a single query may be I/O or CPU bound, that a set of queries is not necessarily bound in the same manner. Certainly the case of CPU-bound queries becoming I/O bound is a pathological case, and the case of I/O bound queries becoming CPU bound is far more common, since most computer systems have more than one disk. It is also clear that the measurement of importance is not just whether the CPU time exceeds the I/O time, or vice-versa, but the total amount of CPU and I/O time incurred by the query. For that reason, both times are given in the analyses in the following chapters.

3.3. INGRES

3.3.1. Introduction

Complete descriptions of the INGRES data management system are contained in [STON76, WONG76, EPST77]. INGRES is an interpretative relational data base management system. A relation is a table with a fixed number of columns, corresponding to the relation attributes, (called domains), and a variable number of rows (tuples). For the purpose of the following discussion the following two relations are defined:

relation: EMPLOYEE

attributes:	name	empno	salary	dept
	jones	12575	10000	12
	smith	2571	15000	13
	adams	1255	12000	12
	jones	4433	10000	11
	.	.	.	.
	.	.	.	.
	(a partial list)			

relation: DEPT

attributes:	number	name	manager
	12	toy	12575
	13	shoe	2571
	11	candy	4433
	.	.	.
	.	.	.
	(a partial list)		

The INGRES function to display data to the user is "retrieve". The functions of "replace" (to change one or more domain values), and "delete" are implemented as retrieves followed by the appropriate action.

The organization of this section is as follows. Parts 3.3.2

(INGRES Storage Structures) and 3.3.3 (I/O Reference Patterns) present general information about INGRES performance patterns. It is necessary that this general information be presented before the specific information in chapter 4 so that the generality of the results in chapter 4 can be judged. Part 3.3.4 is a detailed explanation of multi-relation queries within INGRES. The INGRES process structure is defined in part 3.3.6.

One of the results of the study presented in chapter four is that INGRES is CPU bound in even stand-alone situations. This is a surprising result, and counter to the usual assumptions made of data management systems [YAO78, OZKA77]. Part 3.3.4 presents general results showing under what conditions INGRES becomes CPU bound.

3.3.2. INGRES Storage Structures

The possible organization for INGRES relations are indexed sequential (ISAM) or hashed on any domain or combination of domains, or a simple heap. Secondary indices are possible for any domain. The user controls the storage structure, and the default storage structure is heap.

example:

create EMPLOYEE (name = c5, empno = i2, salary = i4, dept=i2)

creates a heap structure: tuples are ordered by their order of addition to the data base

If the user decides another structure is more suitable to the application, the command is typed:

modify EMPLOYEE to hash on dept

the relation is now stored in a hash structure, where each tuple is stored according to a function applied to its (not necessarily unique) domain "dept"

A secondary index can be created for any domain, also by the user:

index on EMPLOYEE is empx (empno)

The secondary index is stored in a relation (named, in this case, empx). The default structure for secondary indices is ISAM. The secondary index contains the indexed field, logical page number, and position within the page, for the associated tuple.

3.3.3. Reference Patterns

The data reference patterns possible for a single relation retrieve are:

1) complete scan

A complete scan of the relation is always executed if the relation is a heap, if there is no qualification, or if the qualification is not usable in conjunction with the storage structure.

example:

```
retrieve (EMPLOYEE.name, EMPLOYEE.dept)
```

no matter what the storage structure of EMPLOYEE is, a complete scan will result.

```
retrieve (EMPLOYEE.name)
    where EMPLOYEE.salary < 12000
```

if EMPLOYEE is not ISAM, with salary as the key, and if there is not a secondary index on EMPLOYEE with salary as the key, a complete scan will result.

2) partial scan

A partial scan of the relation is executed if the storage structure is an ISAM relation on one of the qualifications, and that qualification indicates a range of values.

example:

```
retrieve (EMPLOYEE.name) where EMPLOYEE.salary < 12000
```

if EMPLOYEE is an ISAM relation, with salary the key, only that portion of the relation for which salary < 12000 will be read.

This is a sequential scan only if there are few randomly-added overflow pages.

3) random reads

Random accesses to the relation pages occur when the relation is hashed, and the hash key is included in a test

for equality in the qualification, when the qualification contains a domain for which there exists a secondary index, or when the qualification is equality on an ISAM key.

3.3.4. Multirelation retrievals

The query:

retrieve(EMPLOYEE.name) where EMPLOYEE.number = DEPT.manager

is a multi-relation query. Queries involving more than one relation are decomposed into a series of one-relation queries. Several techniques are used to perform this decomposition. They are explained in detail in [WONG76] and [YOUS77]. A simplified overview is shown in the following example, using the above query and the relations defined in 3.3.1.

If both relations are very small, simple tuple substitution is performed, i.e.: each tuple from department is read, and the resulting one-relation query is performed. The resulting queries would be:

query 1:

retrieve (EMPLOYEE.name) where EMPLOYEE.number = 12575

query 2:

retrieve (EMPLOYEE.name) where EMPLOYEE.number = 2571

and so on.

A more likely choice of the INGRES strategy algorithm would

be to first perform the projection:

retrieve into temp1 (EMPLOYEE.name, EMPLOYEE.number)

and the projection

retrieve into temp2 (DEPT.manager)

Now, depending on the size of the relations temp1 and temp2, either temp1 can be reformatted to hash on EMPLOYEE.number, and tuple substitution performed:

query 1:

retrieve (temp1.name) where temp1.number = 12575

and so on

or temp2 can be reformatted to hash on DEPT.manager and tuple substitution performed:

query 1:

retrieve (name = "jones") where DEPT.manager = 12575

query 2:

retrieve (name = "smith") where DEPT.manager = 2571

and so on

The choice of which method to use is made by applying cost estimations. If it appears that, for instance, the cost to modify a temporary relation to hash will be greater than simply re-reading the relation several times, the modification will not be made.

The choices for which strategy to use are data-dependent. The strategy is not fully decided until the temporary relations are formed. When the temporary relation is the result

of a simple projection, as in the example above, the size of the temporary relation can be approximately determined prior to execution. However, if the temporary relation is the result of a restriction and projection (if the clause "and DEPT.name = toy" is added to the above, for instance) the size of the temporary relation is unpredictable prior to execution. Knowing the sizes of the temporary relations, and making strategy decisions dynamically, at execution time, is one of the sources of optimization of INGRES. If a query were completely compiled prior to execution, this source of optimization would be lost.

If tuple substitution is performed without reformatting the relation, the same data is being repeatedly re-referenced, once per tuple substituted. This is the most extreme case of re-referencing. If, for instance, in the above example, EMPLOYEE were hashed on number, each one-relation query would be a random read, and there would be no re-referencing at all. Whether a multi-relation query will exhibit re-referencing depends on the structure and size of the relations and the data distribution of the domains involved.

3.3.5. Conclusion

Since the default method of reading relations is to sequentially scan them, we may expect a great deal of sequentiality in the INGRES data reference streams. The multi-relation queries can cause the reference stream to show re-

referencing.

3.3.6. The Process Structure

INGRES runs as five processes, with a sixth process occasionally created to perform sorts. It is forced to run as separate processes because of the address space limitations on the PDP 11/70 on which it is implemented.

INGRES is an interpretative system, which means that, as a query is entered by a user at a terminal, INGRES parses, optimizes, and executes it. The INGRES processes, their functions and sizes (in bytes) are:

INGRES Process Sizes

monitor	interactive terminal monitor	24K
parser	parses the query	53K
decomp	decomposes the query into one-variable queries; plans strategy for retrieving information	49K
ovqp	one-variable query processor retrieves information	50K
dbu	data base utilities: these are overlays that carry out functions such as destroy, help, modify, print, etc. The size given is for the most used overlay, the one with create, modify, and destroy	48K
ksort	sort routine used by INGRES to sort data; used in conjunction with modify	18K
	total	242K

These processes execute sequentially (when the monitor is finished, it passes control to the parser, then the parser passes control to decomp, etc). Process communication is through "pipes". A pipe is a file that is opened by two processes. INGRES pipes are all one-way, so, for instance, the monitor has one pipe to write to the parser, and one to read from the parser. The cost of the process structure of INGRES is measured both in space and time. One third of each of the process (with the exception of the monitor) is identical access method code. The time to read and write

pipes, and the time to swap in processes that are unneeded except to transmit information through the pipes is not trivial, as will be shown in chapter four.

Each user at a terminal executes a separate set of five processes. The text portion of the processes is re-entrant, so only the data portion (about half of each process) is replicated for each user.

3.3.7. CPU Time

In preliminary performance evaluations of INGRES it was found that the system is often CPU bound. The subject of this section is the determination of when INGRES becomes CPU bound while handling data-intensive queries. In Chapter Four the results of running benchmark queries taken from user query streams shall be reported. In this section, we report the general performance characteristics of INGRES that were obtained by using entirely artificial queries, and varying them to find the sensitivity to parameters of the query. Several query streams were written, and relations created, that differed only in one parameter from the previous one. These parameters were the tuples per page and the number of domains named in each query. A regression analysis was done on the resulting CPU times. Under the given conditions:

1) The query is a simple aggregate, with little output, as in the above counts, or a sum, etc.

2) The query is a one-relation query.

3) The qualification is always true.

4) The domains accessed are small (less than 15 bytes each)

The CPU time spent in the OVQP process for the query is:

CPU = (.01 + (.0012 + .001*md) * tuppage) * pages

where

md = number of manipulated domains. A manipulated domain is one that appears in either the target list or in the "where" clause of the query. If a domain appears more than once, it is counted as many times as it appears.

tuppage = number of tuples per page

pages = number of pages in the relation

and the I/O time is

I/O = pages * .03

If the queries are run stand-alone, or on a lightly loaded system, and if the relation is accessed sequentially (stored as a heap), the I/O and CPU time will be overlapped. In that case, the total time to run the query is the maximum of the CPU and I/O times, plus about a second of overhead.

The above estimator is correct within 5% or 2 seconds, whichever is larger.

The following is a discussion of the effect of relaxing the conditions. Condition (1) limits the output. If output is allowed, the time for it must be included in the estimator. If the output is to a disk file, the extra I/O time is

extra I/O = .03 * ((number of bytes of output)/512 bytes per page))

If the output is to a terminal, the extra I/O time is a function of the terminal speed.

The OVQP CPU time becomes

CPU = (.01 + (.0012 + .001*md + .0007 * nbytes) * tuppage) * pages

where

nbytes = number of bytes of output per tuple

If condition (4), that the accessed domains be small, is relaxed large character domains are allowed, the estimator becomes less precise. If the md factor is increased by one for every accessed non-contiguous domain over 15 characters long, the estimator is correct within 25%.

Conclusion:

A data-intensive INGRES query meeting conditions 1-4 above becomes CPU bound on a stand-alone system when

(.0012 + .001*md)*tuppage > .20

If, for instance, only one domain is being touched, as in the query:

```
retrieve (summ = sum(EMPLOYEE.salary))
```

then there would have to be 100 tuples per page before the query would become CPU bound. Since a page is only 512 bytes, that is impossible.

On the other hand, if several domains are referenced in the query, for instance 10, then at only two tuples per page the query becomes I/O bound.

Since typical user-written INGRES queries reference several domains per tuple, INGRES appears to be usually CPU bound, although there is a large class of queries (the ones with low work-density per page) for which it is not.

3.4. The UNIX Operating System

UNIX handles the INGRES I/O functions. Read commands for logical pages are issued to UNIX which maps that logical address into a physical address, and reads the page. Each page is a 512 byte block of data. If the user define a page to be larger than 512 bytes, UNIX maps the page into two or more 512-byte pages which may not be contiguous on the disk and are separately scheduled in read and write operations.

3.4.1. Read-ahead

UNIX uses a read-ahead algorithm: when a process references two logically sequential pages, the third is also read. From that point on, as each logically sequential page is accessed, the next sequential one is read by the system. For example: if a process reads page 24 and then page 25, when the system reads page 25 it will then read page 26, putting the data in a system buffer. When the process executes the read command for page 26, the information in the system buffer is copied into the user's area, and page 27 is read. The reason for this one-page read-ahead is to increase CPU-I/O overlap. This is not a block-read method: after the initial two pages are read, each page is read one at a time, incurring the average access time for each page reference.

The following table contains the results of experiments to test the effect of the one - page pre-fetch strategy currently implemented by UNIX. In this experiment, two relations were created that contained the same amount of data. One was read logically sequentially by INGRES, so that UNIX would perform the one-block read-ahead. The other was disordered by appending one tuple at a time, resulting in many out-of-order overflow pages, so that the logical block numbers read by INGRES were not in sequential order. In the second case, UNIX did not perform the one-page read-ahead. Note that the decrease in response time due to the

increased CPU-I/O overlap which the one-page read-ahead gives almost disappears by the 3 user case. In the 4 user case the extra work resulting from the loss of the read-ahead pages actually results in the read-ahead query taking more time than the one which does not perform read-ahead.

Table 1: response time for a single query
time in seconds

	number users 1	2	3	4
query A: data read logically sequentially	2.51sec	3.82	6.08	7.84
query A: data not read logically sequentially	4.02	4.43	6.25	7.26

There are two problems with the one-page read-ahead strategy: one is that the buffer space must be large enough to retain the read-ahead blocks until the process requesting them needs them; the other is the hiding of the true execution time from the user. The first problem, the loss of read-ahead blocks, results in extra work done by the system. In the tests performed to produce table one, above, the loss of read-ahead pages was measured. In the four-user case, six percent of them were gone from the buffer by the time the process requested them. Since reading them is at no time free because it is an average access time whether they are read as read-ahead pages or when read on demand this was simply extra work that the system had to do.

3.4.2. Sequential Physical Files

The UNIX operating system, unlike most, does not attempt to implement physical sequentiality of logically sequential I/O blocks. Each physical disk is divided into one or more logical devices, and files are then written to one of these devices. Each INGRES relation is a file. Blocks of 512 bytes each are kept in a "free" list per device. When a new page is added to an INGRES relation, the first block on the free list for the corresponding device is assigned to the added block. UNIX makes no attempt to put the new block near its logical predecessor.

In many other operating systems, a file is opened with a declaration of how large it is likely to become; then the operating system reserves the correct number of disk tracks for it. As information is written in logically sequential pages by the user's process, the operating system writes it physically sequentially on the disk. Often the pages are actually stored in a "checkerboard" order: logical page one is stored one (or even two) pages away from logical page two, and so on. This is to allow time for the data to be transferred into the computer's main memory; otherwise, if page one is next to page two, the time it takes for page one to completely be stored in memory may be great enough that page two will have been missed by the disk head, and a full revolution of the disk necessary to fetch it.

An algorithm to force logical sequentiality to imply physical sequentiality in a UNIX system is to copy all the information from one logical device to another, thus emptying the first; then, order the free blocks on the first device; then, copy everything back. This technique was used to examine the effect of reading logically sequential data physically sequentially from the file. It was found that a query stream that took 133.16 seconds to run (single-user) with the pages placed in the usual UNIX random fashion on the disk ran in 115.05 seconds when the pages were placed in physically sequential order by using the above algorithm. The above algorithm saves time by saving disk seek time; however, if the data had not been placed in strictly sequential order, but stored in "checkerboard" order, an even greater saving would have been effected.

3.4.3. Global Buffer Strategy

UNIX has a global buffer strategy. When a user process requests an I/O block, the information is read from the disk and stored in a system buffer, then copied to the user-defined buffer area in the user process space. When a write is done by the user, the data is copied to a system buffer, and the data moved from there to disk. The total operating system overhead is about six milliseconds per 512-byte page [RITC78]. Buffers are allocated on a least-recently-used basis.

3.5. Conclusion

In this chapter we defined terms that are important to the following analysis. Then overviews of INGRES and UNIX were given. There are two factors that are of particular interest:

1) What is the effect of the mini-computer environment and the UNIX operating system on the performance of INGRES.

2) Can the results from this analysis generalize beyond the specific implementation of INGRES?

The effect of the minicomputer environment is two-fold. First, the CPU times will be slower than in many larger computers. The PDP 11/70 is about a 1 MIP machine. Second, the overhead will be larger because the 16 bit word size forces an address space limitation and the use of multiple processes. The use of UNIX means that the I/O times will be larger than on some other operating systems because UNIX does not support sequentially allocated files. Also UNIX constrains the page sizes to be a constant 512 bytes.

4. Chapter Four

In order to determine the exact performance patterns of INGRES, three sets of benchmark programs were developed. These were sets of overhead-intensive, data-intensive, and multi-relation queries. The benchmarks were run single-user on a DEC PDP 11/70.

This is a 16-bit minicomputer, with a 2K byte cache, and is about a 1 MIP machine. The databases were on a disk with an average access time of .030 seconds per 512-byte block. The version of INGRES used for the benchmarks was version 6.1; the next version (6.2) achieved a 10 - 25% pefrormance improvement over version 6.1. The specific improvement depends on the query type. Future implementations of INGRES can be expected to show increasing improvement.

4.1. Overhead-intensive queries

Three benchmark query streams were developed to determine the performance patterns for overhead-intensive queries. The three benchmarks were taken from a set of user queries. The user is the UC Berkeley EECS Department, and the specific application is course and room scheduling. The database contains 24704 pages of data in 102 relations. The

data is information about courses taught: instructor's name, course name, room number, type of course, etc.

The first benchmark shall be denoted as "short1". The following are queries from the Short1 benchmark, which are exactly as the user wrote them.

```
destroy temp
retrieve into temp
  (courseNN.info1, courseNN.info2,....,courseNN.info13)
                where courseNN.instructor = "name"
print temp
```

where courseNN was any one of 8 course relations and "name" was any one of 76 instructor names.

The courseNN relations were hashed on instructor name. The data is stored in one relation per quarter. Since the same relations were used for room scheduling, there is an entry for each course for each day the course is taught.

First, any relation named "temp" that happens to be in the database is destroyed. Then the data needed from the courseNN relation is put into temp, where the implicit actions of removing duplicates and sorting on the first field take place. Then the data is printed. The only reason for using this "destroy - retrieve into - print" technique, according to the user, was to remove the duplicates introduced by having an entry per day per course. That technique is probably

not generally necessary for overhead-intensive queries. Therefore, the benchmark "short2" was created. It is the query stream short1 except the destroy - retrieve into - print set is replaced by

```
retrieve (courseNN.info1,
        courseNN.info2,....,courseNN.info13)
                where courseNN.instructor = "name"
```

This prints directly to the terminal without removing duplicates.

It was decided that the general user also probably will want fewer than 13 items of information per query, so short3 was created. It is:

```
retrieve (courseNN.info1, courseNN.info2)
                where courseNN.instructor = "name"
```

The three benchmarks were run and the following information obtained.

For all three query streams, we show the following:

1) The independent, random reference model holds for references to data pages.

2) There is a high degree of locality of reference to system and temporary relations.

3) The functions best distributed to other processors are those associated with the terminal monitor.

Both (2) and (3) may be highly INGRES specific; (1) appears to generalize

4.1.1. I/O reference patterns

Table 4.1
Overhead-intensive queries: I/O reference patterns
(all times are in seconds)

query stream	number queries	I/O time per query	number system ref	number data ref	% ref seq
short1	228	3.06 (*)	170 (*)	5 (*)	13.3
short2	76	.55	13	3	18.8
short3	76	.25	8	3	21.8

(*) : quantities given for each set of three queries (destroy,retrieve into, print)

Table 4.1 summarizes some of the trace analysis results for the overhead-intensive query streams. The first column is the query stream name, the second the number of queries in the query stream. The third column, the I/O time per query, was obtained by multiplying the number of page references by the average physical I/O time per block for the query stream, and dividing by the number of queries. The average physical I/O time per block for each query stream varies slightly from one query stream to another, depending on the placement of data on the disk. The trace analysis program reports the total number of references to each of the relations. These are separated into system and data references,

divided by the number of queries, and presented in columns four and five. The trace analysis also keeps track of logically sequential references. A page reference is a logically sequential reference if the logical page number is one plus the logical page number of the previous reference to that file. The percentage of the references that were logically sequential are reported in column six.

We note from Table 4.1 that the number of queries in short1 is three times that of short2 and short3, a direct result of the way the query streams were formed.

For short1, the average I/O time for the query set was 3.06 seconds. Most of this time (98%) was spent reading and writing INGRES system relations. The relation that contains information about the relations in the database (the relation relation) is referenced to destroy temp and create it again, and to retrieve information about courseNN. The relation that contains information about each attribute in the database, the attribute relation, is referenced once per attribute in the relation to be destroyed or created, and once per attribute of courseNN in the query. It should be noted that a cache of the system catalog information could be used to substantially decrease the number of system catalog references. This is not currently done.

The data referenced per query set in short1 includes three data pages from courseNN read, and one page written to and

read from the relation temp. The references to sort the data are not included.

Comparing the Table 4.1 entries for short1 and short2 we conclude that the user is paying a lot for duplicate suppression. The I/O time for short2 is significantly less than the time for short1 because the system relations do not have to be referenced as much. The data references are now the three pages read from the courseNN relation. There remain an average of 13 references per query to the system relations because the attribute and relation relations must still be read for verification. The cost of that verification is apparent in short3, where the only difference between that query and short2 is that it references fewer attributes.

4.1.1.1. Sequentiality of reference

The high percentages of sequential references we see in Table 4.1 are the result of reading strings of overflow pages in the system relations. The user's database was copied from the user's machine to the test machine, so the overflow pages in the relations were formed when the relations were first created. In that case, strings of overflow pages tend to be sequential. That would not be the case if the data had been added a little at a time, through updates. It would be the case anytime the queries were run on newly

modified system relations. Therefore, the sequentiality observed in Table 4.1 cannot be assumed to be true for overhead-intensive queries in general, and cannot even be assumed to be generally true for overhead-intensive queries in INGRES.

4.1.1.2. Locality of reference

A commonly accepted measure of locality is the hit-ratio curve [RODR76]. The hit ratio curves for the overhead-intensive queries are presented in Figures 4.1 and 4.2.

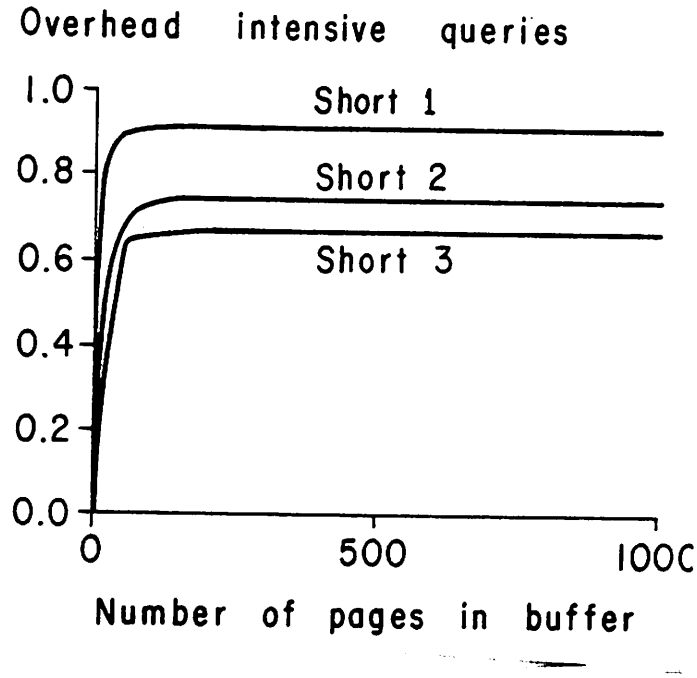


Figure 4.1 Overhead-intensive Queries

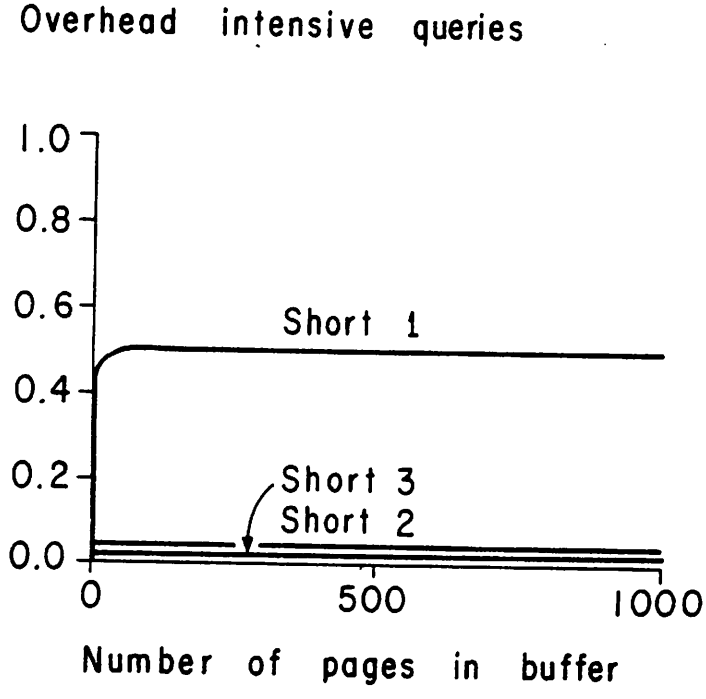


Figure 4.2 Overhead-intensive Queries, System References Removed

The vertical axes are the percentage of requests that would have been buffer hits if the buffer were the size given on the horizontal axes. These curves were calculated by taking the output from the software trace, the logical reads and writes, and simulating the effect of increasing the buffer size. The LRU algorithm for buffer replacement was used.

In this query stream, because each query references so little data, we are only interested in inter-query locality.

Figure 4.1 is the hit ratio for the overhead-intensive query streams. There is a high hit ratio for even a small number of buffers for all three query streams because of the large number of reads and writes to system relations. This is confirmed by Figure 4.2, which shows the same curves, but with the references to the system relations removed. The line for short1 in Figure 4.2 is higher than those for short2 and short3 because short1 is writing and reading small temporary relations.

Since the hit ratios for short2 and short3 are nearly zero, it is apparent that there is no inter-query locality in the data references. The data references for the overhead-intensive queries therefore conform to the random reference models of data references. When the system relations are included, the hit ratios become high. The straight lines of the hit ratio curves indicate that there is no advantage to adding buffers after the few needed for the system refer-

ences are provided.

4.1.1.3. Conclusion

Whether there is locality of data reference in general in overhead-intensive queries depends on the application. In applications such as customer information systems and banking applications, there is little locality of data reference. However, there are systems such as airline reservation systems that may naturally have much locality of data reference (the plane about to leave).

Therefore, for overhead-intensive queries, the only reliable locality of reference is the references to the system relations. There is a great deal of locality of reference there. It is clear that caching the system relations would be very beneficial.

INGRES has a heavier use of system relations than most other data management systems for two reasons. First, the process structure forces greater referencing of system relations because of validity checking in each process. Second, INGRES is interpretive. Many other data management systems are compiled, and do only minimal run-time validity checking of system catalogs. However, as long as successive queries are to the same database, and there is run-time validity checking, caching system catalogs should produce performance improvements for any data management system.

4.1.2. CPU usage patterns

Table 4.2 contains the CPU usage patterns for overhead-intensive queries. The CPU time for each process is given in Table 4.2. Also included is the amount of the time in OVQP spent to fetch and manipulate the data (dp). The OVQP total includes the dp time.

Table 4.2. Overhead-intensive queries: CPU usage patterns
all times are in seconds

query stream	monitor	parser	decomp	OVQP total	OVQP dp	DBU	total
short1	1.91	.30	.21	.52	(.16)	4.48	7.42
short2	1.75	.98	.11	.38	(.15)		3.22
short3	.88	.26	.05	.28	(.12)		1.47

(dp is included in OVQP total)

The time spent in the terminal monitor is a function of the number of characters in the query and is mostly spent looking for macros. The time spent in the parser process for short1 is less than the time spent for short2 because some functions to print the output are done by the parser process for a "retrieve" (in short2) and are done by the utility print for a "print" (in short1). It is greater for short2 than short3 because short3 has fewer domains for verification and because the query in short3 is smaller, thus easier to parse.

Decomposition is the process that breaks the query apart into single relation queries. It must also be called to

pass data through the pipes from one process to the next in line. Since the utilities are at the end of the line, it must be called several times per query set in short1. Therefore, the time in the process decomp is longer in short1. The time difference between short2 and short3 for decomp is accounted for by the difference in length of the message passed.

The time given for OVQP total includes the data processing time in the column in the table marked dp. The time is greater for short1 because OVQP must open two relations (temp and employee) and write the data to one of them. The time difference between short2 and short3 is because there are fewer domains. The data processing time, dp, is the time to process the three pages.

The time in "DBU" is the time to destroy, create, sort, and print the relation temp.

4.1.3. Conclusion

Except for short1, which is probably not a general query stream, the terminal monitor requires the largest percentage of time in the overhead - 54% for short2. The actual data processing time in all cases is much less than the time for setting up the query. The total CPU time per query in Table 4.2 is, for all three cases, greater than the total I/O time per query in Table 4.1. We therefore conclude that INGRES

is always CPU bound when handling overhead-intensive queries.

The analysis of the overhead intensive query streams reveals that the use of multiple processors at the terminal level will have a much greater effect on the performance than the use of multiple processors at the dta level. Distributed processing at the data level (as in DIRECT [DEWI78] and RAP [OZKA77]) will not speed the processing of overhead-intensive queries at all, since they spend little time processing data. In fact, an extra staging device between the I/O device and the user, as in DIRECT, or the inability to support access to a single item through a key , as in RAP, will slow the processing of short queries. Instead, either the processing must be distributed toward the user, through use of intelligent terminals and front-end machines, or the amount of processing reduced through use of a less functional terminal monitor. In the overhead-intensive queries in the benchmark query streams, the movement of the terminal monitor functions would clearly be a performance improvement.

The INGRES terminal monitor provides many functions for the user (eg. macro definitions, abbreviations) and is certainly not a minimal terminal monitor. However, if these functions are to be provided to the user, it is clear they can best be provided through intelligent terminals.

The use of extended storage devices to cache system relations would clearly be a performance benefit. However, any system that pre-compiled queries would not need the systems relations at run time. In that case, extended storage devices would not be useful at all for the overhead-intensive query streams.

4.2. Data-intensive queries

The data-intensive query streams were developed from an accounting application, the UC Berkeley EECS Department's Cost Account and Recharge System. Again, as in the overhead-intensive query case, we use the technique of making the first benchmark correspond exactly to the user queries, the third benchmark correspond exactly to our idea of what a "typical query stream" for data-intensive queries would be, and the second benchmark is in between.

We shall show in this section that:

1) The INGRES implementation of aggregate functions (explained below) leads to locality of reference to temporary relations.

2) There is a high degree of sequentiality of reference in data-intensive queries.

3) It will be advantageous to distribute the CPU functions associated with processing the data.

(1) is INGRES specific; (2) and (3) appear to generalize to

other data management systems.

The query stream "long1" is exactly as the user wrote it. It consists of 58 queries which reference 14 relations which contain a total of 822 pages of data. This query stream prints accounting reports by creating temporary relations in which the domains are both projections of existing relations in the database and zero or blank summary domains. The summary domains are then filled in by using multi-relation aggregate functions, and then the temporary relations are printed. There are an average of 14 domains referenced per query in long1. Long2 is long1 with all multi-relation aggregate functions removed, and the "retrieve into" constructs replaced by "retrieve". Long2 contains single-relation aggregate functions. Long3 is long2 with all aggregate functions removed, and with the average number of domains referenced by each query reduced to two. There was no attempt to do the same work in long1, long2 and long3.

Aggregate functions

The query:

```
retrieve (outstand.acct,
                outstand.fund,
                encumb = sum (outstand.encumb by
                        outstand.acct,
                        outstand.fund))
```

is a query from the accounting application and included in long1 and long2. The relation "outstand" has information about the department's outstanding accounts. The query

results in a list of the totals of the outstanding encumbrances grouped by account number and fund. INGRES processes the aggregate function "sum" by creating a temporary relation "temp1" which contains three domains: account, fund, and sum. It will be hashed on (account, fund) and is initially empty. The relation "outstand" is read once, and for each tuple the (account, fund) pair evaluated, the tuple from "temp1" for that pair read, the sum updated, and the totals replaced in the "temp1" relation. After the last tuple of "outstand" is read, the "temp1" relation is read and the results printed on the user's terminal.

4.2.1. I/O reference patterns

Table 4.3
Data-intensive queries: I/O reference patterns
(all times are in seconds)

query stream	number queries	I/O time per query	number system ref	number data ref	% ref seq
long1	58	15.0	129	290	30
long2	18	16.7	73	484	28
long3	13	5.1	15	155	84

Table 4.3 presents the results of the query analysis of the data-intensive query streams with respect to I/O usage. The number of queries in long1 is greater than the number of

queries in long2 because the multi-relation queries in long1 were not included in long2, and because the "destroy - retrieve into - print" queries were replaced by a single "retrieve". Long3 contains fewer queries than long2 because the aggregates were dropped to create long3. The I/O time per query is the total I/O time for the query stream divided by the total number of queries in the stream. It is greater for long2 than for long1 because the queries that were dropped from long1 in forming long2 were queries that reference little data. The queries that form the long3 subset of long2 reference much less data because the aggregates were dropped from long2 to create long3.

The number of system references per query is a direct result of the number of temporary relations created and the number of attributes referenced per query.

4.2.1.1. Sequentiality of reference

The percentage of sequential accesses is high in all three cases. It is apparent that the INGRES processing of aggregates is dominating the I/O references, because when the aggregates are removed, the sequentiality dramatically increases. This sequentiality is the result of reading entire relations, either to print selected attributes, or to print summary statistics.

4.2.1.2. Locality of reference

In Figures 4.3 and 4.4 note the different types of locality present. In long1 and long2 the hit ratio curve is the gently rising curve indicative of random re-referencing of a relatively small (compared to the buffer size) set of pages. Since the re-referencing is random, each page added to the buffer increases the hit ratio slightly. This pattern is the result of the INGRES implementation of aggregate functions, but the same pattern would result from accessing a relation through a secondary (non-clustering) index.

In long3 we see the result of sequentiality and locality. The same relation was referenced sequentially in several queries; when the buffer size was large enough to hold both that relation and the relations referenced by intervening queries, there was a sharp jump, at 350 pages, in the hit ratio curve.

In Figure 4.4 we see that this locality is not caused by references to system relations. This is not because system relations are referenced less in data-intensive queries than in overhead-intensive queries, but that the proportion of system references to data references has changed. Therefore, although caching system relations will not hurt the performance of the data-intensive query, it will not greatly improve it either.

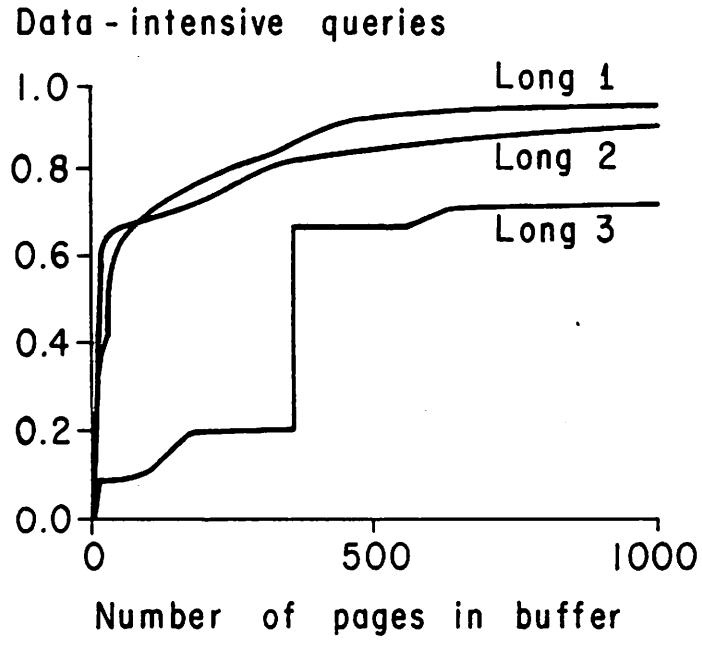


Figure 4.3

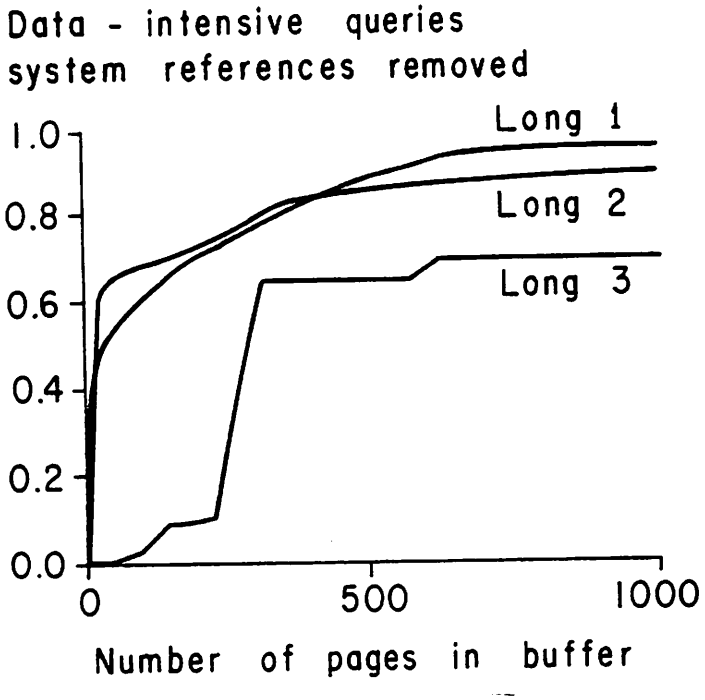


Figure 4.4

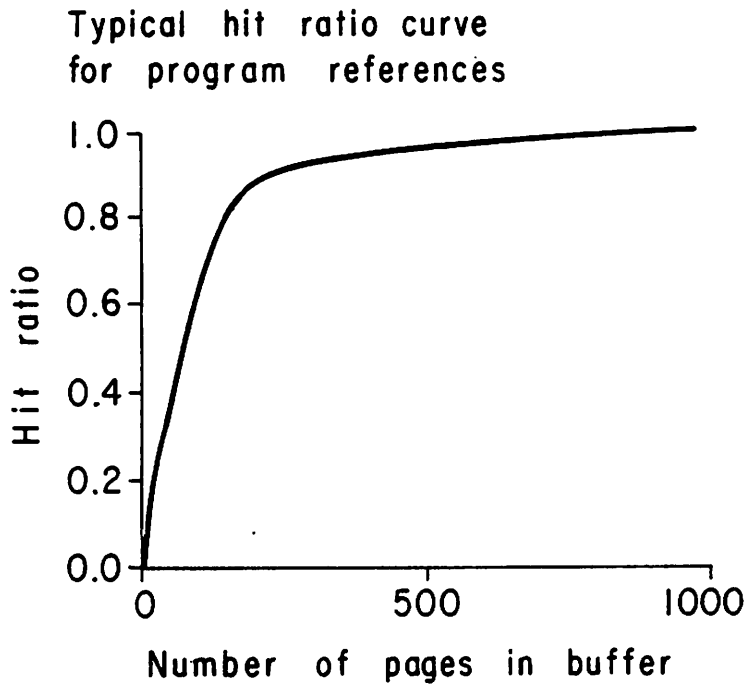

Typical hit ratio curve
for program references
1.0
0.8
0.6
0.4
0.2
0.0
Hit ratio
0
500
1000
Number of pages in buffer

Figure 4.5

_.2.1.3. Conclusion

There was a high degree of sequentiality found in all three reference traces. There were two types of locality found. One type, the locality resulting from the INGRES implementation of aggregate functions (as seen in the hit-ratio curves for long1 and long2) is very like the locality found in program references (figure 4.5). The buffer sizes needed to take advantage of this type of locality can be relatively small. The second type of locality, the cyclic-sequential (as seen in the curves for long3) would require arbitrarily large buffer sizes. The buffers would have to contain the entire relation being re-referenced, as well as all relations referenced between the first and second references to the same relation. It may be best to simply ignore the cyclic-sequential case, and only do the read-ahead that the large amount of sequentiality mandates.

4.2.2. CPU usage

Table 4.4. Data-intensive queries: CPU usage patterns
all times are in seconds

query stream	monitor	parser	decomp	OVQP total	OVQP dp	DBU	total
long1	1.7	.4	.8	7.1	(6.9)	5.8	15.8
long2	2.0	.9	.26	11.23	(11.0)		14.4
long3	1.3	.3	.07	3.73	(3.5)		5.4

(dp is included in OVQP total)

The variation between monitor and parser times is as explained in the overhead-intensive queries. The decomposition time for long1 is much greater than the others because of the presence of multi-relation aggregate functions in long1, which means decomposition has work to do. The OVQP time is greater per query in long2 because long1 includes queries where most of the work is being done in the utilities. When those queries were dropped, the average per-query time in OVQP increased. The time spent in the utilities (DBU) is mostly spent sorting relations and printing them.

Long3 has less time in OVQP because fewer attributes were manipulated in the queries in long3.

4.2.3. Conclusion

We note that in all cases the data processing time (dp) in OVQP is the greatest single item of CPU time. In no case is the I/O time markedly greater than the CPU time. INGRES interprets queries, and processes the data by interpreting on a per tuple, per domain within the tuple basis. The data processing time for INGRES is therefore probably higher than the time in other, compiled systems. However, in any system that supports data-intensive queries the majority of the CPU time should be spent processing the data. In such systems, distributing the processing toward the data, as is proposed in the design of several data management machines, may increase performance. This is further explained in Chapter 5.

The use of intelligent terminal montors would minimally affect the performance of the data-intensive query. The use of extended storage devices, to buffer the data, would be very advantageous since the data is read sequentially.

4.3. Multi-relation queries

Multi-relation queries are specific to relational systems. The INGRES implementation of them does not necessarily generalize to other relational systems. They were included in this analysis because the potential benefits of extended storage devices and distributed processing may make a very efficient implementation of multi-relation queries possible.

In this section we shall show that:

1) There is extensive locality of reference to temporary relations

2) The CPU time to process data is most of the total CPU time to process the query.

In INGRES multi-relation queries are processed through the formation of temporary relations and the use of tuple substitution. The technique is described in detail in [WONG76] and [YOUS78] and shall be illustrated by an example. We shall call this example the "rooms" query. It is included in the benchmark, and is from the user application. The query is:

```
retrieve ( rooms.building, rooms.roomnum, rooms.capacity,
                course.day, course.hour)
        where rooms.roomnum = course.roomnum
                and rooms.building = course.building
                and rooms.type = "lab"
```

The relation "course" contains information about all the courses taught by the UC Berkeley EECS Department in the last four years. It contains 11436 tuples in 2858 pages, and is stored in an ISAM storage structure, keyed on instructor name and course number. The relation "rooms" contains information about every room that the EECS Department can use for teaching courses. It contains 282 tuples in 29 pages, and is hashed on room number.

The result of this query is a list which contains the building, room number, capacity, day, and hour of the use of any

lab for the last four years.

To process this query, first INGRES will note that there is a one-relation restriction ("where rooms.type = "lab"), so that restriction will be done first. The query is issued

```
retrieve into temp1 (rooms.building, rooms.roomnum,
                                    rooms.capacity)
        where rooms.type = "lab"
```

The temporary relation "temp1" which resulted from the actual query in this case contained 20 tuples in 2 pages.

The relation "course" is not stored in a way that is helpful to the processing of this query, and only a few domains of each tuple are needed for this query. So INGRES performs the projection of "course" by issuing the query:

```
retrieve into temp2 ( course.day, course.hour,
                course.building, course.roomnum)
```

This results in a relation "temp2" which contains the same number of tuples as "course" (11436) but less space (867 pages) since the tuples are smaller.

The final step is tuple substitution, where each tuple in "temp1" is compared to each tuple in "temp2", and the result printed on the terminal. For instance, the first tuple in temp1 is the tuple (cory, 119, 15), so the query is issued:

```
retrieve (building = "cory", roomnum = "119",
                capacity = 15,
                temp2.day, temp2.hour)
        where temp2.roomnum = "119"
                and temp2.building = "cory"
```

This process of tuple substitution is repeated 20 times, once per tuple in temp1. Since temp2 is unordered, the result is that the entire temp2 relation is scanned 20 times, resulting in 17,340 data pages read. INGRES includes a set of heuristics which dynamically decide when to reformat a temporary relation. Temp2 could have been reformatted to a relation hashed on building, room number. It was not reformatted because the cost functions associated with modifying the relation to hash showed that cost would be greater than re-scanning the relation 20 times.

The multi-relation benchmark was prepared by assembling a collection of unrelated users' queries which were multi-relation queries and which referenced the same database. The database was the UC Berkeley EECS Department's course and scheduling database. The patterns observed were dominated by the "rooms" query. Most of the queries referenced about 109 pages; the "rooms" query referenced 19000 pages of data. Most of the queries used about 7.25 seconds of CPU time; the rooms query used 709.5 seconds of CPU time. Therefore the results are reported without the query "rooms" in multi1, and for "rooms" alone.

4.3.1. I/O reference patterns

Table 4.5
Multi-relation queries: I/O reference patterns
(all times are in seconds)

query stream	number queries	I/O time per query	number system ref	number data ref	% ref seq
multi1	24	3.27	23	86	35
rooms	1	505.16	70	19023	85

There are few system references compared to the number of data references in the "rooms" benchmark because the temporary relation is being read so many times. The temporary relation is read sequentially each of the 20 times it is read, which is why the percentage of sequential references is so high. Figures 4.6 and 4.7 show the hit-ratio curves for the multi-relation queries. We note that there is a high degree of locality in the multi-relation queries, and that that locality is not from referencing system relations. The hit-ratio curve for the rooms query takes a sharp jump as soon as the window size is above the 867-page size of the relation being continually re-referenced. This is the cyclic-sequential referencing found in the query stream long3, but with a difference: the size of the cycle is precisely known by INGRES, and could be communicated to the operating system. The operating system could then arrange to retain the entire relation in extended storage.

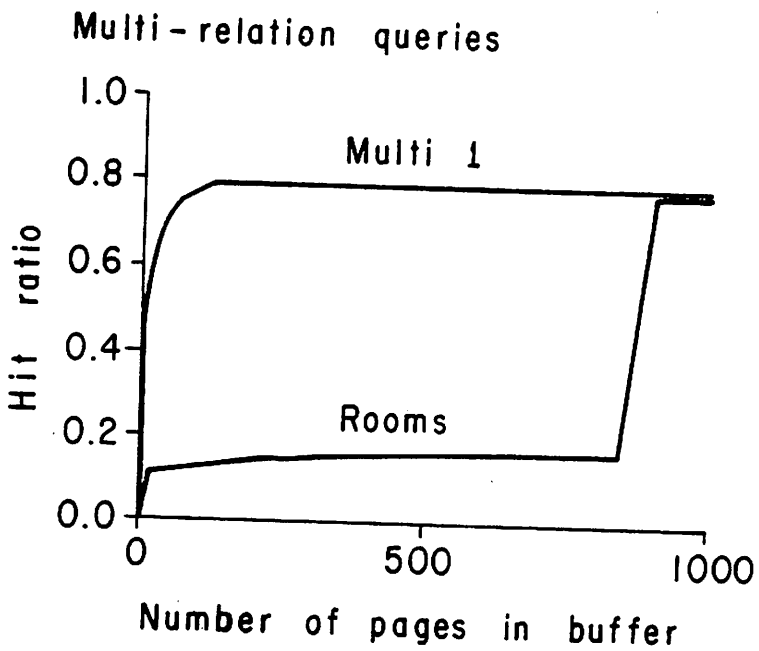
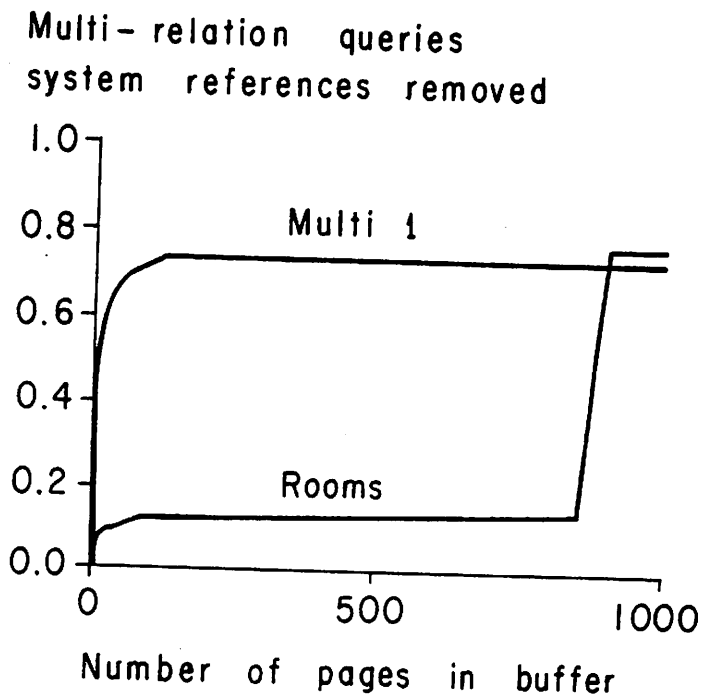


Figure 4.6

Figure 4.7



4.3.2. CPU usage

Table 4.6. Multi-relation queries: CPU usage patterns
all times are in seconds

query stream	monitor	parser	decomp	OVQP total	OVQP dp	DBU	total
multi1	1.0	.25	.52	4.33	(4.2)	1.15	7.25
rooms	1.37	.81	.71	705.75	(705.4)	1.86	709.5

(dp is included in OVQP total)

Since these queries are data-intensive as well as multi-relation queries, the cpu time spent in the data-processing portion of the data management system is the greatest component of the cpu time.

4.3.3. Conclusion

The increased data-independence and functionality provided by relational data management systems over other systems is provided by the capability of handling multi-relation queries. However, the algorithms for multi-relation queries are sometimes claimed, in part, to be responsible for relational data management systems being less efficient than other systems.

There are many algorithms for handling multi-variable queries other than those INGRES uses. In [BLAS76] several other algorithms are described. However, it appears to be very difficult to devise algorithms that never resort to re-referencing the same data. The algorithms in [BLAS76]

which do not rely on methods involving the re-referencing of the same data pages rely instead on sorting the data or indices. Sorting involves a limited kind of re-referencing of the data, at least to write it, then read it in again. There is in INGRES and may be in most relational systems extensive locality of reference in multi-relation queries. The window size for that locality may be very large - the size of an entire relation, for sequential referencing, - or it may be smaller, for random re-referencing, but it is there, and can be used to increase efficiency. The large cache sizes necessary will become possible through the use of extended storage devices.

The CPU time per query in Table 4.6 is greater than the I/O time per query (Table 4.5) so INGRES at this time would see little benefit from caching the relations to be re-referenced. However, combined with the compilation of the queries and/or distributing processing on the data level, using extended storage devices may make the increased functionality of relational data management systems possible without the present cost in efficiency.

4.4. Chapter Four Conclusion

The performance patterns of overhead-intensive queries have been shown to be completely different from the patterns of data-intensive queries. It is apparent that machines that distribute the processing toward the data, as database

machines do, are only effective in the case of data-intensive queries. It has also been shown that data-intensive queries do not benefit from the distribution of the processing toward the user, as in the use of intelligent monitors. The multi-relation queries were shown to be cyclic-sequential, and could benefit from the use of extended storage devices.

Extended storage devices were also shown to benefit data-intensive queries when used as read-ahead buffers. They also benefit overhead-intensive queries for those systems that interpret queries when used to cache system data.

Chapter Five

Analysis of

the Use of Data Management Machines

5. Chapter 5

5.1. Introduction

The rapid advances in the development of low-cost computer hardware have led to many proposals for the use of this hardware to improve the performance of data management systems. Usually the design proposals are quite vague about the performance of the system with respect to a given data management application. In this chapter we predict the performance of several of the proposed data base management machines with respect to INGRES benchmark query streams.

The term "data management machines" is used here to describe any special-purpose hardware built to enhance the performance of data management systems. The systems analyzed in this section include both those actually built, and those that remain designs on paper.

The systems that are described and analysed in this chapter are associative disks [LANG78, LIN76, SLOT75]; RAP [OZKA75, OZKA77, SCHU78]; CASSM [SU75]; DBC [BAN78A, BAN78B, BAUM76, HSI76A, HSI76B, KANN78]; DIRECT [DEW78A,DEW78B]; and CAFS [BABB79, COUL72].

The underlying focus of each of the above machines is that they associatively retrieve data. A data element is requested by content, not by position, from secondary storage. In section two an overview of each machine is presented. Section three contains the prediction of the performance of the machines when executing benchmark queries, and section four is the conclusion of the chapter.

5.2. Data Management Machines

5.2.1. Overview of machine architectures

In this section each machine is briefly described and the use of the machine illustrated with an example. That example is the following query, Q1:

Query Q1:

```
retrieve (EMP.name, EMP.salary)
        where EMP.dept = 10
```

Each machine is also illustrated with a figure. For comparison, figure 5.1 shows a standard computer system (i.e., one that does not include a back-end machine). In that system, the data blocks which are read are serially processed by the disk controller and the channel.

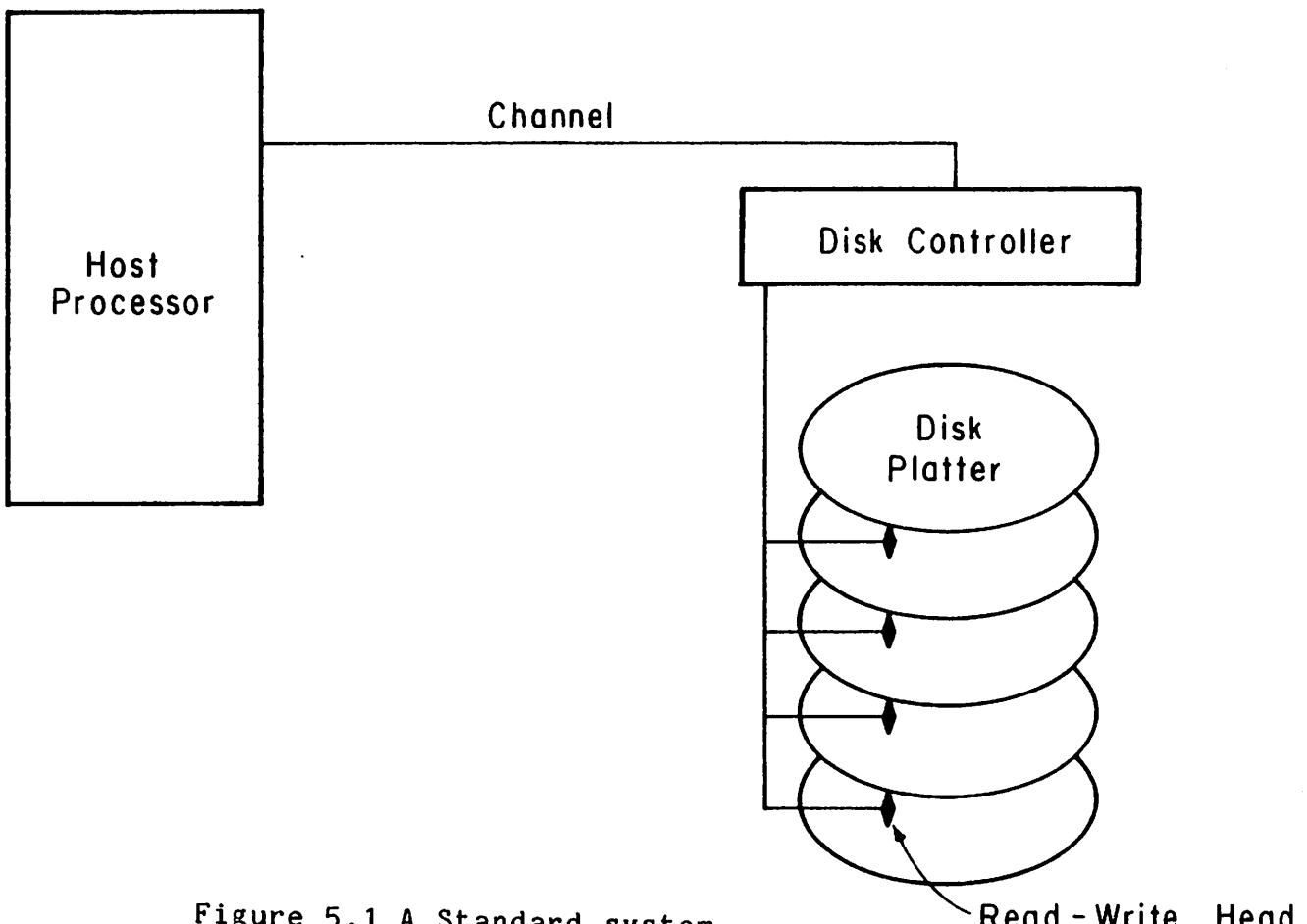


Figure 5.1 A Standard system

5.2.1.1. Associative disks

Most earlier designs for hardware to enhance the performance of data management systems were associative disk designs. First proposed by Slotnik [SLOT70], the design is to attach a processor to each of the heads of a head-per-track device (disk or drum). Figure 5.2 shows an associative disk system. The per-track processors are denoted as cell processors, and the processor that co-ordinates their activities is the controlling processor.

The cell processors can be loaded by the controlling processor with the value or values to search on, the search can take place in parallel, and the only data returned to the main computer are the records with the required values. As originally designed [SLOT70], the cell processors performed no arithmetic functions (e.g. sum, max, min, etc.). They were only search engines.

In RARES [LIN76] the design was extended by writing and reading the data in parallel, across several tracks (in "bands") instead of serially, as is usually done. The advantage is to increase the bandwidth to the controlling processors, and enable the decision to return the data element to the main computer to be made more quickly. The drawback to such a scheme is the complicated error detection and recovery. If the data is to be processed on a word-by-word basis instead of by blocks, then the parity, checksums

and error-correcting codes must be on the order of those used for core memory, rather than those used for disks. However, the error rate of disks is much higher than that of core memory. These problems are questions never addressed in [LIN76]. Therefore, this analysis is confined to the more standard associative disk designs.

In Slotnik's associative disk system, query Q1 above would be processed in the host machine, then a command sent to the associative disk to return all records where the dept field = 10. The host processor would then format the tuples for printing, taking only the salary and name field from each tuple.

If all the cell processors attempt to return a value to the controlling processor at the same time serious performance problems could occur. There may be bus contention problems on the data bus from the cell processors to the controlling processor. If the system does not bottleneck at the bus, the controlling processor may have problems keeping up with the datarate of all of the cell processors transferring at once.

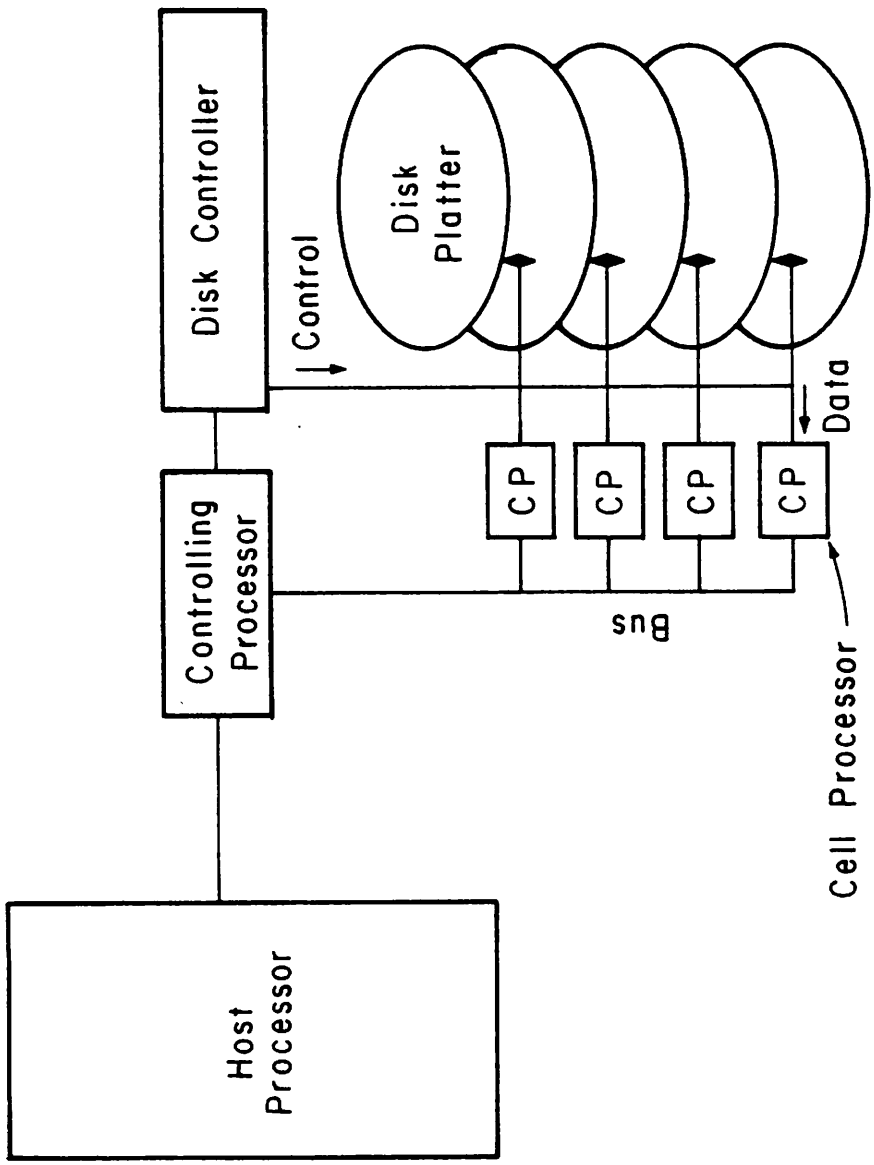


Figure 5.2 Assocative Disks, DBC, CASSM

5.2.1.2. CASSM

CASSM (Context Addressed Segment Sequential Memory) is the data management machine developed at the University of Florida. It was developed after associative disks, but before RAP, and represents a middle ground between them. CASSM is essentially a processor-per-head device, like an associative disk, but the processors have added capability in that they can perform a few arithmetic functions (integer sum, min, max).

The data is laid out in segments, where each segment is operated on by a single processor. The segments are not independent modules, however; data can freely migrate across segment boundaries. The data and processors operate under the direction of a separate controlling processor. On the fixed head disk, each segment is one disk track.

In the following analysis, it is assumed that CASSM is implmented on a moving-head disk. In that case its architecture closely resembles that of the associative disk, and is represented in figure 5.2 The differences in the systems lie in the power and function of the cell and controlling processors.

In CASSM, the processors all execute the same function at the same time, where the functions are data-processing directives (search for, delete, add, etc.) The processors have a minimum of buffer space, and will only test for a

single qualification on one attribute at a time. The system includes special-purpose functions for string searches, updates, inserts, and garbage collection. The processors also have the capability of following pointers within the records written on the disk, so the hierarchical and network data models can be supported as easily as the relational model. The capability exists to write instructions on the disk itself, to be read later by the processors, and change their actions. This capability is used, for instance, to change functions after one pass over the data.

In CASSM data is stored by attribute rather than by tuple or record. Each attribute is flagged with its name, and attributes are grouped together in records, each record defined by delimitor fields and a record number. Groups of mark-bits are associated with both individual attributes and records. One of the mark-bits is a collection bit, which if set signifies that the CASSM processors should send the data to the host processor.

A data encoding algorithm is implemented in the hardware: each character-string value is stored only once, in a table of values. This table is stored as one of the segments. In the actual record, a pointer to the value is kept.

CASSM is limited in that only one attribute at a time may be tested or output. The testing of one attribute may be done at the same time as the output of another.

The following explanation of the processing of a query in CASSM is based on a narrative of the execution of a similar query in [SU75]. To process Q1, CASSM will, in the first cycle, mark all records that have the attribute-value pair (relation, EMP). Then, in the next cycle, the attributes within the marked records are inspected and marked for collection if the record contains the attribute-value pair (dept,10). The marked attributes "salary" are then returned to the host on the next cycle, while CASSM follows the name pointers in the marked attributes "name". The final cycle returns the name fields. The host machine must assemble the tuples to be printed by matching record numbers.

In the case that many values at once must be sent to the controlling processor, the same performance problems occur as in the associative disk. CASSM deals with these problems by having the cell processors request the bus when they have a data item to transmit; if it is unavailable, the processor waits until it can transmit the data item it has before reading the next block. Therefore bus contention problems may result in the query execution requiring several extra disk revolutions.

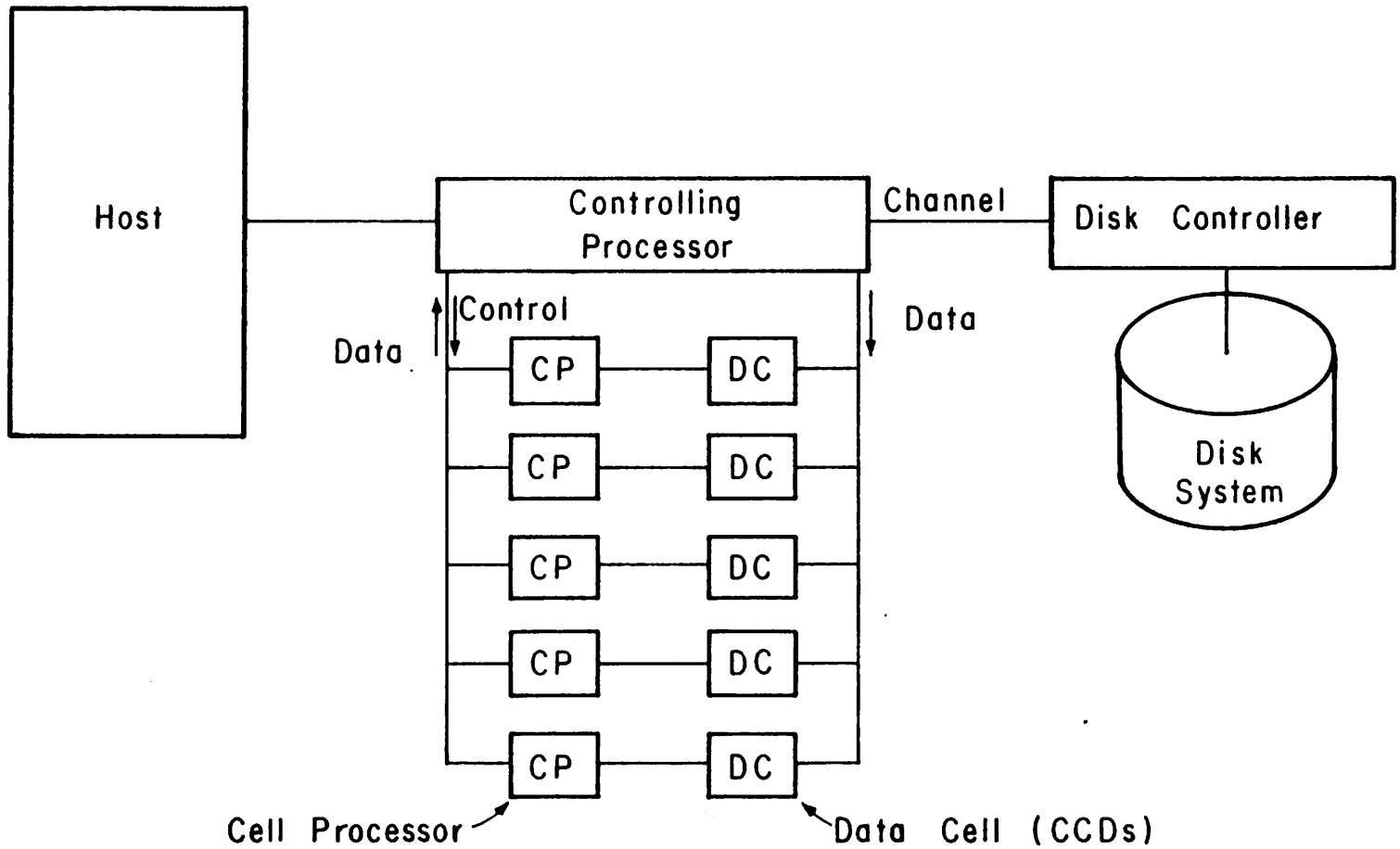


Figure 5.3 RAP

5.2.1.3. RAP

The relational associative processor (RAP) is similar to CASSM in that it was, at first, implemented by attaching multiple processors, one per head, to a fixed-head disk. However, RAP is very different from CASSM in that RAP supports only the relational data model.

There are three functional parts to the RAP design: the controller, the statistical unit, and the cells. On a fixed-head disk each cell is a disk track. There is one processor per cell. The controller communicates with the front-end computer and directs the actions of the cell processors and the statistical unit.

Figure 5.3 shows the RAP system. The "controlling processor" box includes the statistical unit, and directs the functions of the cell processors. RAP is more powerful in function than CASSM because of the existence of the statistical unit, which performs more functions than the CASSM processors, and because each cell processor contains several comparator units, so several attributes at once can be tested. RAP organizes data in relations which are stored in the data cells.

RAP, unlike CASSM and the associative disks, is a currently active research project. Subsequent implementations of RAP have used CCDs [OZKA77] and may eventually use random access memory [SCHU78] in place of the fixed-head disk. The

version of RAP used in this discussion is that reported in [SCHU78]. It allows an individual cell to contain parts of several relations.

Q1 would be processed by RAP by passing the query to the RAP controller, which would determine which cells contain the "emp" relation, and direct the appropriate processors to return all tuples for which the dept = 10. The controlling processor would then pass these tuples to the host machine. In the case of bus contention the cell processors hold the data until the bus is free, thus potentially losing revolutions of the cells.

5.2.1.4. DBC

The data management machine developed at Ohio State is called DBC (Data Base Computer). Like CASSM, it is an attribute based system. It is a backend machine with seven major components, which are:

1) the data base command and control processor (DBCCP), which fields queries, communicates with the host machine and controls the functioning of the other components.
2) the keyword transformation unit (KXU), which makes an encoded version of the keywords to send to the next unit.
3) the structure memory (SM), which takes the encoded keyword and looks it up in a directory to determine the positions (indices) of the matching attributes in secondary

storage.

4) the structure memory information processor (SMIP), which performs set operations on the information from the structure memory.

5) the index translation unit (IXU) which decodes the information about the location of the attributes required in order to produce a physical address on the secondary memory.

6) the mass memory (MM), which is composed of moving head disks. Each disk has a processor per track (the Track Information Processors) that will perform basic functions.

7) the security filter processor (SFP), which contains a capability list of who has permission to access what data.

In the block diagram figure 5.2, all of the above components except the MM are grouped together in the "controlling processor" box.

The following explanation of the processing of query Q1 is derived from [BAN78A] and [BAN78B].

The DBCCP receives the command

Retrieve (name, salary)((relation = 'emp') & (dept = 10))

from the host processor.

The predicate ((relation = 'emp') & (dept = 10)) is analyzed by the keyword transformation unit, which determines if dept or relation are clustering keywords. In the case that they both are, the KXU transforms them into a coded, internal

representation for look-up within the Structure Memory. The SM produces a series of index terms for relation = emp and for dept = 10. These index terms are a coded representation of the cylinder numbers for the data and the security atoms associated with that data. The Structure Memory Information Processor performs a logical AND on the coded cylinder numbers so that only the coded cylinders and security atoms for the data satisfying the predicate are passed to the next unit, the Index Translation Unit. It transforms the internal, coded representations into actual disk cylinder numbers, and passes the information to the DBCCP. The DBCCP then directs the Mass Memory to perform the given task on the proper cylinders (find those records with dept = 10 and relation = emp, and output the name and salary fields). The data goes from the MM to the Security Filter Processor, which checks the security atoms to determine if the access is proper; if so, the results are sent to the DBCCP, which sends them on to the host computer.

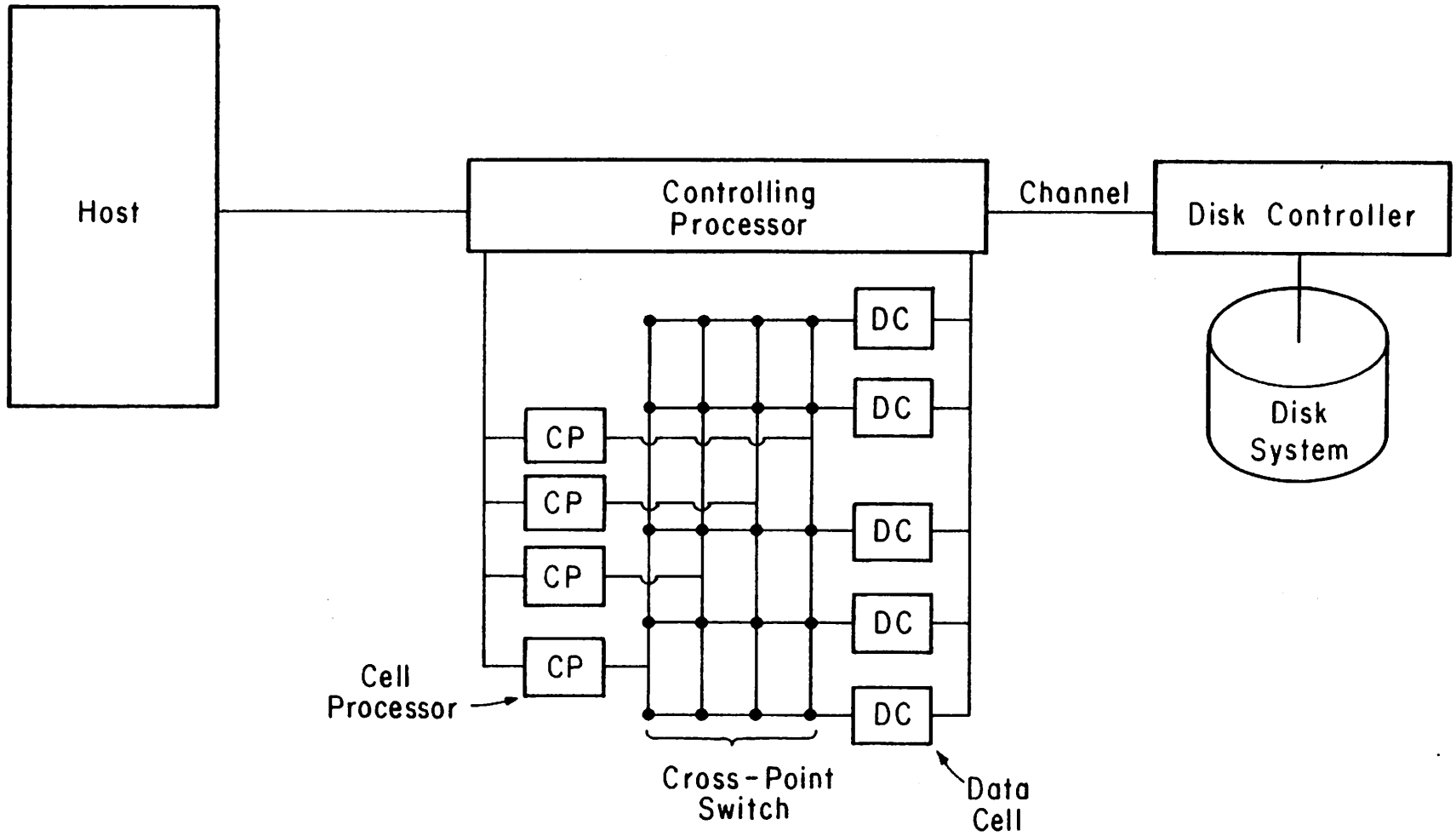


Figure 5.4 Direct

5.2.1.5. DIRECT

DIRECT is the backend data management machine under development at the University of Wisconsin-Madison (refer to figure 5.4). It is composed of a controller, an arbitrary number of query processors and CCD cache memory modules, and secondary storage. In the initial implementation there are to be eight LSI-11/03 microprocessors that will function as query processors. Thirty-two 16k byte CCD page frames will be the memory modules, and a PDP 11/40 will function as the controller. The page frames are connected through a crosspoint switch to the processors, so that any processor can act on data on any page; the controller receives the query from the host computer and directs the processors to act on the proper pages. If the pages are not in the cache, the controller directs that they be read in, and allocates the pages for them. Each processor can execute its instructions independently, so that it is possible that each processor can be executing a different query. The controller allocates as many processors as necessary to execute a given query.

To execute Q1, DIRECT would receive a representation of the query from the host computer; if there were no other queries running, DIRECT would order that all the processors work on the query. The processors each request the "next_page" of the EMP relation from the controlling processor; it coordinates which processor is assigned to which page, and

notifies each when the page it needs is present. As each processor scans the page, it writes qualified tuples to a new, temporary relation which resides on another cell. When the query processor finishes scanning a page of the source relation it requests the next page from the controlling processor. Upon filling a page of a temporary relation or receiving a "no next page" notice it transfers the temporary relation page to the controlling processor. Processing proceeds in that fashion until the end of the relation.

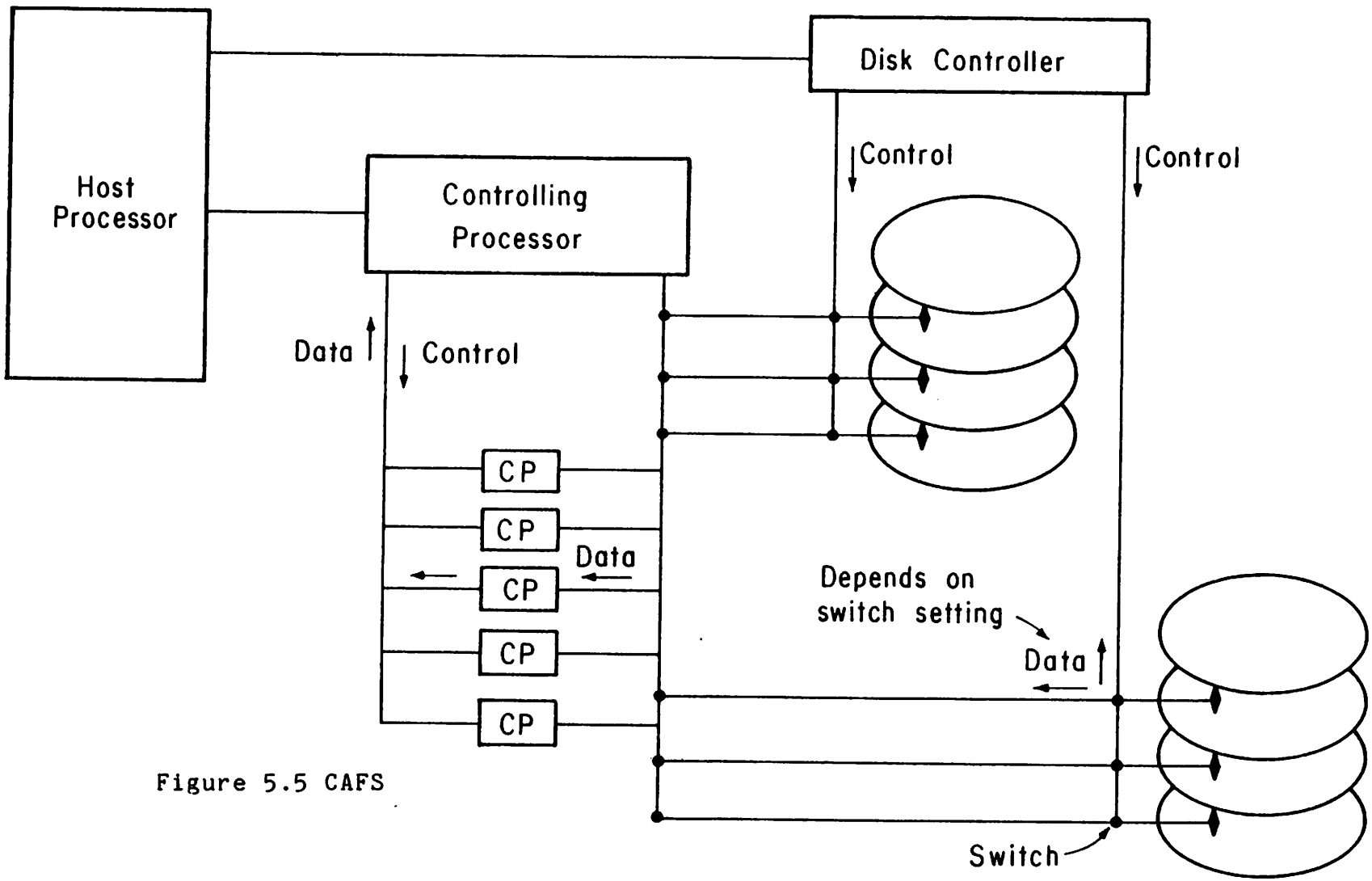


Figure 5.5 CAFS

5.2.1.6. CAFS

The Content Addressable File Store (CAFS) consists of a filter box between the disks and the host computer. This box can simultaneously operate on twelve different data streams to allow only qualified tuples to pass through to the host. The system reported in [BABB79] involves using disks whose track heads can be multiplexed to the CAFS box; they can also transfer data directly to the host, bypassing the CAFS system. Figure 5.5 shows CAFS.

The first action by the host processor to run query Q1 with CAFS is to load the CAFS box with the search key (dept = 10) and to connect each of the disk tracks that contain the EMP relation to the box; then the qualified tuples will be passed to the host.

Collisions that occur in CAFS will result in the extra disk revolution time.

5.2.2. Comparison of the machines

Each of the machines described is a cellular system: data is stored in cells, with a (in some cases dynamically assigned) processor per cell. Then operations on the cells take place in parallel. There are minor implementational differences in the machines: e.g., the RAP CCD pages are larger than those of DIRECT [DEWI78, OZKA77] and have a slower transfer rate.

Since none of the machines is in volume production, the minor implementational differences can easily change and should be ignored in an analysis of the machine. The purpose of the following study is to evaluate the effect on the machine performance of the major design differences. In the following list of the machines the differences of interest are explained.

5.2.2.1. Associative disks

There have been several designs proposed for associative disks. For the purposes of this analysis it is assumed that the associative disk is a very simple search engine which returns to the host the entire qualified tuple. The cost of the processors for this system would be lower than in the more complicated systems (see Section 5.3.2.2 for a discussion of cell processor design).

5.2.2.2. CASSM

Only CASSM includes a data-encoding algorithm as a part of its fundamental design. Additionally, it uses a very different algorithm for multi-relational joins than the other machines do. The cell processors are simplified in CASSM by the limitation that only an attribute at a time can be tested or output; this is a limitation that no other machine has.

5.2.2.3. RAP

RAP.1 [OZKA75] is a head-per-track device that, except for the presence of mark-bits, is quite similar to the associative disk system described in Section 2.1.1. It is proposed in [OZKA77] that RAP.2, which was implemented using CCD pages as the data cells, be used as a cache for a large database system. In this analysis, the use of RAP in a caching system is studied. RAP uses mark-bits to keep information about the tuples. It shall be assumed that the mark-bits are kept in a RAM bit map. The implications of that use shall also be explored.

5.2.2.4. DBC

DBC is designed specifically for implementation on a moving head device. It contains hardware to use index selection to narrow the search space for a query to only a few cylinders: none of the other machines have this functionality.

It also contains hardware to implement security restrictions on the data. This functionality, although important, is not present in any of the systems under discussion and was not used in the INGRES benchmark query streams. Therefore it will not be a part of the following comparisons.

5.2.2.5. DIRECT

Direct is the only system where the design is dependent on a cache. The cache in the prototype is made up of CCD pages.

Direct is also the only system that implements a cross-point switch so that all of the processors can work on the same page, or all on different pages. It is also the only system that forms temporary relations. RAP and CASSM use mark bits; DBC, associative disks, and CAFS send temporary results to the host processor. It is the cross-point switch that gives DIRECT the capability of writing temporary relations because it allows a cell processor to read one cell and write the results to another. It is only through the facility of the cross-point switch that the cell processor can be attached to two cells at once.

5.2.2.6. CAFS

CAFS is radically different from the other machines in that it can dynamically connect the cell processors to a subset of the tracks of several moving-head disks. This increases the functionality of the machine because more data is available to it. The cell processors themselves are simple, on the order of the associative disk processors. So the machine, when operating with the same number of cell processors as the associative disk, and when both are implemented on a moving head disk, will have the same single-relation query performance as the associative disk machine. It uses a different algorithm for multi-relation joins than the other machines.

5.2.2.7. Summary

The following points are of major interest:

1) mark bits

CASSM and RAP use mark bits in their query processing algorithms; the effect of the use of these bits will be investigated.

2) the use of temporary relations

Dewitt states [DEWI78] that the use of temporary relations to hold intermediate values is better than the use of mark bits. The analysis of the implications of the use of temporary relations is therefore of interest.

3) the effect upon performance of the transfer of entire tuples to the host

The simpler database machines return the full tuple to the host; the effect of this design is explored.

4) joining algorithms implemented in the machines

Each of the machines uses a different join algorithm; these shall be investigated.

5.3. Performance

5.3.1. Introduction

In this section the performance of the data management machines is analyzed. It is assumed that each is connected

to a host system supporting a relational query language. Their performances are compared to a conventional system supporting INGRES. Additionally, the performances of the database machines are compared to INGRES running on a conventional system augmented with a large buffer cache.

In the following analysis the execution times of four queries (from the INGRES benchmark query streams) is predicted for each of the systems. This prediction is made in terms of both numbers of seconds, to obtain intuitive insight into the functioning of the machines, and algebraic parameter values, to easily compare performance under changed assumptions. When it is not possible to estimate exactly how long a function will take "best case" and "worst case" values are given. These are denoted by (best case -- worst case).

The organization of the remainder of this section is as follows. Section 5.3.2 contains the physical specifications of the machines. It also contains the specification of the conventional and augmented systems supporting INGRES. There is certain work that must be performed in the host that is independent of query type and independent of the type of backend system. That time is described in Section 5.3.3. Section 5.3.4 contains the query performance analyses, and Section 5.3.5 the conclusion.

5.3.2. Specification of the systems

5.3.2.1. Cell storage media

For the purposes of comparing the systems, it is assumed that all except RAP and DIRECT use moving head disks as the data cell storage media. Actually, associative disk machines and CASSM were designed for fixed-head disks, but the use of the same media for all the systems helps make the analysis of the design differences independent of the storage media.

Since RAP and DIRECT are very interesting as caching systems, they are assumed to have the faster, but sequential, storage media that CCDs provide.

5.3.2.1.1. Physical specifications for disks

The physical specifications for the disks are assumed to be those of the disk on the INGRES standard machine. The rotation time of the moving head disk is denoted by DROT. For the numerical analysis, this is taken to be .0167 seconds, the rotational speed of the disks on the INGRES system. The average access time, DAVAC, is .030 seconds on the same disk. The block size is the same as the sector size; it is denoted by BSIZE and is 512 bytes on the INGRES system. The data transfer rate of the disk, through the controller and mass bus, is DRATE = .0012 seconds per 512 byte block. The

time for a cell processor to read a block is not the same as DRATE because the block goes directly to the cell processor, instead of through the bus and the operating system protocols into the computer's main memory. This time, DREAD, on the INGRES disk would be .0167 sec. per rotation / 22 blocks per track = .0008 seconds per block. DRATE is not equal to DREAD in most computer systems. DRATE is the time required to completely transfer a page into memory; it reflects the time to interrupt the operations of the computer and use the services to the operating system to store the data; it is a function of the speed of the controller, the DMA interface to the host, and the amount of memory interference at the host. DREAD, on the other hand, is merely dependent on the disk and disk controller speeds. The fact that DRATE is usually slower than DREAD is reflected in the widespread practice of "half-tracking" disks, where sequential disk blocks are stored one block away from each other, in effect slowing down the data rate of the disk.

The number of 512-byte blocks per cylinder is DCYL. This is, on the INGRES disk, 22 blocks per track * 19 tracks per cylinder = 418 blocks of 512 bytes each.

The following table summarizes the above.

Table 7. Disk parameters and values

parameter	meaning	value
BSIZE	block size	512 bytes
DROT	disk rotation time	.0167 seconds
DAVAC	average access time	.0300
DRATE	data rate to host	.0012 / block
DREAD	cell read time	.0008 / block
DCYL	number 512-byte blocks per cylinder	418 blocks

5.3.2.1.2. CCD physical specifications

For the CCD pages, it will be assumed that both RAP and DIRECT use the high-speed, smaller pages specified for DIRECT [DEWI78]. This assumption is made simply to keep the obviously changeable parameter of page size from entering the analysis. The size of the pages (CSIZE) is taken to be 16k bytes, the DIRECT page size. This is 32 times the block size of the moving-head disk. The scan time for an entire page is CSCAN = .012 seconds for the DIRECT pages. The above is summarized in the following table.

Table 8. CCD parameters

parameter	meaning	value
CSIZE	size of CCD page	16k bytes
CSCAN	page scan time	.012 seconds

5.3.2.2. Speed of the cell processors

It is assumed that the cell processors are fast enough to keep up with the rotation speed of the cell storage media. This speed is the page scan time on CCDs and the cell read time on disks. This is a non-trivial assumption. The implications of this assumption are discussed in this section.

If the cell storage media is a disk track, it can be assumed that the functions of converting the analog signal from the disk to a digital signal, and of error detection and correction, are handled in separate per-track devices. Then the work that the cell processors must do is independent of the storage media. This work is:

1) store the tuples in a buffer until the entire block is read and the per-block error checking done to be certain that the data is correct;

2) if there are qualifications, compare the attribute values in each of the records in the block to the values in the query qualification;

3) if a tuple is a match or is unqualified, any of the following:

   a) perform the arithmetic function specified in the query

   b) transmit the tuple from the buffer to the host over a bus.

   c) transmit the specified attributes to the host

In order to operate optimally, this work must be done at the rotational speed of the cell. This rate is .0008 seconds per 512-byte block on most standard disks. Therefore, the processors have less than 2 microseconds per byte to process the block. If the storage media is CCD pages, the time is about 1 microsecond per byte. If there are three instructions per byte, the cell processor for the disk tracks must be about a 1 MIP processor, and that for the CCD cells a 2 MIP processor. Such speed is within the state-of-the-art processor speeds, but the processors could be very expensive.

The fact that data base machines which require numerous cell processors are at all feasible is due to the following considerations:

1) All 512 characters in the block do not have to be manipulated. The data can be stored in the processor's memory through a DMA device, without taking processor time. The addresses of the attributes to be tested can be stored into cell processor registers by the controlling processor. Then the only data that has to be touched by the cell processor are the attributes involved in the query qualification. There are two figures of merit in this case:

i) the ratio of bytes in the attributes involved in the qualification to the total number of bytes per block (the qualification ratio)

ii) the ratio of equal comparisons to the total number of

tuples (the attribute hit ratio)
If both the attribute hit ratio and the qualifcation ratio are small, the cell processors can be slower and still keep up with the cell cycle time.

2) In [SU75 and BAUM76] it is observed that the cell processors can be made inexpensively to be very fast because they are simple, single function devices. The cell processors do not have to have to functionality of a conventional processor: in conventional processors there are many addressing modes, protection modes, interrupt handling, etc. It is reported in [OZKA77] that the RAP cell processors were bound in speed by the datarate of the CCD pages, not the processor speed.

3) If the cell processor is not fast enough, recovery routes are possible. One is that taken by the data base machines investigated in this analysis: if the query qualification can not be completed in the time alloted, the next block is not read into the cell processor's memory. Then the cell processor must wait a full revolution of the cell storage media until it can read the next block. Another is that the controlling processor would limit the number of attributes in the qualification to the maximum number the cell processors can handle; then it would send the partially qualified tuples to the host. Yet another is to use cell storage mediums that do not revolve, such as bubbles or RAM.

In the following analysis of INGRES benchmark queries,

condition 1, above, held. In the queries that contained qualifications, the qualifications involved only a few attributes per page; additionally, there was a low incidence of equal comparisons. Therefore, the assumption that the cell processors can operate at the rotation speed of the cells is a proper assumption for this analysis, but may not always be reasonable for all systems.

5.3.2.3. Cylinder selection time in DBC

The machine DBC contains special hardware to narrow the number of cylinders that need to be processed. This is done through the use of indices similar to INGRES secondary indices.

The DBC indices specify only the cylinder number of the associated data. Part of the design effort in DBC has been to make the index selection and manipulation very fast. Nevertheless, a time must be associated with the cylinder selection operation in DBC. This is denoted by DBCY. If it is assumed that the indices reside in RAM the time to fetch and manipulate them is probably on the order of the time to fetch a disk- resident page in UNIX, .006 seconds. Therefore, for the numerical analysis it is assumed that DBCY = .006 seconds.

5.3.2.4. Number of cells

The moving-head disk on the INGRES system contains 19 tracks per cylinder. The number of tracks per cylinder is odd since the top track of each cylinder, the 20th track in this case, is reserved for the disk controller to use as a timing track to sense when blocks start and stop. Consequently, the number of cells for those systems that are disk-resident, NDCELL, is assumed to be 19.

The cells in DIRECT are not tightly coupled to the cell processors. There are 8 processors and 32 data cells in the prototype. Therefore for DIRECT the number of processors, NDRP, is 8; the number of data pages, NDPAG, is 32.

RAP tightly couples the processors to the data cells. The controlling costs of the database machines will probably be the expensive high-speed processors. Therefore, for comparison purposes, the number of processors in DIRECT and RAP is assumed to be the same. Therefore, the number of cells in RAP, NRCELL, is assumed for this analysis to be 8, the number of processors present in DIRECT.

The following table summarizes the above.

Table 9. Cell parameters and values

parameter	meaning	value
NDCELL	Number of cells for CASSM, CAFS, DBC and associative disks	19
NDRP	Number of cell processors for DIRECT	8
NDPAG	Number of data cells in DIRECT	32
NRCELL	Number of cells in RAP	8

5.3.2.5. Specification of conventional systems

It is evident that an analysis based on the current INGRES implementation may not lead to general results, due to the particular constraints of INGRES. Therefore, it is assumed for the purpose of this analysis that the conventional system supports a new, high-performance INGRES. The high-performance INGRES has a low amount of overhead and pre-compiles the queries. The overhead is explained in Section 5.3.3.1. On inspection of the INGRES code it appears that a decrease of half the CPU time per query would be possible if the query were pre-compiled. It is assumed that the system supporting INGRES is the system used in the benchmarks.

It is of interest to determine if the conventional system, when augmented with a large number of buffers, will provide significantly improved performance. It is therefore assumed

that there is a 1 Megabyte cache in the augmented system. Therefore NBUF, the number of 512-byte pages in the cache, is 2000. This augmented system will be denoted as "Large Buffer".

5.3.3. Host time

The host for all the backend machines, and the entire system for the conventional machines, is assumed to be a DEC PDP 11/70.

There are two functions that the host system must perform no matter what machine is used as the backend system. These are the functions of processing the query to put it in a form acceptable to the backend and communicating with the backend. The former is denoted as "overhead", the latter "communication time". The time that these functions take is described in the following sections.

5.3.3.1. Overhead

Each of the backend machines will accept the query in a form similar to that which is sent to the INGRES process "Decomp". That is, the query must have been retrieved from the user's terminal, parsed and validity checked before it can be sent to the backend. This time shall be split into two parts for this analysis: OVCPU denotes the overhead CPU time, and OVIO denotes the overhead I/O time.

It was shown in Chapter 4 that a back-end machine will not significantly affect response time if the query is heavily overhead-intensive. Therefore the only queries that need be analyzed are those for which the overhead is small. To assign numerical values to OVIO and OVCPU it shall be assumed that the queries are pre-compiled, and that terminal communication time is so small that it can be disregarded.

Both of these assumptions heavily favor backend systems. If the host overhead time per query is very large, the added advantage of the backend is reduced. It is a realistic assumption because any system involving a backend machine would surely implement the functions of pre-compiling the queries and fast terminal handling.

The usual strategy for pre-compiling queries is that the user defines a query prior to execution time; the data management system parses, validity checks, and sets up a run-time module for the query. When it is executed, information about what the query depends upon (e.g. presence and type of attributes) must be checked. The minimal I/O time for such checking is one reference per relation. Therefore OVIO = DAVAC = .030 seconds per relation. This is assuming the validity checking is done entirely in the host and does not involve the backend.

The overhead CPU time (OVCPU) is the time to perform the work outlined above. It takes .006 seconds of CPU time to

request a page of data from the operating system and transfer it into the user space. In Table 4.2 it is reported that the total OVQP time for an average query in the short3 query stream was .28 seconds. Of that time, .12 seconds was data processing time, and .16 seconds was overhead. This overhead was the time for validity checking and setting up a minimal query. Clearly the time for the minimal overhead must fall between .006 seconds and .16 seconds. Where it falls between these two values depends on the functionality of the validity checking: the amount of optimization present or possible, the level of security, etc. These two values shall be assigned to OVCPU as best case -- worst case assumptions. It must be pointed out that this is an absolute minimum; that it will be much more common that there is more work to do. If the database has changed, and the query must be re-compiled, the extra work of creating the run-time module actually increases the overhead. Any security algorithm would add overhead to check access rights. Additionally, any natural-language interfaces would also increase the overhead.

5.3.3.2. Host - to - Backend Communication Time

BCOM denotes the host CPU time to send a query to the backend system and receive the results. To assign a numerical value to BCOM, the following considerations must be taken into account.

In UNIX, the inter-process communication time (from chapter 4) is .12 seconds for 5 processes to send messages to and receive messages from each other linearly. This is .03 seconds for a send-and-receive transmission per process pair. The time, in UNIX, for a process to request a page of data from the operating system, and to receive that page, is .006 seconds if the page is in memory. Whether the time to communicate between the backend system is closer to the inter-process time or the page-transfer time depends on the design and the functionality of the interface. Rather than attempting to narrow the time any further, the .006 sec. time is assigned as the "best case" time, and the .03 sec. time as the "worst case" time to set up a single communication with the backend system.

5.3.3.3. Host data processing time

Each of the machines relies on the host to format the results for printing and move the results to the user's terminal. Some rely on the host to perform arithmetic functions. The amount of time required in the host is query and machine dependent. It is denoted by HDP, the host data processing time.

The host parameters are summarized in the following table.

Table 10. Host parameters and values

parameter	meaning	value best case --worst case
OVIO	host overhead I/O time	.0300 seconds
OVCPU	host overhead CPU time	(.006--.1600) sec.
BCOM	host CPU time to communicate with backend	(.006--.0300) sec
HDP	host time to format results for printing, perform math functions, etc.	query dependent

5.3.4. Query performances

It will be recalled that there were three classes of benchmark queries: data-intensive, overhead-intensive, amd multi-relational queries. Four queries were chosen for the following analysis: one from short3 (S1), one from long3 (L1), the "ROOMS" query (M1), and an aggregate function query from "long2" (A1). The query S1 is an overhead intensive query; L1 is a data intensive query; M1 is a multi-relation query; and A1, an aggregate function, is also a data intensive query.

The first part of this section contains an analysis by query type. Both the total work and response times are calculated for each query on each machine. The total work is the sum of the time spent in all components of the machine. The response time is the sum of the component times that cannot be overlapped. The second part contains the conclusions.

5.3.4.1. Short queries

The query chosen for this analysis is:

query S1:

```
retrieve(QTRCOURSE.day, QTRCOURSE.hour)
   where QTRCOURSE.instructor = "despain, a.m."
```

QTRCOURSE contains 1110 tuples. Each tuple has 24 attributes, and is 127 bytes long. The relation can be stored as a heap in 274 pages. The attribute "day" is a character field, 7 bytes long; "hour" is also a character field, and is 14 bytes long. The query resulted in 3 qualified tuples returned to the user.

This particular query was picked because its performance characteristics match those of the "average query" in the benchmark short3, as reported in Table 1 of Chapter 4. In the following machine comparisons the following considerations are important.

1) The relation fits on one cylinder because the number of pages is less than DCYL, the number of disk blocks per cylinder.

2) Since there were only four tuples returned to the host, the time to process the retrieved data and send it to the user terminal is assumed to be 0. Also, since the hit ratio is so low, it is assumed that there is not a problem with bus contention or controller processor speed.

3) There is one relation in the query, so the host overhead cpu time, OVCPU, is a constant factor, (.006--.160) seconds. OVIO, the host overhead I/O time, is .030 seconds.

The machine comparisons follow.

5.3.4.1.1. Fast INGRES

The work to execute this query in a "fast INGRES" system is :

FWORK = OVIO + OVCPU + DPIO + DPCPU

OVIO = overhead I/O time = .0300 seconds.

OVCPU = overhead CPU time = .006--.160 seconds.

DPIO, the data processing I/O time, is calculated in the following section.

In chapter 4 it is reported that there were 11 I/O references to process this query: three to the QTRCOURSE relation, and eight to system relations. The references to QTRCOURSE are the data processing references. QTRCOURSE was hashed on instructor name. The three QTRCOURSE references include two overflow pages and one main page. The time for the three references is in the worst case an average access per reference. This would occur if the pages reside on separate cylinders. If they are on the same track the best case condition arises. Then the reference time is one access and 3 block transfers. Therefore:

DPIO = DAVAC + 3*DRATE, best case -- 3*DAVAC, worst case

numerically, this is

DPIO = .0336 best case -- .090 worst case

DPCPU, the data processing CPU time, is calculated below. In Table 2 of Chapter Four it is reported that the query required .12 seconds of data processing CPU time. Since the "Fast INGRES" system is compiled, the data processing CPU time will be half of the measured interpreted query. Therefore DPCPU = .06 seconds.

The total work that a query incurs is a measure of the impact of the query on system performance. The total work for this query is:

```
FWORK = OVIO + OVCPU + DPIO + DPCPU
      = OVIO + OVCPU + ((DAVAC+(3*DRATE)) -- 3*DAVAC)
                + DPCPU
      = .0300 + (.006--.160) + (.0336--.0900) + .06
      = (.13 --.34) seconds
```

The response time is that amount of the work which cannot be overlapped with any other part of the work. Clearly the overhead CPU and I/O times must be done serially since the one I/O reference contains information necessary for the validation of the query. However, the data processing I/O and CPU times can be overlapped. The response time is therefore:

```
FRES = OVIO + OVCPU + max [DPIO, DPCPU]
     = OVIO + OVCPU +
        max[((DAVAC+(3*DRATE)) -- 3*DAVAC) , DPCPU]
```

The best case response time uses the maximum best case

values; the worst case the maximum worst case values.

FRES = (.096 -- .280) seconds

The best case is dominated by the CPU time to process the 3 pages which at .06 seconds is almost half of the total best case work and almost two-thirds of the best case response time. The overhead CPU, at .16 seconds worst case, dominates the worst case work and response times.

5.3.4.1.2. Large Buffers

It is assumed that the presence of large buffers only serves to decrease the I/O time, not the CPU time. This is a slight simplification, since the operating system does require time to schedule requests to disks, field interrupts, etc., in processing I/O requests. This time would be saved if the data were already in memory, but it is assumed to be negligible in relation to the time the queries require.

If that time is very small the large buffer system uses the same amount of CPU time as the fast INGRES system.

The I/O time for query S1 is composed of the one page of overhead and three pages of accesses to the QTRCOURSE relation; therefore, with buffers large enough to cache the system relations, the I/O time would decrease to the time to

access the QTRCOURSE relation. This time, as calculated above, is at best.0336 seconds, and at worst .09 seconds. If the entire relation is stored in the cache, the time is, of course, zero.

Then the work in large buffer systems is:

```
LWORK = OVIO + OVCPU + DPIO + DPCPU
LWORK = 0 + (.006--.160) + (0--.0900) + .06
      = (.066--.229) seconds
```

and the response time is the same:

LRES = (.066 -- .229) seconds

The best case time is dominated by the data processing CPU; the worst case time by the overhead CPU time.

5.3.4.1.3. Associative Disks

The associative disks and CAFS have the same functionality for this query. The time in either system is:

AWORK = OVCPU + OVIO + BCOM + DAVAC + DROT

BCOM, the time to communicate with the backend, is (.006--.030) seconds. There must be one disk access to position the machine at the right cylinder. Then, since the data resides on a single cylinder, and since the cell processors operate at the rotational speed of the disk, the data processing will take one revolution of the disk.

Numerically, this is:

AWORK = (.006--.160) + .030 + (.006--.0300)
+ .030 + .0167
= (.0887--.2667) seconds.

The response time is exactly the same,

ARES = (.0887 -- .2667) seconds.

The major components of the best case time are the two disk access times: one to perform the overhead functions, the other to position the disk arm for data processing. The worst case time is dominated by the host CPU overhead.

5.3.4.1.4. CASSM

First the storage requirements for the QTRCOURSE relation on CASSM must be determined. There are 1110 tuples in the relation, so assuming a two-byte encoding of the relation name there must be 2220 bytes to store the per-tuple relation name. There are 25 attributes per tuple; assuming a 1 byte encoding of attribute names is stored per attribute per tuple, 27,750 bytes are necessary to identify attributes. Each tuple is 127 bytes long; however, in CASSM no character fields over 4 characters long are directly stored. They are stored once, then pointers to the correct character string are stored in each tuple. Using that method for storing the relation requires 74 bytes per tuple, which includes 9 pointers to character strings. The 9 attributes that are represented as character strings were measured to require 6417 bytes of storage. So the total storage required is 221 blocks, validating Langs conjecture that the extra storage

required for the delimitors in CASSM is offest by the data encoding algorithm it uses [LANG78]. This data will fit on one cylinder.

Therefore, the work performed in this query is:

CWORK = OVCPU + OVIO + BCOM + DAVAC + n*DROT

The number of rotations, n, is:

```
1 rotation to mark all tuples
        for the relation QTRCOURSE
1 rotation to find the pointer to
        character string "despain,a.m."
1 rotation to mark all tuples with
        the pointer to "despain"
1 rotation to return all "day"
        attributes in marked tuples
1 rotation to return all "hour"
        attributes in marked tuples
----
5
```

It is assumed that CASSM sends the attributes to the host in their coded form, and also sends the decoding information. The host will therefore have the CPU time to decode the attributes, but since there were only 3 tuples returned from this query, that time can be disregarded.

The value for n for this query is therefore 5, and the work to process the query is:

```
CWORK = (.006--.16) + .030 + (.006--.030)
                + .030 + 5 * .0167
      = (.156--.234) seconds
```

The response time is the same

CRES = (.156 -- .234) seconds

The best case time is dominated by the five rotations. The major component in the worst case is the same as in the previous systems, the host overhead CPU time.

5.3.4.1.5. DBC

The time in DBC is:

DBWORK = OVCPU + OVIO + BCOM + DCYL + DAVAC + DROT

This differs from the associative disk case in only the time required to perform the cylinder select in DBC, DCYL. If that time is .006 seconds, the work in DBC is:

DBWORK = (.0947--.273) seconds

Since the cylinder selection must be done before the data processing can begin, it cannot be overlapped. The response time is therefore the same as DBWORK,

DBRES = (.0947 -- .273) seconds

5.3.4.1.6. DIRECT

The work DIRECT must perform is:

DWORK = OVCPU + OVIO + BCOM + n*CSCAN + DPIO

CSCAN is the CCD page scan time; DPIO is the time to read QTRCOURSE from the disk.

First, the value of DPIO will be determined. The QTRCOURSE relation is 274 512-byte blocks long so it will fit in 9 of DIRECT's 16K byte data cells. In this query the cell processors will read from one data cell and write the requested attributes of the qualified tuples to another cell, thus forming a temporary relation. Therefore 8 of the cells must be reserved as processor output cells. Assuming no other queries are running, 24 cells are available to read in the relation. Therefore the entire relation can be read at once. DPIO therefore is one disk seek plus the transfer time. Of course the best case I/O time is if the relation is already in DIRECT's buffers; in that case the time is zero.

```
DPIO = (0 -- (DAVAC + 274*DRATE))
     = (0 -- .030 + (274*.0012))
     = (0 -- .359) seconds
```

Next, the number of page scans required (n) is determined. Since 9 data cells are required to store this relation, and there are only 8 cell processors, two scan times are required to write the qualified tuples to the temporary relation: one for the first 8 pages, another for the last one.

If the partial results of queries are stored in separate CCD pages, DIRECT exhibits performance problems. These problems are related to the assignment of processors to CCD pages, and are explained in [DEWI79]. In order to avoid the

problems, the partial pages are compressed into a fewer number of full pages. Since there are 7 idle processors in the second scan, the compression of the first 8 temporary relation pages can be done during that scan. The compression of the remaining page and the work of sending it to the host can certainly be accomplished in a third scan time. Therefore n, the number of scans required, is 3 for this query.

Then the total work is:

$$\begin{aligned} DWORK &= OVCPU + OVIO + BCOM + 3*CSCAN \\ &\quad +(0 \text{ -- } (DAVAC + 127*DRATE)) \\ &= (.006 \text{ -- } .16) + .030 + (.006 \text{ -- } .030) + 3*.012 \\ &\quad + (0 \text{ -- } .359) \\ &= (.078 \text{ -- } .615) \end{aligned}$$

The response time in the best case is the same as the total work, because no overlapping is possible. If the pages must be read from secondary storage, as in the worst case, the data transfer time for the blocks entering the last data cell can be overlapped with the processing of the first 8. Since there are 32 512-byte blocks per data cell, the response time is

$$\begin{aligned} DRES &= OVCPU + OVIO + BCOM + 2*CSCAN \\ &\quad + (1*CSCAN \text{ -- } \\ &\quad\quad (DAVAC + 242*DRATE + \max(1*CSCAN, 32*DRATE)) \\ &= (.006 \text{ -- } .16) + .030 + (.006 \text{ -- } .030) + .024 \\ &\quad + (.012 \text{ -- } (.030 + .29 + \max(.012, .038) \\ &= (.078 \text{ -- } .603) \end{aligned}$$

In both the response and work times, the best case is dominated by the scan times for the CCD data cells, and the worst case by the data transfer time from the disk.

5.3.4.1.7. RAP

The time in RAP is:

RWORK = OVCPU + OVIO + BCOM + n*CSCAN + DPIO

First the time to read the data from the disk, DPIO, will be discussed.

In order to provide RAP with the same amount of storage as DIRECT, it is assumed that there are 24 16K-byte CCD pages available to act as a buffer cache. These are not a part of any proposed or actual design of RAP, and are not represented in the RAP block diagram (figure 5.3). On that diagram the buffers would appear between the disk and the controlling processor. These pages will be denoted by "buffer cells" to differentiate them from the 8 cells attached to the cell processors (the "processor cells").

QTRCOURSE, because it is 9 cells long, will not completely fit in the RAP processor cells. If QTRCOURSE is not already in the cache when the query begins, a reasonable algorithm to read it in is to write it into both the buffer cells and the processor cells at the same time. Otherwise the first 8 cells would have to be loaded into the buffer cells, then

into the processor cells, which takes more time.

The data must be sequentially sent from the buffer cells to the processor cells because to do otherwise would require a cross-point switch. Therefore the most reasonable best case time occurs if the relation is stored in the buffer cache, is the transfer time of the first 8 cells, or 8 * CSCAN. The worst case time is the same as in DIRECT.

DREAD is :

DREAD = (8 * CSCAN -- (DAVAC + 274*DRATE)

The following section calculates the value for n, the number of cell scans that must be performed for data processing. The number of cell scans is:

```
1       to mark qualifying tuples
                in 8 processor cells
1       to send marked tuple to host
1       to transfer the 9th page from buffer
                to processor cell
1       to mark 9th page
1       to send marked tuples in 9th page to host
--
5
```

Therefore n is 5, and the necessary work to perform the query is:

```
RWORK = OVCPU + OVIO
          + BCOM + (8*CSCAN -- (DAVAC + 274*DRATE))
          + 5*CSCAN

      = (.006 -- .16) + .030 + (.006 -- .030)
        + (.096 -- .359) + .060

      = (.20 -- .64) seconds
```

The response time is the time above, except that the disk transfer rate for the last cell can be overlapped with the scan time for the previous cells. The best case response time remains the same as the work, and the worst case response time is .012 seconds less. The response time is therefore:

RRES = (.20 -- .63) seconds

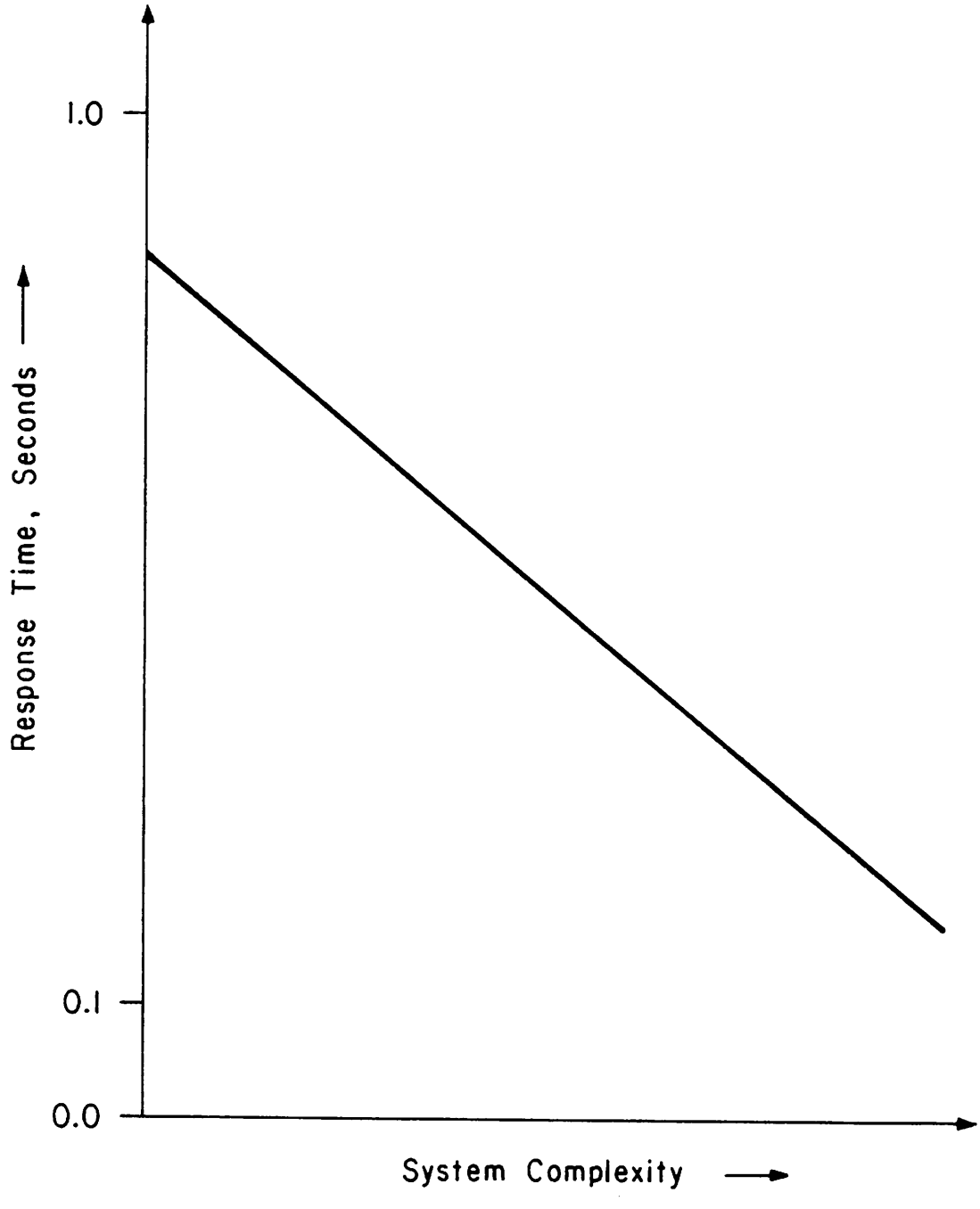


Figure 5.6 Response time vs. Complexity
Hypothetical

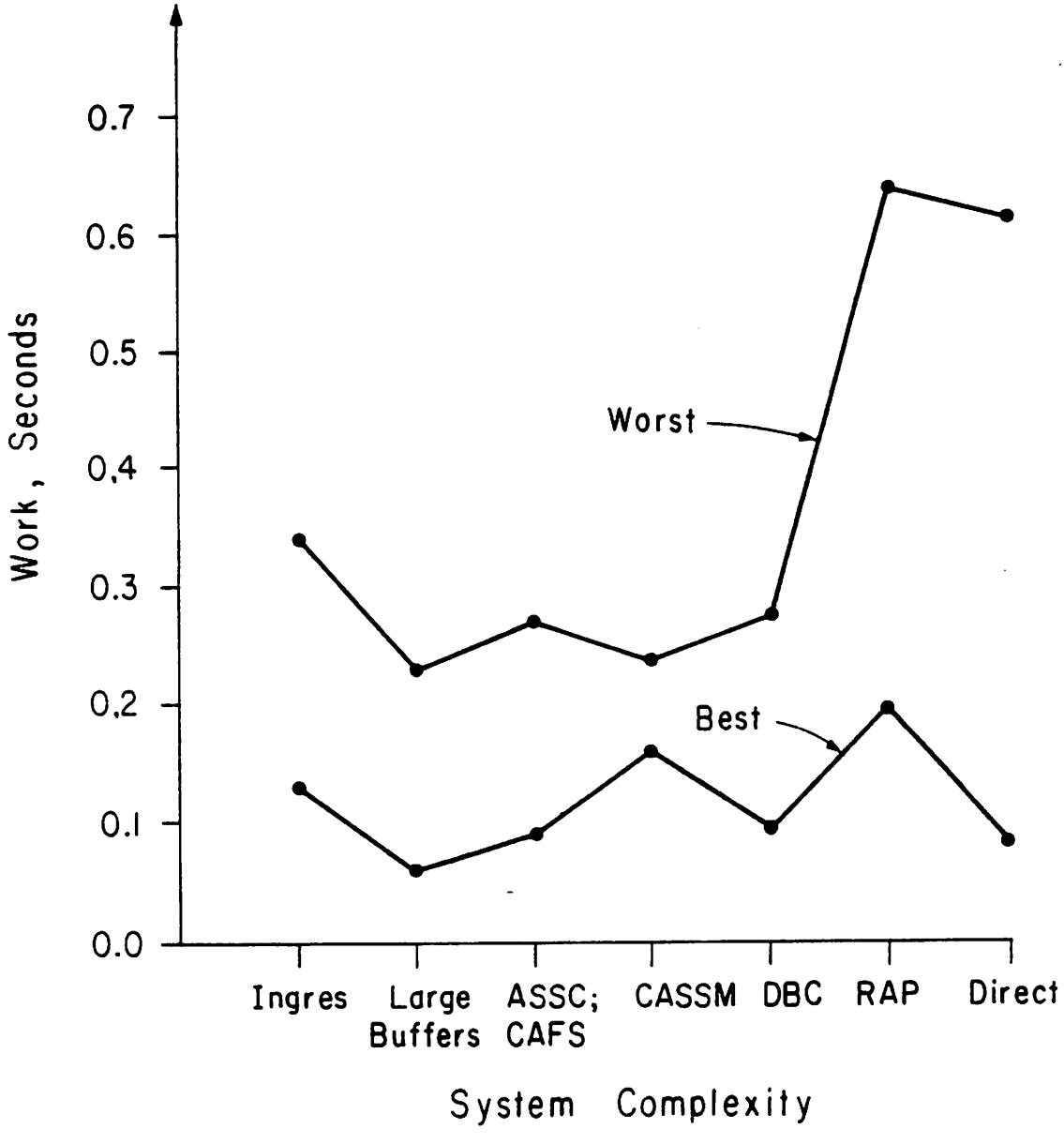


Figure 5.7 Total Work vs. System Complexity
Query S1

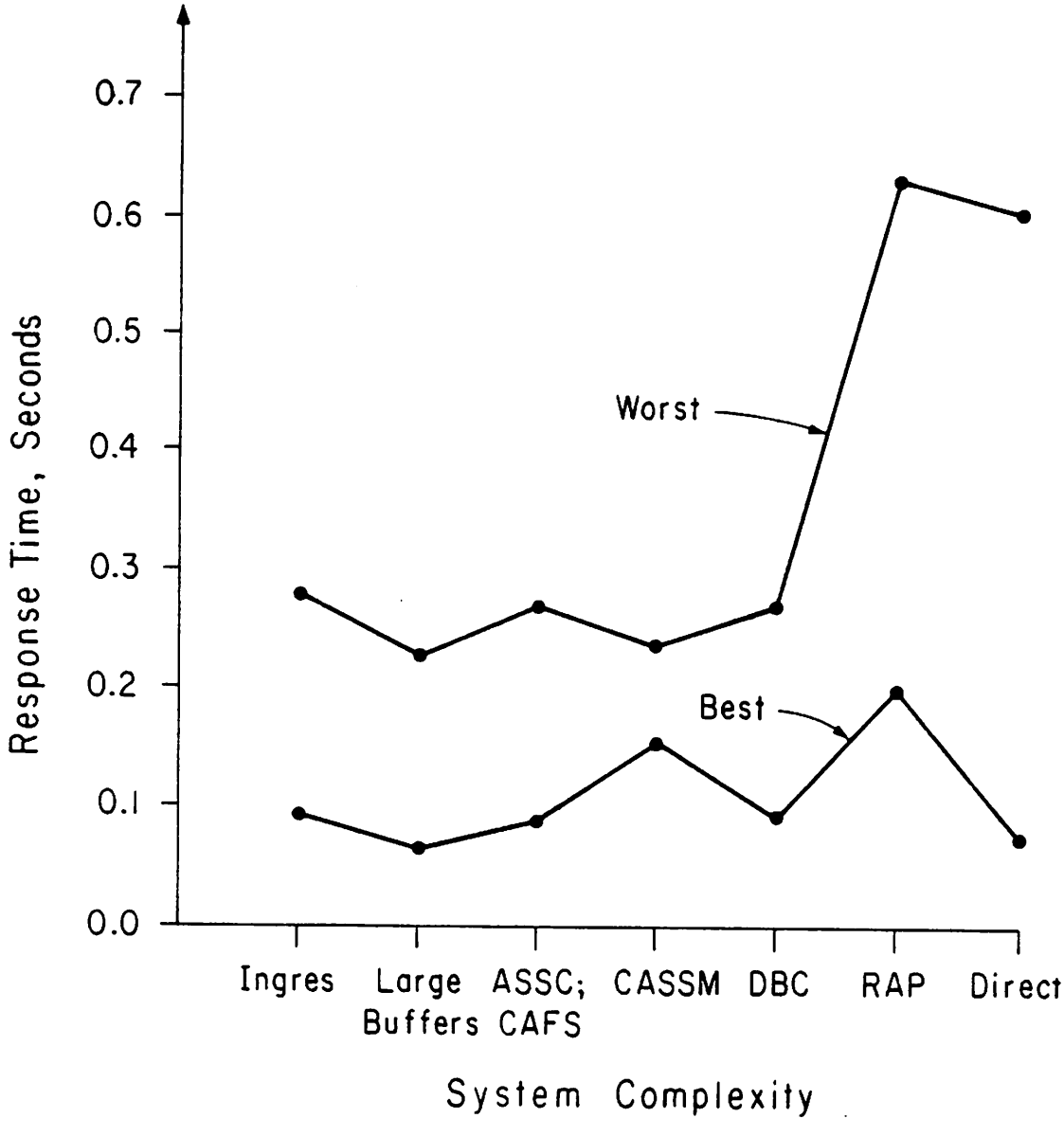


Figure 5.8 Response time vs. Complexity
Query S1

5.3.4.1.8. Conclusion

Figure 5.6 shows a hypothetical curve relating system work to a machine's complexity. It is to be expected that a more complex system, where that complexity consists of added buffers, processors, etc., will result in less response time and less system work.

In Figure 5.7, the system work for each of the systems described above is plotted against the complexity of the system.

The systems are ordered along the horizontal axis as follows:

1) INGRES, the "basic" system
2) Large buffers added to a basic system
3) Associative Disks, CAFS
   the cell processors are relatively simple
4) CASSM
   more intelligence in the cell and controlling processors
5) DBC
   highly functional controlling processor
6) RAP
   very fast cell processors; CCD cells
7) DIRECT
   cross-point switch

It is apparent that neither the best nor worst case curves follow the expected curve. Figure 5.8 relates the response time for query S1 to the machine complexity. It also shows that the increased complexity of the machines do not result in faster response times. In

both figures 5.7 and 5.8 the worst case times of the caching systems (RAP and DIRECT) is very much higher than the worst case INGRES time. That is because INGRES maintains QTRCOURSE as a hashed relation, and only read three pages of it. The caching systems had to serially read the entire relation.

Even the best case times of the machines barely fluctuate below the INGRES best case. RAP and CASSM actually require more time than INGRES because they must scan the cells so many times.

It is apparent that, for this query, the increased cost and complexity of the database machines do not result in a significant increase of performance.

5.3.4.2. Long Queries

The query chosen as query L1 is:

```
retrieve (outstand.fund, outstand.acct)
```

This query was chosen because its performance pattern was very close to the average for the benchmark it is from, the "long3" benchmark. The relation "outstand" has 914 tuples of 97 bytes each. In INGRES it is a compressed heap, and requires 145 pages of storage. The query prints the fund and account numbers for each entry in the relation.

The following conditions apply to the analysis of this query:

1) None of the database machines support compressed relations. OUTSTAND was 185 pages when uncompressed. The data can be entirely stored within one cylinder of the disk.

2) The time to perform the host data processing is not trivial for this query, since 914 values are returned to the host.

3) The query is a single-relation query, so the host overhead CPU time remains the same as in Section 5.3.4.1.

4) There was no duplicate suppression in the INGRES benchmark query, so no time is allotted to that function in the following analysis.

5) This query is an unqualified projection of a relation. In the backend database machines it is quite likely that such a query will exhibit performance problems due to bus contention or buffering speed limits in the controlling processor. In order that the analysis be as uncomplicated as possible, these problems are ignored. This greatly favors the backend systems. That is because the bus contention and buffering problems can be a serious drain on the database machines' performance.

5.3.4.2.1. Fast INGRES

The work in INGRES is:

FWORK = OVCPU + OVIO + DPIO + DPCPU

The time to perform the data processing I/O, DPIO, is first discussed.

It is assumed that the relation is stored on a single cylinder, so only the initial seek is required to access it. However, few standard systems can or will furnish the 145 pages necessary to store the entire relation if it is read all at once. Therefore it is assumed that the standard system reads it a track at a time. The relation occupies 8 tracks; to read the first track requires the average access time, DAVAC. If the disk heads remained positioned at the cylinder that the relation resides on, reading the remaining 7 tracks requires only the average latency time. This time is one-half of a disk revolution time, or DROT/2. DPIO is therefore:

```
DPIO = DAVAC + (7 * DROT/2) + 145*DRATE
     = .030 +  (7 * .008) + 145*.0012
     = .26 seconds
```

The DPCPU time is discussed below.

When run on a standard system this query required 3.2 seconds of CPU time. A compiled system by assumption will require half of that, so DPCPU is 1.6 seconds.

The total work is therefore:

```
FWORK = (.006 -- .16) + .030 + .26 + 1.6
      =  (1.90 -- 2.05)
```

The DPIO and DPCPU times can be overlapped. Therefore the

response time is:

FRES = (.006 -- .16) + .030 + 1.6
= (1.636 -- 1.79) seconds

The time, in all cases, is dominated by the CPU data processing time.

5.3.4.2.2. Large buffers

In a large buffer system, the time is

LWORK = OVCPU + OVIO + DPIO + DPCPU

The times are as defined for the INGRES system, except the I/O time. In the best case, the relation is in the buffers and the time is 0. Otherwise, the presence of the buffers insures that the relation can be read a cylinder at a time. so DPCPU becomes:

DPCPU = (0 -- (DAVAC + 145 * .0012))

The total time is therefore:

LWORK = (.006 -- .16) + 0 + (0-- (.03 + .174)
+ 1.6
= (1.61 -- 1.96) seconds

The data processing I/O time can be overlapped with the CPU time in the worst case. The response time is therefore:

LRES = (1.61 -- 1.76) seconds

The work and response times are dominated by the CPU time.

5.3.4.2.3. Associative disks and CAFS

The associative disks and CAFS require the following time:

AWORK = OVIO + OVCPU + BCOM + DAVAC + DROT + DTRANS + HDP

The host overhead I/O, CPU and backend communication times are the obvious quantities. DAVAC, the disk access time, is necessary to position the disk arm. DROT is the rotation time of the disk, the time to process the query. DTRANS is the time to transfer the data to the host machine; HDP is the data processing time in the host.

DTRANS will first be calculated.

Both of these machines return the full tuple to the host instead of qualified attributes. It is assumed that:

1) the database machine is connected to the host by a channel that has the same maximum datarate as that of the disk channel, DRATE sec / block, and

2) the machine controller can buffer pages that the host is not ready for

Then the maximum speed that the data can be sent to the host is DRATE per block. Therefore DTRANS = 185 * DRATE

Now HDP will be calculated.

The host processor must perform the exact functions on the tuples as INGRES does, so the host processor time, HDP, is assumed to be the same as in the INGRES system.

The total work is therefore:

```
AWORK = OVIO + OVCPU + BCOM
        + DAVAC + DROT + 185 * DRATE + HDP
      = (.006--.16) + .030
        + (.006--.03) + .0167+ 185*.0012 + 1.6
      = (1.88 -- 2.06) seconds
```

The data transfer and disk rotation time scan be overlapped with HDP, so the response time is:

```
ARES = (.006--.16) + .03 + (.006--.03) + 1.6
     = (1.64 -- 1.82)
```

The data processing time in the host dominates both the work and CPU times.

5.3.4.2.4. CASSM

The time in CASSM is

CWORK = OVIO + OVCPU + BCOM + DAVAC + n*DROT + HDP + DTRANS

OVIO, BCOM, DAVAC have been defined above.

HDP, the time to format the tuples for printing, is at least as great as in the INGRES system because the attributes are received in coded format, and the host must decode them.

The number of rotations required is:

```
1       mark tuples of OUTSTAND
1       retrieve attribute "fund"
1       retrieve "out"
1       retrieve decoding information
---
4       n, number of rotations required
```

DTRANS is the time to send the coded representations of the attributes and the decoding information to the host. The data to be sent takes 4 512-byte blocks. If the same channel is used to connect CASSM to the host as is used for the INGRES disks, and if there is no bus contention, that time is DRATE * 4.

then CWORK is:

CWORK = .03 + (.006--.16) + (.006--.03)
+ .03 + 5*.0167 + 1.6 +.0048
= (1.77 -- 1.94) seconds

CRES = CWORK since there is no significant overlap possible.

5.3.4.2.5. DBC

The time in DBC is:

DBWORK = OVCPU + OVIO + BCOM + DCYL + DAVAC + DROT
+ DTRANS + DHP

The time to transfer the data to the host, DTRANS, will be discussed in the following paragraph.

The transfer time is the time to transmit the data from the controlling processor to the host. It is assumed that the controlling processor will buffer the data coming from the cell processors. When a 512 byte block is filled it is sent to the host. Since the qualified attributes will fit in four blocks, DTRANS is the time to transfer four blocks.

Therefore,

DTRANS = 4 * DRATE

DHP shall be calculated in the following paragraph.

An important question is to what extent the host data processing CPU time is reduced when the database machine returns only the needed attributes. The functions that the database machine has performed for the host are:

1) physically separated the "fund" and "acct" attributes from the other attributes in the tuple.

2) sent less data to the host, so the host is relieved of the CPU time to request and transfer pages into its working space.

The CPU time in INGRES to transfer a page is .006 seconds. Therefore the 1.6 seconds of the INGRES CPU data processing time can be reduced to .75 seconds because the number of page reads is reduced from 145 to 4, and because the query is assumed to be compiled. It is estimated, from inspection of the INGRES code, that the actual time to separate the attribute from the tuples takes one tenth of the remaining time. Therefore HDP is .68 seconds.

The total time in DBC is therefore:

$$\begin{aligned} DBWORK &= OVCPU + OVIO + BCOM + DCYL + DAVAC + DROT \\ &\quad + DTRANS + DHP \\ &= (.006 - .16) + .03 + (.006 - .03) + .006 + .03 \\ &\quad + .0167 + .0048 + .68 \\ &= (.78 - .96) \text{ seconds} \end{aligned}$$

The response time is the same.

5.3.4.2.6. DIRECT

The work DIRECT must perform is:

DWORK = OVCPU + OVIO + BCOM + n*CSCAN + DPIO + DHP + DTRANS

DPIO will first be calculated.

The 185 pages of data fit in 6 of DIRECT's data cells. Since the relation can be completely read into the buffers, the read time is:

DPIO = DAVAC + 185*DRATE

If the relation is already in the buffers the read time is 0.

The number of scans required (n) is calculated next.

DIRECT will send this query to the processors as a "project" command, so they will not attempt to perform any qualification on the tuples, merely move the needed attributes from a page in the source relation to a page in a temporary relation. Therefore the number of scans necessary is:

1	6 processors moving tuples; 2 compacting
1	4 processors compacting; 4 sending data to host
1	rest of data is sent to host
--	
3	n, number of scans

The above numbers assume that as one cell processor puts data into a temporary relation cell, another can be reading

from that cell to compact the relation into a third cell. This assumes that the controlling processor can schedule the cell processors to perform these functions, and is highly favorable to DIRECT.

DTRANS, the time to transmit the qualified attributes to the host, is included in the scan time and need not be separately calculated.

HDP, the host data processing time, is the same as in the host for DBC, since DIRECT transfers only the necessary attributes.

The total time for DIRECT is therefore:

```
DWORK = OVCPU + OVIO + BCOM
                + 3*CSCAN + (0 -- (DAVAC + 185*DRATE))
                + DHP
      = (.006--.16) + .03 + (.006--.03)
                + .036 + (0 --(.03+.222))
                + .68
       = (.758 -- 1.188) seconds
```

The response time is exactly the same.

The data processing time in the host dominates the query times.

5.3.4.2.7. RAP

The time in RAP is:

RWORK = OVCPU + OVIO + BCOM + n*CSCAN + DPIO + DHP

The work in RAP will be the same as the work in DIRECT, except that the number of scans is different.

The number of scans (n) is calculated below.

```
RAP requires

1        scan to mark tuples
1        scan to send marked attributes to host
--
2       total scans, n
```

Therefore the total work for RAP is:

```
RWORK = OVCPU + OVIO + BCOM + 3*CSCAN
              + (0 -- (DAVAC + 185*DRATE)) + DHP

      = (.746 -- 1.176) seconds
```

The response time is the same.

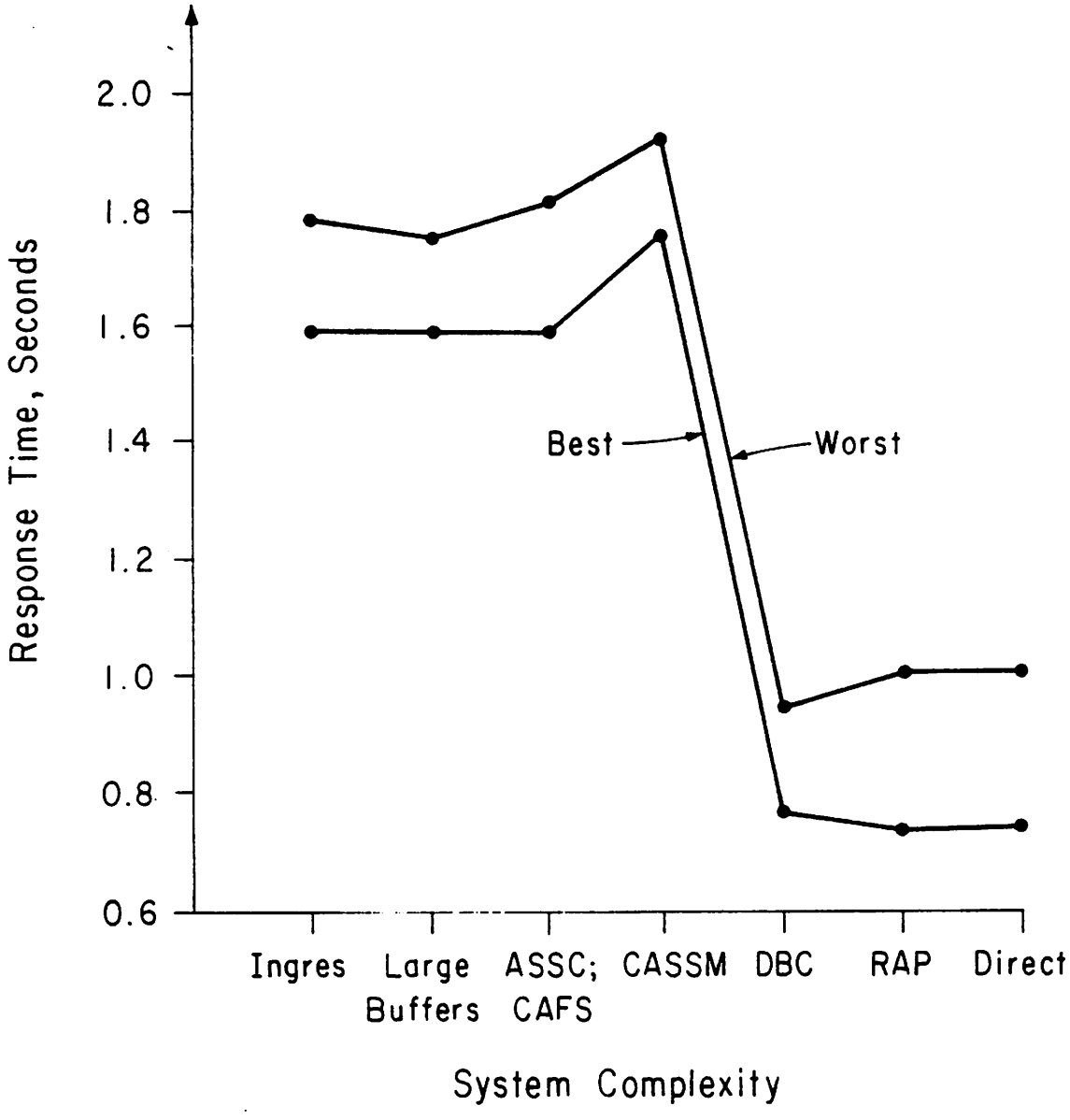


Figure 5.9 Response time vs. Complexity
Query L1

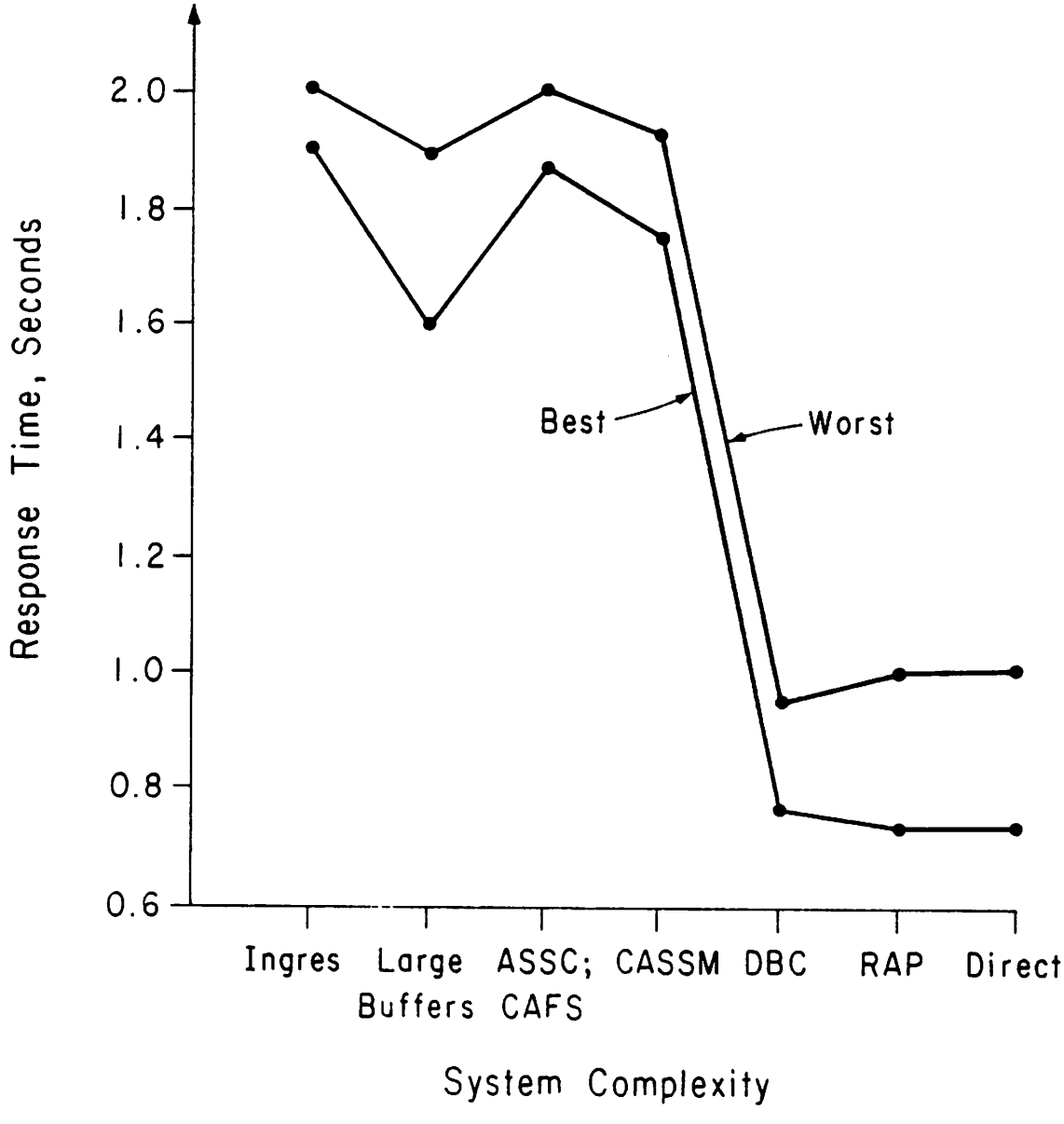


Figure 5.10

Total Work vs. System Complexity
Query A1

5.3.4.2.8. Conclusion

Figures 5.9 and 5.10 are the comparisons of response and total work to system complexity. As in Query S1, neither time decreases significantly when compared to the increased complexity of the systems. We note the following points:

1) The effect of the query was simply to move data. In this case, it can be expected that the response time is not significantly improved by the use of database machines.

2) The the major improvement in the response time that was seen was in those machines that sent only the needed attributes, not the entire tuple, to the host. The specific improvement that can be thus obtained is entirely dependent upon the implementation in the host system. If the system implements the splitting apart of the tuples into the attributes in a very fast manner, the improvement will be much less than is shown here.

3) There were several assumptions made that heavily favored the backend machines. These were:

i) There is no bus contention.

ii) The controlling processor is smart enough, and fast enough, to fully buffer the results and send them to the host without itself becoming a bottleneck.

iii) The controlling processor is capable of synchronizing the cell processors in the most optimal way.

In spite of these assumptions, the improvement of performance effected by the use of the database machines is not

commensurate with their increased complexity.

4) The query response time and work were in all cases bound by the speed of the host in processing the output data. In other systems, or with other queries, this time will vary from the time estimated in this analysis. Nevertheless, in an unqualified scan of an entire relation it is to be expected that most of the work of the system must reside in that part of the system which handles the output functions. This class of queries is typical in reports.

5.3.4.3. Aggregate functions

An aggregate function is an aggregate with a "by clause", which is a "group by" operator. The following query was chosen for the aggregate function query, A1:

```
retrieve (GMASTER.acct,
                GMASTER.fund,
                encumb = sum (GMASTER.encumb by
                        GMASTER.acct,
                        GMASTER.fund))
```

This query was chosen because it is typical of the aggregate functions in the long2 benchmark. GMASTER is a relation of 194 tuples, 2 tuples per page. There are 17 unique values for the (acct,fund) pair. The query returns to the user the 17 unique (acct,fund) pairs, and their associated sums.

The following are considerations in analyzing this query.

1) The relation is 94 pages long; therefore it will reside on a single cylinder.

2) The query is a single-relation query, so the host overhead time is the same as in the previous queries

5.3.4.3.1. Fast INGRES

The work to execute this query in a "fast INGRES" system is:

FWORK = OVIO + OVCPU + DPIO + DPCPU

OVIO and OVCPU are the same as in previous queries. DPIO is calculated in the following paragraph.

GMASTER can be stored in 5 tracks of the disk. As is explained in section 5.3.4.2.1, the time to perform the read is:

DPIO = DAVAC + (4*DROT/2) + 97*DRATE

DPCPU must be calculated next.

The optimal algorithm to perform this query is to read the GMASTER relation once, retaining in memory only the (encumb,acct,fund) attributes from each tuple. These are stored in a temporary storage area, denoted by T1. T1 is then sorted on (acct, fund), and the sums performed and output.

The CPU time to execute this algorithm is composed of the time to perform the scan of GMASTER and the projection (HSCAN), the time to perform the sort (HSORT), and the time to perform the sum (HSUM). These are calculated below.

HSCAN is the CPU time to read the 97 pages of the relation,

which is .006 seconds per block, plus the time to perform the projection. This time can be no more than the .006 seconds per page that is required by the operating system.

Therefore

HSCAN = .012 * 194 = 2.328 seconds

HSORT, the time to sort the (encumb,acct,fund) tuples will be calculated next.

The (encumb, acct, fund) tuples are 24 bytes long; 194 of them are stored in 9 pages. On the 11/70, sorting 9 pages requires about .24 seconds of CPU time. Therefore HSORT = .24 seconds.

The time to perform the sum, HSUM, can require no more time than to sort them. Therefore HSUM = .24 seconds.

Then the total CPU time to process the query is:

DPCPU = 2.568 seconds

Therefore the time to process query A1 is:

```
FWORK = OVIO + OVCPU + DAVAC + (4*DROT/2)
              + 97*DRATE + DPCPU
      = (.006--.16) + .03 + .032 + .117 + 2.57
      = (2.755 -- 2.909) seconds
```

For the response time the data processing I/O can be overlapped with the data processing CPU time. Then the response time is

FRES = (2.576 -- 2.73) seconds

The time in both cases is overwhelmed data processing CPU time.

5.3.4.3.2. Large buffers

In a large buffer system, the time is

LWORK = OVCPU + OVIO + DPIO + DPCPU

The times are as defined for the "Fast INGRES" system, except the I/O time. In the best case, the relation is in the buffers and the time is 0. Otherwise, the presence of the buffers insures that the relation can be read a cylinder at a time. Therefore DPIO is:

DPIO = (0 -- (DAVAC + 97 * DRATE))

The total time is therefore:

LWORK = (.006 -- .16) + 0 + (0-- (.03 + .116)
+ 1.64
= (1.65 -- 1.95) seconds

The data processing I/O time can be overlapped with the CPU time in the worst case. The response time is therefore:

LRES = (1.65 -- 1.81) seconds

The work and response times are dominated by the CPU time.

5.3.4.3.3. Associative Disks and CAFS

The work in the associative disk system is:

$$AWORK = OVCPU + OVIO + DAVAC + (n' + 1)*DROT + (n' + 1)*BCOM + HSORT + HDP + DTRANS$$

Since associative disks and CAFS do not directly support aggregate functions, the host processor must determine the proper algorithm to use. In this case, a good algorithm is to retrieve all the (acct,fund) pairs from the GMASTER relation and sort them to remove duplicates. Then the host will issue a query for each unique value found:

```
Retrieve (GMASTER.encumb) where acct = value
        and fund = value
```

The host processor must then accumulate the sum, and print it, with the values of the acct and fund attributes, on the user's terminal.

The parameter n' is the number of unique values in the (acct, fund) pair. There were 17 unique values in this query. Therefore the number of times the relation is read is 18, the 17 unique values and the 1 to read it to perform the projection. There are an equal number of communications to the backend system.

HSORT, the time to sort the projection in the host, will now be calculated.

In this example, there were 194 tuples in the source relation, and the two attributes are 10 bytes each (both are carried as character fields since numbers such as "109A" are allowed). There are therefore (194*20)/512, or 8 pages, to be sorted. This is was measured at .22 seconds of CPU time on a PDP 11/70.

Therefore, HSORT = .22 seconds

In the following section, DTRANS is calculated.

The total amount of data transferred by the 17 scans is precisely the size of the relation, since whole tuples are transferred and each tuple belongs to exactly one (acct, fund) pair. The first scan also transferred the entire relation, so the projection could be formed.

Therefore DTRANS = 2*97*DRATE = 194*DRATE

HDP will now be calculated.

The work that the host processor must do, independent of the sort, is

1) form the projection on GMASTER so the (acct,fund) pairs can be sorted.

2) As the matching tuples are sent to the host from each of the 17 sub-queries, it must add the "encumb" attribute, and, when the sub-query is finished, print the result on the user's terminal. In query L1 the CPU time was calculated to be 1.6 seconds, or .0018 seconds per tuple. The amount of work that must be done by the host processor for the

associative disk for query A1 is on the order of the amount of work INGRES does per tuple for query L1. Therefore the DHP time is:

DHP = 2 * 194 * .0018 = .70 seconds

The total time to process this query is therefore:

AWORK = OVCPU + OVIO + DAVAC + (n' + 1)*DROT
+ (n' + 1)*BCOM + HSORT + HDP + DTRANS

= (.006--.16) + .03 + .03 + 18*.0167
+18(.006 -- .03) +.22 +.70 + 194*.0012

= (1.628 -- 2.212) seconds

5.3.4.3.4. CASSM

The time in CASSM is

CWORK = OVIO + OVCPU + BCOM + DAVAC + n*DROT + HDP
+ DTRANS

The number of rotations, n, shall be determined below.

CASSM includes as a part of its cell processors the function of summing attribute values. The following is an algorithm that will result in the processing of the entire query in the backend machine.

1) find the first value of acct, and set one of its mark bits to show it is participating in this sum. Mark every tuple that contains a pointer to that value of acct. This requires one pass over the data.

2) Find the first value of fund for the marked tuples; set another mark bit to show these tuples are to be summed in the next pass. This requires one pass over the data.

3) Add the encumb amount to the sum area in the CASSM cell processors for each doubly marked tuple, and mark the tuple in another mark bit to show that it is removed from the algorithm. This requires one pass.

4) Now find the next unmarked value of the fund field for this value of acct; mark, sum as in 2 and 3.

When all qualifying values of fund have been found, repeat from step (1) until there are no more values of acct.

In this particular query, the results are skewed because there was only one value for acct, and 17 values for fund. Therefore the algorithm takes 1 pass to mark all tuples for the one acct, then 17 passes for the 17 unique values of the fund field. It is assumed that the overlapping of the output of an attribute value and the testing of another attribute can be used, so the total number of cycles to produce the results would be 20 cycles. This includes an extra, unoverlapped cycle at the end to output the last sum, and the first cycle to mark the records of the GMASTER relation. The value of n is therefore 20.

The time to transfer the 17 sums to the host machine, DTRANS, is minimal and may be neglected.

The time in the host to format the sums for printing is also

minimal and shall be neglected.

The total work of this query is therefore:

```
CWORK = OVIO + OVCPU + BCOM + DAVAC + n*DROT + HDP
                + DTRANS
      = .03 + (.006--.16) + (.006--.03) + 20*.0167
      = (.376 -- .554)
```

5.3.4.3.5. DBC

The time in DBC is:

```
DBWORK = OVCPU + OVIO + (n' + 1)*BCOM + DCYL + DAVAC
          + (n' + 1)*DROT + DBSORT
               + DTRANS + DHP
```

Since DBC has no internal mechanism for handling aggregate functions, the host processor must determine the best algorithm to use. The Security Filter Processor contains a sort mechanism, so when the host processor issues the request:

(Retrieve (acct, fund)(relation = GMASTER)

sorted by (acct,fund))

the 17 unique values of (fund,acct) will be returned to the host. Then 17 sub-queries can be issued to DBC:

(Retrieve(sum(encumb))(relation = GMASTER)

&(acct=value)&(fund=value))

The number of communications with DBC and the number of rotations is therefore = 18.

The value for DBSORT will now be calculated.

It cost .24 sec. of CPU time in an 11/70 to perform the sort of the (acct,fund) attributes; that will be considered the

best case time. If the SFP is a slower, cheaper processor such as an 11/40 the sort time would be longer. An 11/40 is 1/3 the speed of an 11/70; so the worst case time is .72 seconds.

DBSORT = (.24 -- .72) seconds

The time to transfer the 17 sums from the controlling processor to the host processor, and the time to format them for printing, is very small and will be neglected.

The total DBC time is therefore:

```
DBWORK = OVCPU + OVIO + (n' + 1)*BCOM + DCYL + DAVAC
           + (n' + 1)*DROT + DBSORT
       = (.006--.16) + .03 + 18*(.006--.03) + .006 + .030
              + 18*(.0167) + (.24--.72) seconds

       = (.72 -- 1.78) seconds
```

5.3.4.3.6. DIRECT

The design of DIRECT includes an 11/40 as the processor that controls the cell processors. It can be assumed that the 11/40 is programmed to perform aggregate functions. The best algorithm for it to use for this query is to:

1) form the sorted projection of the (fund, acct) attributes

2) issue the command to the cell processors:

```
if acct = value1 and fund = value2
{
        sum = sum + encumb
}
```

(2) is issued for each of the 17 unique values of the (acct,

fund) pair. Using the above algorithm, the work in DIRECT is:

DWORK = OVCPU + OVIO + BCOM + n*CSCAN + DPIO + DPSORT

OVCPU, OVIO, and BCOM are the host times as discussed in prior queries. DPSORT is the time to perform the sorted projection in the 11/40 in the backend. DPIO is the time to read the relation into the cell processors, and n is the number of cell processor scans required.

DPSORT will first be determined.

Since the cell processors will return to the 11/40 the attributes required, the only work the 11/40 must perform is to sort the attribute pairs returned to it. This time was determined in 5.3.4.2.4 as .72 seconds.

The number of scans required (n) is determined in the following paragraph.

1) formation of the sorted projection
The relation fits in 3 of DIRECT's pages, so the initial projection requires 3 scan times: one to write the attributes to temporary relation pages, one to compact those pages into a single page, and one to pass that page to the host.

2) performing the 17 sub-queries
The cell processors are not constrained to execute the same

query at the same time. Therefore, each of the cell processors can be performing a different sub-query. pair. No temporary relation need be written as a result of the summing operation; each query processor can keep its own sum and pass it back to the 11/40 when it has read the last page of the relation. There are therefore 8 processors available to execute 8 sub-queries at once. Each processor must scan the 3 pages of the relation. The data-page scans can operate in parallel, 8 processors at a time. Therefore the number of scans for the sub-queries is:

```
3   1st 8 sub-queries
3   2nd 8 sub-queries
1   last sub-query can be performed by 3 processors
        at once
--
7   total scans, sub-query execution
```

Therefore, n, the total number of scans required, is 8.

The total time in DIRECT is therefore:

```
DWORK = OVCPU + OVIO + BCOM + n*CSCAN +
        (0 -- DAVAC + 97 * DRATE) + DPSORT
      = (.006 -- .16) + .03 + (.006 -- .03) + 8*.12
                + ( 0 -- .03 + 97 * .0012) + .72

      = (1.722 -- 2.046)
```

The response time is the same.

5.3.4.3.7. RAP

RAP does not directly perform aggregate functions. It will be assumed that the same algorithm is used to perform the

aggregate function in RAP as in DIRECT. Since RAP does not have a general-purpose computer as a part of the back-end, the sorted projection must be performed in the host. Therefore the work in RAP is:

RWORK = OVCPU + OVIO + (n'+1) *BCOM + HSORT + m*CSCAN
+ DPIO

n', the number of unique values in the sorted projection, is 17.

HSORT, the time required in the host to sort the attributes, was calculated in 5.3.4.3.4 as .24 seconds.

The number of data cell scans (m) will be determined in the following paragraph.

1) forming the projection

It is assumed that RAP will perform a projection by simply passing the attributes required to the host. This requires one scan time, since all 8 processors can perform the scan at the same time.

2) performing the sub-queries

Each of the 8 processors is associated with one page of data. All must perform the same query at the same time. Since the relation fits on 3 data cells, only 3 processors at once can be in use. The query requires:

```
17 scans, one per subquery, to mark quaifying tuples
17 scans, one per subquery, to perform sum
--
34 scans
```

The total number of scans is therefore 35.

The total work is:

```
RWORK = OVCPU + OVIO + (n'+1) *BCOM + HSORT + m*CSCAN
                + DPIO
      = (.006 -- .16) + .03 + 18* (.006 -- .03) + .24
          +35 * .012 + ( 0 -- .030 + 97 * .0012)

      = (.804 -- 1.54)
```

The response time is the same.

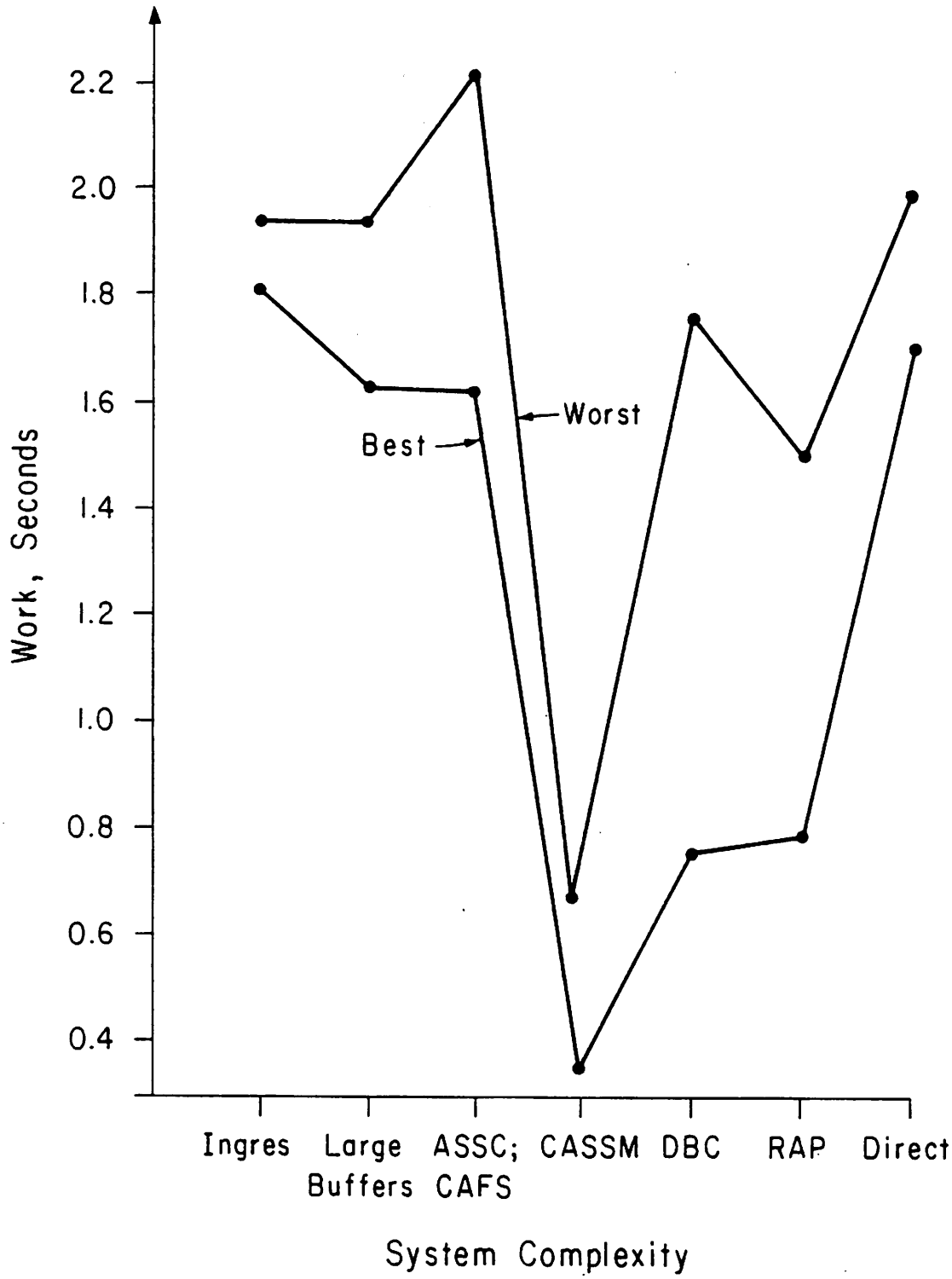


Figure 5.11 Total Work vs. System Complexity
Query L1

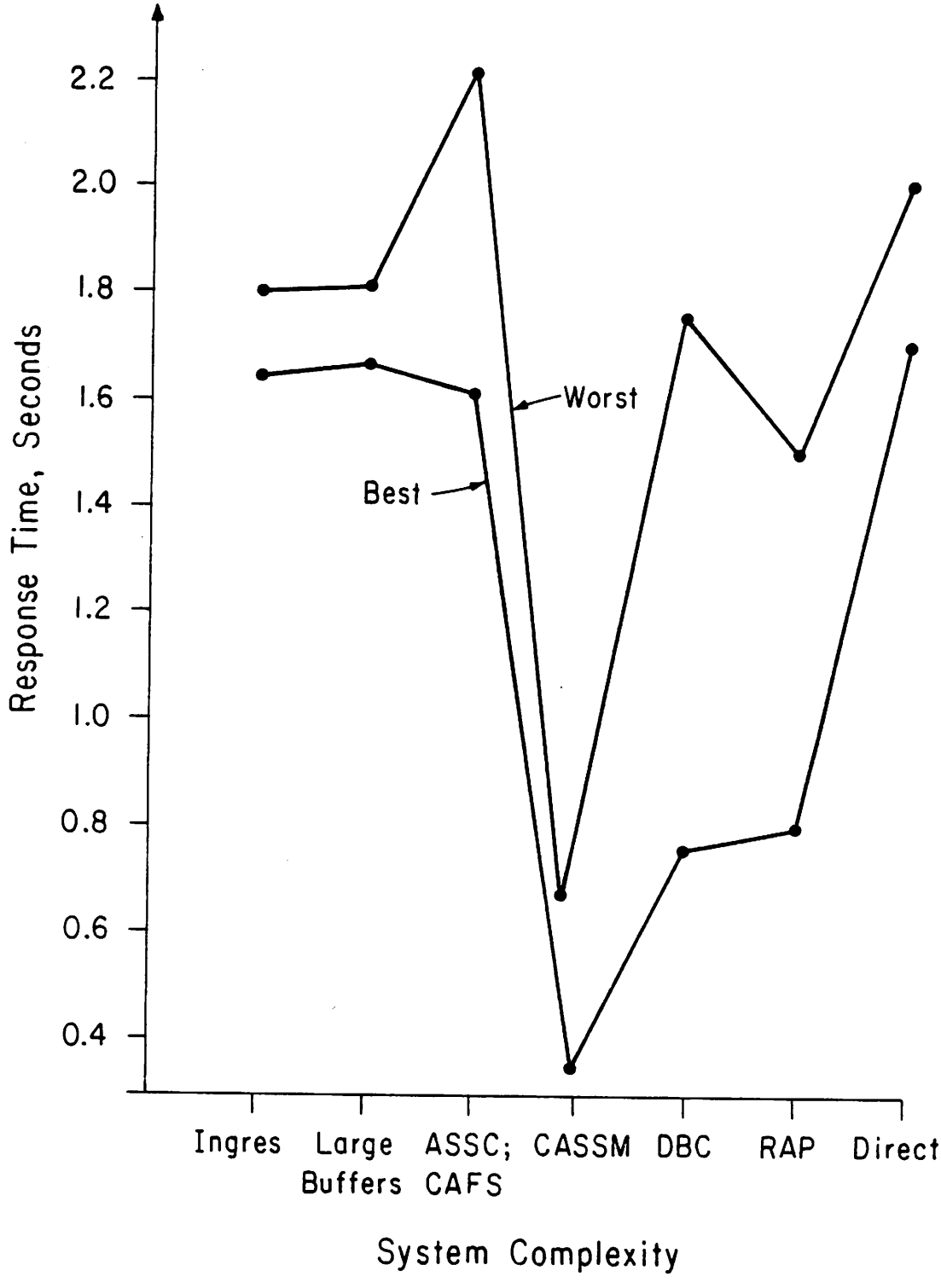


Figure 5.12 Response time vs. Complexity
Query A1

5.3.4.3.8. Conclusion

In Figures 5.11 and 5.12 it is apparent that the best improvements over a standard "Fast INGRES" system are seen in the DBC and CASSM machines. RAP and DIRECT are limited by the necessity of reading the data from the disk, serially, into the CCD pages. When that limitation is removed in the "best case" figure for the two systems, the time is still not extremely improved over an associative disk system. The reason that the improvement is so small is that the query requires a CPU-bound sort of one of the relations. The response times and work for this query are significantly improved over the standard system; however, the improvement is never more than 75%.

5.3.4.4. Multi-relation Queries

The multi-relation query chosen is:

Query M1:

```
retrieve ( ROOMS.building, ROOMS.roomnum, ROOMS.capacity,
                COURSE.day, COURSE.hour)
        where ROOMS.roomnum = COURSE.roomnum
                and ROOMS.building = COURSE.building
                and ROOMS.type = "lab"
```

This query is the "rooms" query from Chapter 4; as explained in that chapter, the relation "COURSE" contains information about all the courses taught by the UC Berkeley EECS Department in the last four years. It contains 11436 tuples in 2858 pages, and is stored in an ISAM storage structure, keyed on instructor name and course number.

The relation "ROOMS" contains information about every room that the EECS Department can use for teaching courses. It contains 282 tuples in 29 pages, and is hashed on room number.

The result of this query is a list which contains the building, room number, capacity, day, and hour of the use of any lab for the last four years.

Considerations in the analysis of this query:

1) "ROOMS", since it is stored in 29 pages, can be stored on 2 tracks of one cylinder.

2) "COURSE" requires 130 tracks (7 cylinders) of disk space.

3) The query is a two-relation query; therefore it will be assumed that the overhead to for this query is double that of the single-relation queries. The overhead CPU time is therefore .012 seconds best case -- .32 seconds worst case; the overhead I/O is .06 seconds.

5.3.4.4.1. Fast INGRES

It is assumed that the "fast INGRES" system uses the following algorithm for this query:

1) Form R2, the sorted, restricted projection of "ROOMS", retaining R2 in an in-memory array.

2) Scan the "COURSE" relation, and compare the join fields of each tuple in COURSE to each tuple in R2.

3) Output the required attributes of any matching tuples.

R2 contains 22 tuples in 2 pages, and can easily be kept in memory.

Using this algorithm, the total work for fast INGRES is:

FWORK = OVCPU + OVIO + RSORT + CCOMP + CDP + DPIO

OVCPU and OVIO are as defined above.

RSORT, the time to perform the sorted, restricted projection of ROOMS, will be determined first.

The work that must be performed to create R2 is:

1) Read the entire ROOMS relation, testing each tuple to determine if ROOMS.type = "lab"

2) For those tuples that are the correct type, write the building, roomnum, and capacity attributes to an in-memory array (R2)

3) Sort R2 on "roomnum, building" and remove duplicates.

The CPU time to perform the above is composed of the time to sort R2, which is .06 seconds on the 11/70, and the time to read ROOMS, which is equal to .006 * 29 = .174 seconds.

Therefore RSORT = .23 seconds.

CCOMP, the time to compare the (roomnum, building) attributes in COURSE to those in R2, will be determined in the

following section.

There are 22 tuples in R2, and 11436 in COURSE. Since R2 is sorted, each of the COURSE tuples need be compared only until a "greater than" result is found. The (roomnum, building) attribute pair is 20 characters long. Therefore the average number of comparisons that must be made per tuple in COURSE is 20 * 11, or 220. Each character comparison requires at least 3 instructions: the comparison, the test for condition, and te loop control. Therefore, there are a minimum of 660 instructions for each of the 11436 tuples in COURSE. This is about 7.5 million instructions; on the 11/70, which is a 1 MIP machine, it will require 7.5 seconds. This is a minimum time; it does not include the time to split the tuple into attributes, do type conversion, etc. The worst case time, however, should be no more than 15.0 seconds.

CDP, the time CPU time required to read the data, is calculated next. There are a total of 2887 pages read; the UNIX CPU time required to read a page is .006 seconds. Therefore CDP = 17.322 seconds.

DPIO, the I/O time to read the data into memory, is calculated in the following paragraph.

1) ROOMS

Since the ROOMS relation is on two tracks of the same

cylinder, the time to read it is:

RIO = DAVAC + ROT/2 + 29 * DRATE

where DROT/2 = the latency required to access the second track.

2) COURSE

The I/O time to read COURSE is

CIO = DAVAC + 6*CYAVAC + (130 - 7) * DROT/2 + 2858*DRATE

DAVAC is the average access time to begin reading the first cylinder. CYAVAC is the time to move the disk head from one cylinder to an adjacent one; on the disk used on the INGRES system, that time is .010 seconds. As is usual on disk systems, the time to move the head one cylinder is a large proportion of the average access time because the time to overcome the head inertia is a large proportion of the average access time.

COURSE is stored in 130 tracks; the time to access the first track on each cylinder is included in the terms DAVAC and 6*CYAVAC; the time to access the remaining tracks is the average latency, DROT/2. Finally, the time to transfer the data to memory is 2858 pages * DRATE.

Therefore the total DPIO time is:

DPIO = DAVAC + ROT/2 + 29 * DRATE
+

DAVAC + 6*CYAVAC + (130 - 7) * DROT/2 + 2858*DRATE

= 2*DAVAC + 6*CYAVAC + 124*DROT/2 + 2887*DRATE
= 4.622 seconds

The total work to process this query is:

FWORK = OVCPU + OVIO + RSORT + CCOMP + CDP + DPIO
= (.012 -- .32) + .06 + .06 + (7.5 -- 15.0)
+17.332 +2*DAVAC + 6*CYAVAC + 124*DROT/2
+ 2887*DRATE

= (.012 -- .32) + (24.942 -- 32.442) +
.06 + .06 + 1.038 + 3.464

= (29.576 -- 37.384) seconds

The data processing I/O can be overlapped with RSORT, CCOMP, and CDP. Therefore the response time is

FRES = (.012 -- .32) + (24.942 -- 32.442)

= (24.954 -- 32.762) seconds

This time is dominated by the operating system time to read the pages into memory.

5.3.4.4.2. Large Buffers

The use of large buffers only impacts the I/O time: The minimal I/O time is 0, if all the relations are in memory. The least that a large buffer system will supply is the capability of reading an entire cylinder at a time.

The work in a large buffer system is therefore:

```
LWORK = OVCPU + OVIO + RSORT + CCOMP + CDP + DPIO
      = (.012 -- .32) + 0 + (24.882 -- 32.382)
                + (0 -- RCYL + CCYL)
```

RCYL, the time to read ROOMS a cylinder at a time, is

```
RCYL = DAVAC + 29 * DRATE
     = .065 seconds
```

CCYL, the time to read COURSE a cylinder at a time, is

```
CCYL = DAVAC + 6*CYAVAC + 2858*DRATE
     = .03 + 6*.0083 + 2858*.0012
     = 3.51 seconds
```

Therefore the total work to process the query is:

```
LWORK = (.012 -- .32) + 24.882 + (0 -- .065 + 3.51)
      = (24.894 -- 36.277) seconds
```

The response time is:

LRES = (.012 -- .32) + (24.882 -- 37.382)

since the I/O and CPU times can be overlapped; therefore it is

LRES = (24.894 -- 32.702) seconds

5.3.4.4.3. Associative Disks and DBC

It shall assumed that the following algorithm is used in processing query M1 in an associative disk system:

1) The sorted, restricted projection of ROOMS (R2) is formed in the host

2) The host issues 22 queries, one for each tuple in R2, to retrieve from the associative disk the tuples in "COURSE" with matching roomnum and building attributes as those in the query. These queries shall be denoted as the 22 subqueries, and are of the form

```
retrieve (COURSE.day, COURSE.hour)
        where COURSE.roomnum = value1
        and COURSE.building = value2
```

As matching tuples are returned, they are output.

The time to perform this query using the above algorithm is:

```
AWORK = OVCPU + OVIO + RSORT + (n'+1)*BCOM
                  + RPROC + n' * CPROC
```

OVCPU and OVIO have been defined above; RSORT, the time to perform the sorted, restricted projection of the ROOMS relation in the host, is the same as in section 5.3.4.4.1.

The number of communications required between the host and the associative disk (BCOM) is the number of subqueries (n' = 22) plus the one query to form the sorted, restricted projection.

RPROC, the time to process the ROOMS restricted projection, is:

```
RPROC = DAVAC + DROT
```

This is one average access time, plus the disk rotation time to perform the query.

The time to process each of the 22 subqueries, CPROC is:

CPROC = DAVAC + 6*CYLAVAC + 7*DROT

which is the time to access each of the cylinders, and to perform the subquery on each of them.

Therefore the total time is:

$$\begin{aligned} AWORK &= OVCPU + OVIO + RSORT + (n'+1)*BCOM \\ &\quad + RPROC + n' * CPROC \\ &= (.012 -- .32) + .06 + .06 + 23*(.006 -- .03) \\ &\quad + .03 + .0167 + 22*(.207) \\ &= (4.87 -- 5.73) \text{ seconds} \end{aligned}$$

The time to perform the 22 subqueries (4.55 seconds) dominates both the best and worst case times.

DBC:

The key question in DBC is whether the search space in the 7 cylinders used to store the "COURSE" relation can be narrowed to one or two cylinders by the use of attribute indices. The DBC system is built around the idea that the attribute indices, which are stored in the Structure Memory, will narrow the search space. The "COURSE" relation was clustered on course number, which is not an attribute even mentioned in this query. Therefore, even if indices existed for each of the "COURSE" attributes in this query, they would not narrow the search space because, in the absence of any strong correlation between course number and the other attributes, it is to be expected that qualifying tuples will

occur on each of the 7 cylinders.

DBC handles single-relation queries only, and in [BAN78B] it was shown how joins are split into single-relation queries. Using the queries in [BAN78B] as a guide, we find that DBC will process the query M1 exactly as the associative disk does, and that the time remains the same as in the associative disk.

5.3.4.4.4. CASSM

CASSM uses the following algorithm to implement joins.

There are two bit-maps, one for each of the relations to be joined.

1) First, a pass is made over the smaller relation (ROOMS), and the join attributes (roomnum,building) of qualifying tuples used to hash to a bit map to mark the presence of a value.

2) A pass is made over the second relation (COURSE) and a second bit map is marked if the (roomnum,building) attribute pair hash to a location that is marked in the first bit map. Also, if both bit-maps are marked the tuple is itself marked for collection.

3) A second pass is made over the first relation, and those tuples whose (roomnum,building) pairs hash to marked locations in both bit-maps are marked for output.

The marked attributes are sent to the host processor to perform the actual join.

The time to execute this query in CASSM is:

CWORK = OVCPU + OVIO + BCOM + HJOIN + n*RPROC + m*CPROC

HJOIN is the amount of time required in the host to perform the join. RPROC is the time required to scan "ROOMS". CPROC is the time to scan COURSE.

First, HJOIN will be computed.

Using the above algorithm, 22 tuples of "ROOMS" and 422 tuples of "COURSE" are marked in both bit maps. Only the necessary attributes from those tuples are sent to the host. Those attributes require a total of 47 pages of storage. With such a small number of pages it is entirely possible that the host can complete the processing of this query by using an in-memory sort of the qualified tuples of the two relations, then a linear search to match the correct tuples. The CPU time to sort the 47 pages is 14.1 seconds on the 11/70; therefore HJOIN = 14.1 seconds.

Next the number of scans of the rooms relation (n) will be computed.

1	scan to mark bit map
1	scan to check COURSE's bit map and mark tuple
4	scans to return 4 attributes
6	n, total scans required of the ROOMS relation

The number of scans of the COURSE relation (m) is:

```
1 scan to mark map and mark qualifying tuples
4 scans to send attributes to host
--
5        m, the required number of scans of COURSE
```

The total time to do this query is, therefore:

```
CWORK = OVCPU + OVIO + BCOM + HJOIN + n*RPROC + m*CPROC
      = (.012 -- .32) + .06 + (.006 -- .03) + 14.1 +
                6*.047 + 5*.207

      = (15.50 -- 15.83) seconds
```

The best and worst case times are dominated by the time to perform the join in the host.

5.3.4.4.5. DIRECT

It is assumed that the algorithm used in the back-end 11/40 is the same as that used by the associative disk system. That is:

1) form sorted, restricted projection of "ROOMS", holding it in memory.

2) perform 22 sub-queries

Using the above algorithm, the work is:

DWORK = OVCPU + OVIO + BCOM + RSORT + r*CSCANS + c*CSCANS + DPIO

The parameter r is the number of CCD page scans required to form the sorted, restricted projection of ROOMS. The

parameter c is the number of scans required to process the 22 sub-queries. RSORT is the time required in the host to perform the restricted, sorted projection.

First, r will be calculated.

The entire "ROOMS" relation will fit on one CCD cell. Therefore r is:

```
1 scan to read "ROOMS", writing qualified tuples
        to temp
1 scan to compact temp
1 scan to send tuples to controlling processor
---
3   r, number scans required
```

Next, c will be calculated.

The 2858 pages of "COURSE" will fit on 33 CCD cells. The 22 queries can be performed in parallel, 8 at a time, for a total of three scans per page of the "COURSE" relation. Therefore there are 33*3 = 99 scans required to produce the joined tuples. An additional scan is required to compress the data, and another to send it to the host, for a total of 101 scans.

Therefore, c = 101.

DPIO is calculated next.

The time to read the ROOMS relation, if it is not in memory, is:

RPROC = DAVAC + 29 * DRATE

The entire COURSE relation will not fit within the CCD cache. The entire relation will therefore have to be read 3 times, once for each scan. The time to perform the three reads is:

CPROC = DAVAC + 6*CYAVAC + 2825*DRATE +2*(7*CYAVAC + 2858*DRATE)

The above formula assumes that the average access time must be used for only the initial access to the relation; the last two scans will be for adjacent cylinders.

The time to perform the sorted, restricted projection of ROOMS is denoted by RSORT. That time was estimated at .06 seconds for an 11/70. The 11/40 is one-third the speed of an 11/70, so the time for RSORT is .18 seconds.

The total time is therefore:

DWORK = OVCPU + OVIO + BCOM + RSORT + r*CSCANS
+ c*CSCANS + DAVAC + 29 * DRATE
+ DAVAC + 20*CYAVAC + 8574*DRATE

= (.012 -- .32) + .06 + (.006 -- .03) + .18
+ 3*.012 + 101*.012 + .03 + .034
+ .03 + .20 + 10.289

= (12.089 -- 12.421) seconds

The I/O and the CCD scan times can be overlapped, so the response time is:

DRES = (10.841 -- 11.173) seconds

The response and total work times of this query are totally

dominated by the time to read the COURSE relation three times (10.51 seconds), which was necessary because the cache was not large enough to store the entire relation. An addition of 9 CCD pages would have alleviated this problem. The figure is 9, not 1, because 8 of the cells must be used at all times as output cells. Therefore the assumption will be made that in the best case there are 9 more data cells available, and that the best case times are therefore

DWORK = 4.189 seconds
DRES = 2.94 seconds

5.3.4.4.6. RAP

The best algorithm for RAP to use in executing query M1 is the algorithm used by DIRECT. However, the RAP backend system does not include a general-purpose computer. Therefore, the sorted, restricted projection of ROOMS must be performed in the host, and the 22 sub-queries issued by the host.

Therefore, the time in RAP is:

RWORK = OVCPU + OVIO + RSORT + (n'+1)*BCOM +
r*CSCAN + n'*(c * SCAN) + DPIO

RSORT, the time to perform the sorted, restricted projection, was calculated above as .06 seconds.

The number of cell scans to produce the restricted projection of ROOMS (r) is:

1 scan to mark output

1 scan to return tuples to the host

---

2 r, number of scans for ROOMS

The number of scans for each sub-query on COURSE (c) is calculated in the following section.

Each of the cell processors will execute the same query, but on a different page of data. Therefore approximately 4 cell scans are required for each of the 22 sub-queries to mark the tuples, and an additional 4 scans to return the marked tuples to the host. Therefore, c = 8.

DPIO, the data processing I/O time, is discussed in the following section.

The entire COURSE relation cannot fit in the cache at once; therefore, the relation will have to be read twice for each sub-query. It must be read once to mark the tuples, and once to return the marked tuples to the host. Since RAP maintains the mark-bits in a separate RAM bit map, the relation does not have to be written before it is read. Therefore the I/O time is:

```
DPIO = RPROC + CPROC
     = DAVAC + 29*DRATE
        + DAVAC + 6*(CYAVAC) + 2858*DRATE
                  (1st read of COURSE)

        + 43* (7*CYAVAC + 2858*DRATE)
                  (43 remaining reads)

     = 2*DAVAC + 307*CYAVAC + 125791*DRATE
```

The total work is therefore:

```
RWORK = OVCPU + OVIO + RSORT + (n'+1)*BCOM +
                r*CSCAN + n'*(c * SCAN) + DPIO

      = (.012 -- .32) + .06 + .06 + 23*(.006 -- .03)
                +3*.012 + 22*8*.012 + 154.08
      = (2.418 -- 3.278) + 154.08 seconds

      = (156.49 -- 157.36) seconds
```

As in DIRECT, the time is dominated by the I/O time. If RAP contained a CCD cache of at least 33 data cells, 43 of the reads would have been avoided. Then the time would be:

RWORK = (6.584 -- 6.862) seconds

5.3.4.4.7. CAFS

CAFS handles joins in the same way as CASSM except that it does not mark qualified tuples, but passes them on to the host. To review the algorithm:

There are two bit-maps, one for each of the relations to be joined.

1) First, a pass is made over the smaller relation (ROOMS), and the join attributes (roomnum,building) of qualifying tuples used to hash to a bit map to mark the presence of a value.

2) A pass is made over the second relation (COURSE) and a second bit map is marked if the (roomnum,building) attribute pair hash to a location that is marked in the first bit map. Also, if both bit-maps are marked the tuple is itself sent

to the host.

3) A second pass is made over the first relation, and those tuples whose (roomnum,building) pairs hash to marked locations in both bit-maps are sent to the host.

The host must perform the actual join on the smaller relations passed to it.

The time to process query M1 in CAFS is therefore:

CAWORK = OVCPU + OVIO + BCOM + HJOIN + 2*RPROC + CPROC

The time to perform the join in the host, HJOIN, was discussed in the section on CASSM, and found to be 14.1 seconds.

RPROC, the time to scan the ROOMS relation and mark the first bit map, is

RPROC = DAVAC + DROT

CRPOC, the time to scan the COURSE relation, is:

CPROC = DAVAC + 6*CYAVAC + 7*DROT

The total time is therefore:

CAWORK = (14.48 -- 14.81) seconds

and the response time is the same.

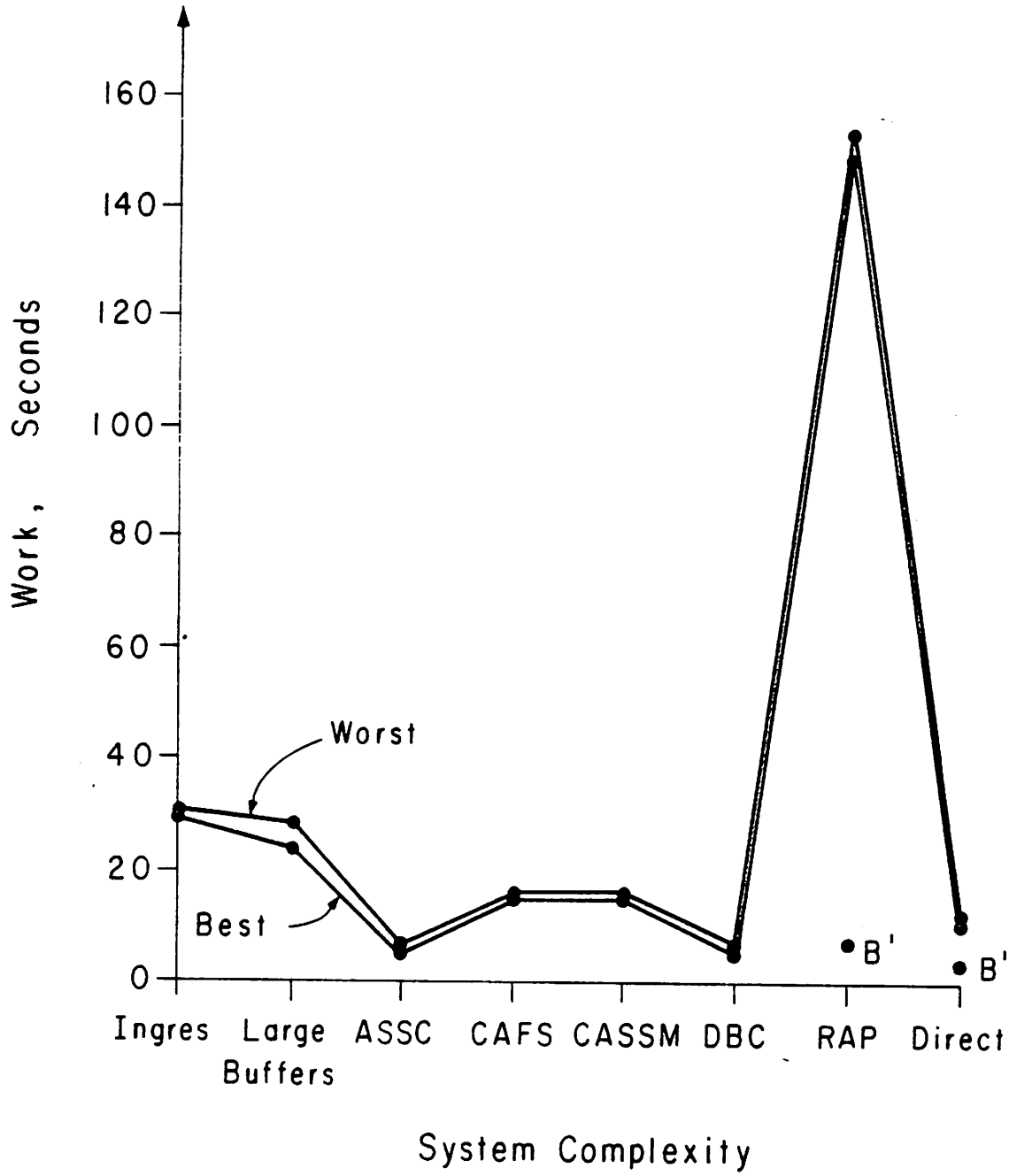


Figure 5.13 Total Work vs. System Complexity
Query M1

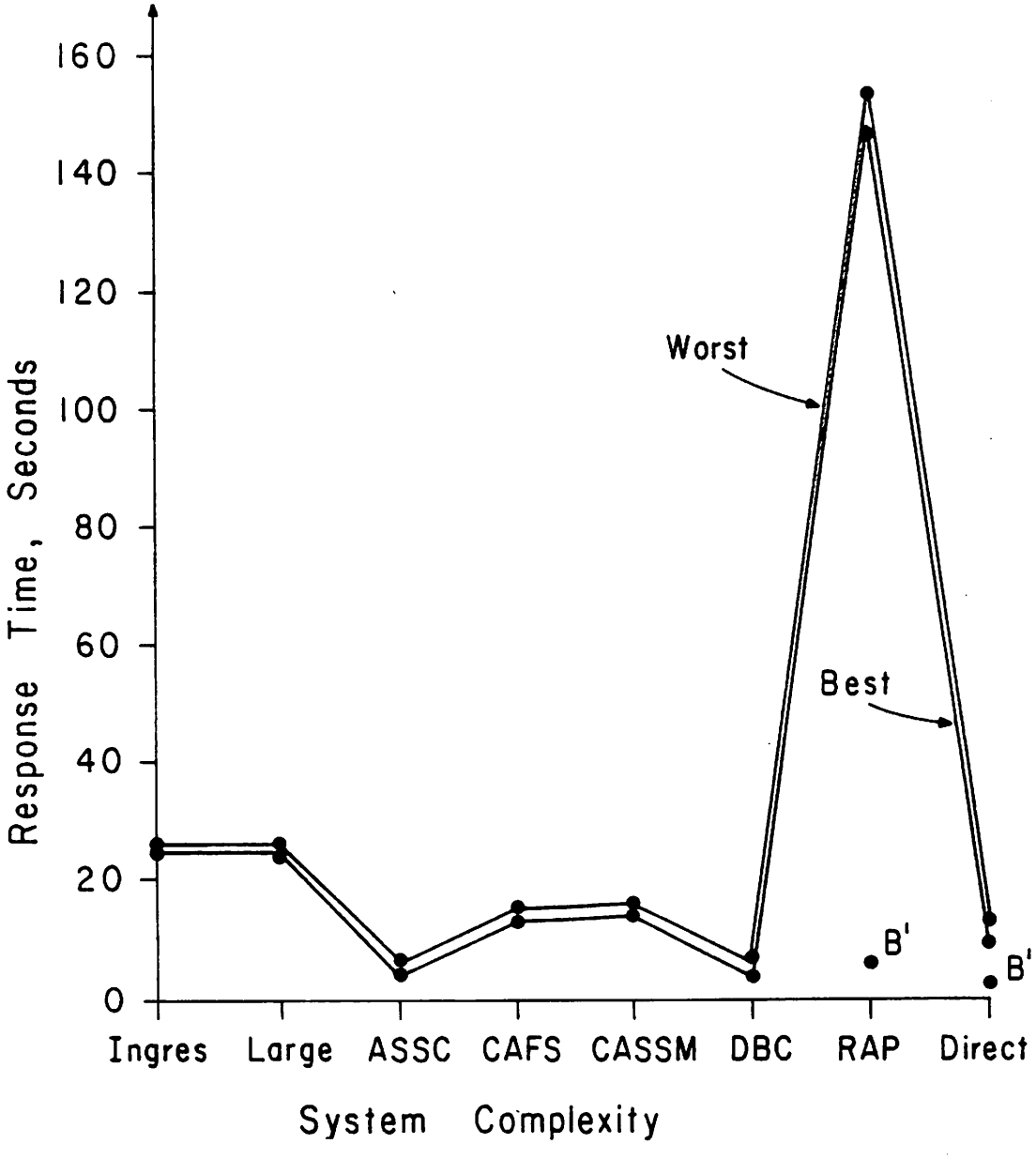


Figure 5.14 Response time vs. Complexity
Query M1

5.3.4.4.8. Conclusion

As shown in Figures 5.13 and 5.14, the best response and total work times are in the associative disk and DBC systems. That is because they process, in parallel, the data as it comes from the disk. The CAFS and CASSM systems are handicapped by the fact that they must perform the join in the host.

DIRECT performs better than INGRES, by almost an order of magnitude less response time, when the cache is large enough to hold the entire relation. Otherwise, it operates at only half the speed of INGRES. Considering the tremendous increase of complexity of DIRECT over a standard system, halving the response time does not appear to be significant.

The response time of RAP is very sensitive to whether the relation can be completely stored in the data cells. That is because RAP depends on re-scanning the relation, testing for mark-bits, to output tuples to the host. The best case RAP time assumes there are enough data cells to store the entire relation, and in that case the response time drops to less than a third of the response time of INGRES.

5.4. Conclusion

The purpose of this chapter has been to gain insight into the possibility of the use of database machines. In section 5.4.1 the performance of the database machines is compared

to the performance of the database system on a general purpose machine. Section 5.4.2 compares the performances of the machines. Section 5.4.3 analyzes the use of extended storage systems.

5.4.1. Comparison to a standard system

The two query types previously defined are data-intensive and overhead-intensive queries. It has been shown that database machines are not cost-effective if the application supported is mainly overhead-intensive queries.

It was shown that data-intensive queries can be performed very efficiently on database machines if the function performed on the data is a function the database machine provides. For instance, if the function is a simple test for equality, the database machine can perform the query entirely in the backend system, thus causing a gain in the system's performance. However, if the queries are such that the function on the data is one that the database machine does not provide, as in the function of printing the data in query L1, the host processor is heavily impacted and the database machine causes little gain in the system's performance.

Multi-relation queries were performed better on some machines and worse on others depending on the join algorithm used.

5.4.2. Machine comparison

It will be recalled from section 5.2.2.7 that the points of comparison of the database machines were the use of mark bits, temporary relations, transferring entire tuples to the host, and the join algorithms. These shall be discussed, in turn, in the following paragraphs.

The use of mark-bits in RAP clearly hampers the performance of the system. RAP consistently required more time than DIRECT to perform the same function, and the extra time was time required to re-scan the cells.

The use of temporary relations in DIRECT was the alternate method to the use of mark-bits. Because DIRECT forms temporary relations, it performed better than RAP.

The effect of the transfer of partial tuples to the host is not clear. There is some advantage, but the precise advantage appears to be entirely dependent on the speed of the data management system in the host.

The particular join algorithm used effected the total performance of the systems in that some algorithms required more work in the host systems than others did. RAP, DIRECT, and DBC required very little host time. CAFS and the associative disk systems required much more.

5.4.3. Extended Storage

A surprising result was that the use of large buffers alone did not significantly affect the performance of the system. This is because the system is CPU bound, even in the optimal "Fast INGRES" that was assumed in this chapter. Only when the large buffer systems are combined with fast processors can performance improvements be made.

6. Conclusion

The focus of this thesis has been the performance of one data management system and possible methods to improve the performance. These methods have involved the use of advanced computer architecture designs. The specific architectures under consideration were multiple processors and extended storage systems. These will be discussed in turn. Then recommendations will be made for future studies.

6.1. The use of extended storage systems

It was found that the data-intensive queries exhibited a high degree of sequentiality. Therefore using large buffers to store read-ahead blocks can reduce the I/O times of data-intensive queries. However, the use of the buffers requires a level of communication with the operating system that is not currently available. The data management system must be able to indicate when a relation is being read sequentially, so the read-ahead can take place.

The overhead-intensive queries exhibit locality of access to system relations, so storing the system relations in buffers will reduce the response times of those queries. INGRES is an interpretative system, and therefore does more run-time checking than systems that pre-compile queries. Nevertheless, even those systems that pre-compile queries perform

some validity checking of the query and would benefit from caching the validity information.

The caching of system relations will not affect the performance of data-intensive queries because their response times are dominated by the time to process the data, not by the overhead. In the cases studied in this analysis, the system was CPU bound most of the time. Therefore, the use of extended storage devices must be combined with methods to decrease the CPU processing time to increase in the performance of the system.

6.2. Parallel Processing

The use of multiple processors in the form of intelligent terminals was shown to be optimal only for overhead-intensive queries. Intelligent terminals and special-purpose processors to parse and validity check the data are best used for those queries where the overhead is a significant portion of the query response time. The use of such processors does not noticeably improve the response time of data-intensive queries because most of their processing is done at the data level.

Database machines were explored. It was found that in several cases the time to execute a query in a database machine was longer than in a standard system. This was because the database machines are designed to be relatively

simple, and do not necessarily execute optimal algorithms. It was shown that those machines that operate on the data at the disk level (DBC, CASSM, associative disks) tend to have better performance than the caching systems (DIRECT, RAP) unless the data is already fully within the cache. If it is, DIRECT has the best performance of any of the machines, because of its cross-point switch.

6.3. Future work

We have, in this analysis, concentrated on the simple, common queries. An interesting extension is to analyze more complicated queries, such as those that involve secondary indices. The analysis of multi-user situations would be very helpful in comparing database machine architectures, since the machines may exhibit different performance characteristics when they are multi-programmed.

The use of different buffer replacement algorithms should also be studied. The LRU algorithm may not always be optimal, since the data pages are only referenced once but, under LRU, may push the system relation pages out of the buffers.

It would be interesting to develop different analytic or simulation models for the different query types. These models could then be calibrated with the results of this analysis. Such models would be helpful in evaluating the

results of system changes.

In general, the actual measurement of all assumed variables would be most instructive. Especially in the case of the database machines and the "Fast INGRES" system, the actual speeds of the systems may affect the results produced here.

Appendix A. REFERENCES

[ALLM76] Allman, E., Stonebraker, M. and Held, G., "Embedding a Relational Data Sublanguage in a General Purpose Programming Language," Proc. Conference on Data: Abstraction, Definition, and Structure, FDT, vol 8, No 2, March 1976.

[BABB79] Babb, E., "Implementing a Relational Database by Means of Specialized Hardware", ACM Transactions on Database Systems, Vol. 4, No. 1, March 1979, pages 1-29.

[BAN78A] Banerjee, J., and D.K. Hsiao, "Performance Study of a Database Machine in Supporting Relational Databases," Proc. Fourth International Conf. on VLDB.

[BAN78B] Banerjee, J., and D.K. Hsiao, "The Use of a 'Non-Relational' Database Machine in Supporting Relational Databases," Proc. Fourth Workshop on Computer Architecture for Non-numeric Processing, Syracuse, Aug. 1978.

[BAUM76] Baum, Richard I., David K. Hsiao, and Krishnamurthi Kannan, "The Architecture of a Database Computer - Part I: Concepts and Capabilities," Technical Report OSU--CISRC-TR-76-1, Computer and Information Science Research Center, The Ohio State University, Columbus, Ohio (National Technical Information Service Number AD-A034 154)

[BLAS75] Blasgen, M.W. and Eswaren, K.P., "On the Evaluation of Queries in a Relational Data Base System," IBM Research Report RJ-1745, 4/76.

[BRIC77] Brice, R. S., and S. W. Sherman, "An Extension of the Performance of a Database Manager in a Virtual Memory System Using Partially Locked Virtual Buffers," ACM Transactions on Database Systems, Vol.2, No. 2, June 1977, P. 196-207

[CHU72] Chu, Wesley, and Holger Opderbeck, "The Page Fault Frequency Replacement Algorithm", Proc. FJCC 1972, pages 597-609.

[COUL72] Coulouris, G.F., J.M. Evans, and R.W. Mitchell, "Towards content Addressing in Data Bases", Computer Journal, Vol. 15, No. 2, 1972, pages 95-98.

[DATE75] Date, C. J., An Introduction to Data Base Systems, Addison-Wesley, Reading, Mass. 1975.

[DENN68] Denning, P. J., "The Working Set Model for Program Behavior," CACM, May, 1968, Vol. 11, No. 5, pp. 323-333.

[DESP78] Despain, A. M. and D. Patterson, "X-Tree: A Tree Structured Multi- Processor Computer Architecture," Proceedings, Fifth Annual IEEE Symposium on Computer Architecture, April 3-5,1978.

[DEWI78] Dewitt, D. J., "DIRECT - A Multiprocessor Organization for Supporting Relational Data Base Management Systems," Proc. Fifth Annual Symposium on Computer Architecture, 1978.

[DEWI79] Dewitt, D. J., "Query Execution in DIRECT", Proceedings, SIGMOD International Conference on the Management of Data, 1979, pages 13-22.

[EAST75] Easton, M. C., "Model for Interactive Data Base Reference String," IBM J. Res. Develop. 19, 550 (Nov. 1975).

[EAST77] Easton, M. C., "Model for Data Base Reference Strings Based on Behavior of Reference Clusters," IBM Reasearch Report # 6450, Watson Research Center, 3/11/77. also published in IBM J. Res. and Develop. 22, #2, 197 (Mar. 1978).

[EPST77] Epstein, R., "Creating and Maintaining a Database Using INGRES," Electronics Research Laboratory, University of California, Berkeley, Ca., Memo #M77-71.

[FERN78] Fernandez, E. B., Lang, T., and C. Wood, "Effect of Replacement Algorithms on a Paged Database System," IBM Journal of Research and Development, Vol. 22, No. 2, Mar. 1978, pp. 185-196.

[FRAN76] Franaszek, P. A., and B.T. Bennet, "Adaptive Vari-

ation of the Transfer Unit in a Storage Hierarchy", IBM Research Report RC 6310, Watson Research Center, 11/30/76. also published in IBM Jour. Res. and Develop. Vol. 22, #4, 405 (July, 1978).

[GOLD74] Goldberg,, R., and R. Hassinger, "The Double Paging Anomaly", Proc. AFIPS 1974 NCC, Vol. 43, AFIPS Press, Montvale, N. J., pp. 195-199.

[GRAY78] Gray, James, "Notes on Data Base Operating Systems," IBM Research Report RJ2188 (30001) 2/23/78.

[HSIA79] Hsiao, David K., "Data Base Machines Are Coming!", IEEE Computer, March 1979, pages 7-10.

[HSI76A] Hsiao, David K. and Krishnamurti Kannan, "The Architecture of a Database Computer - Part II: The Design of Structure Memory and itsd Related Processors," Technical Report OSU--CISRC-TR-76-2, Computer and Information Science Research Center, The Ohio State University, Columbus, Ohio (National Technical Information Service Number AD/A-035 178)

[HSI76B] Hsiao, David K. and Krishnamurti Kannan, "The Architecture of a Database Computer - Part III: The Design of the Mass Memory and its Related Components," Technical Report OSU--CISRC-TR-76-3, Computer and Information Science Research Center, The Ohio State University, Columbus, Ohio (National Technical Informa-

tion Service Number ADA-036 217)

[IMS360] "Information Management System/360, Version 2, General Information Manual," Form GH20-0765-3, IBM Corporation, Technical Pub. Dept.

[KANN78] Kannan, Krishnamurthi, "The Design of a Mass Memory for a Database Computer," Proc. Fifth Annual Symposium on Computer Architecture, Palo Alto, CA. April 1978.

[KNUT73] Knuth, Donald E., The Art of Computer Programming, Vol.3. Addison-Wesley, Reading, Mass., 1973.

[LANG77] Lang, Thomas, Christopher Wood and Fernandez, Eduardo f., "Database Buffer paging in Virtual Storage Systems," TODS, Vol. 2, No. 4, December, 1977.

[LANG78] Langdon, Glen G., "A Note on Associative Processors for Data Management," TODS, Vol. 3, NO. 2, June 1978, Pages 148 - 158.

[LAVE75] Lavenberg, S.S., and G.S. Shedler,"A Queueing Model of the DL/1 Component of IMS", IBM Research Report RJ 1561, April 21, 1975.

[LEWI73] Lewis, P.A.W., and G.S. Shedler, "Empirically Derived Micrmodels for Sequences of Page Exceptions", IBM Journal of Research and Development, March 1973, pages 86-100.

[LIN 76] Lin, S.C., D.C.P. Smith, and J.M. Smith, "The Design of a Rotating Associative Memory for Relational Database Applications," TODS vol. 1, No. 1, pages 53 - 75, Mar. 1976.

[LIPO78] Lipovski, G. J., "Architectural Features of CASSM: a Context Addressed Segement Sequential Memory", Proceedings, Fifth Annual IEEE Symposium on Computer Architecture, April, 1978. [MARC78] March, Salvatore, "Models of Storage Structures and the Design of Database Records based on a User Characterization," Ph.D. Dissentation, Cornell University, 1978.

[MCGR76] McGregor, D.R., R.G. Thompson, and W.N. Dawson, "High Performance Hardware for Database Systems," Systems for Large Data Bases, P.C. Lockemann and E.J. Neuhold, eds. North-Holland Publishing Co., 1976.

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

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

[RAGA76] Ragaz, Nicklaus, and Juan Rodriguez-Rosell,

"Empirical Studies of Storage Management in a Data Base System", IBM Research Report RJ 1843, 10/7/76.

[RODR73] Rodriguez-Rosell, Juan, "Locality in Data Base Systems", Report, Center for Computer and Information Sciences, Division of Applied Mathematics, Brown University, Providennce, R. I.

[RODR76] Rodriguez-Rosell, Juan, "Empirical Data Reference Behavior in Data Base Systems," Computer, Nov., 1976, Pages 9-13.

[RODR75] Rodriguez-Rosell, Juan and David Hildebrand, "A Framework for Evaluation of Data Base Systems", IBM Research Report RJ 1975, May 23, 1975.

[REIT76] Reiter, Allen, "A Study of Buffer Management Policies for Data Management Systems," Mathematics Research Center, University of Wisconsin-Madison, Technical Summary Report # 1619, March 1976.

[RITC74] Ritchie, D. M., and Thompson, K., "The UNIX Time-Sharing System," Commmunications ACM 17, 7 , July,1974.

[RITC78] Ritchie, D. M., "A Retrospective", Bell System Technical Journal, vol 57, number 6, part 2, July-August 1978, pp 1949-1969.

[SADO78] Sadowski, Paul J. and S.A. Schuster, "Exploiting Parallelism in a Relational Associative Processor",

Proc. Fourth Workshop on Computer Architecture for Non-numeric Processing, Syracuse, Aug. 1978.

[SCHU78] Schuster, S. A. et al, "RAP.2 - An Associative Processor for Data Bases", Proceedings, Fifth Annual IEEE Symposium on Computer Architecture, April, 1978.

[SHER76] Sherman, Stephen W., and Brice, B. W., "Performance of a Database Manager in a Virtual Memory System", ACM Transactions on Data Base Systems, Vol. 1, No. 4, Dec. 1976, Pages 317-343.

[SMIT76] Smith, Alan Jay, "Sequentiality and Prefetching in Data Base Systems," IBM Research Report RJ 1743, March 19, 1976.

[SLOT75] Slotnik, D.L. "Logic per Track Devices" in "Advances in Computers", Vol. 10., Frantz Alt, Ed., Academic Press, New York, 1970, pp 291 - 296.

[STON76] Stonebraker, M. et. al., "The Design and Implementation of INGRES," TODS, Vol 1, No. 3, September 1976.

[SU 75] Su, Stanley Y. W., and G. Jack Lipovski, "CASSM: A Cellular System for Very Large Data Bases", Proceedings of the VLDB, 1975, pages 456 - 472.

[SU79] Su, Stanley Y. W., "Cellular-Logic Devices: Concepts and Applications", IEEE Computer, March 1979, pages 11-28.

[THOR70] Thornton, J. E., "Design of a Computer The Control Data 6600", Scott Foresman and Co, Glenview Ill., 1970.

[TUEL75] Tuel, W.G., and Juan Rodriguez-Rosell, "A Methodology for Evaluation of Data Base Systems", IBM Research Report RJ 1668, Oct. 15, 1975.

[TUEL76] Tuel, W. G., "An analysis of Buffer Paging in Virtual Storage Systems," IBM Journal of Research and Development, Vol. 20, No.5, September 1976.

[WONG76] Wong, E. and Youssefi, K., "Decomposition - A Strategy for Query Processing," TODS, Vol. 1, No. 3, September 1976.

[YAO 78] Yao, S.B.,DeJong, D., "Evaluation of Database Access Paths," Proceedings, SIGMOD International Conference on the Management of Data, 1978.

[YOUS78] Youssefi, Karel A., "Query Processing for a Relational Database System," Electronics Research Laboratory, University of California, Berkeley, Ca., Memo #M78-3.