THE USE OF TECHNOLOGICAL ADVANCES TO ENHANCE DATA MANAGEMENT SYSTEM PERFORMANCE

by

P. Hawthorne and M. Stonebraker

Memorandum No. UCB/ERL M79/3

15 January 1979

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

The Use of Technological Advances to Enhance Data Management System Performance

by

Paula Hawthorn and Michael Stonebraker

Electronics Research Laboratory
University of California, Berkeley

ABSTRACT

The effect on the performance of data management systems of the use of extended storage devices, multiple processors and prefetching data blocks is analyzed with respect to one system, INGRES. Benchmark query streams, derived from user queries, were run on the INGRES system and their CPU usage and data reference patterns traced. 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. It is shown that the random access model of data references holds only for overhead-intensive queries, and then only if references to system catalogs are not considered data references. 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. A third benchmark set, multi-relation queries, was devised, and proposals are made for taking advantage of the locality of reference which was found.

Keywords: relational data management systems, buffering techniques, distributed processing, performance evaluation

---

This research was sponsored at the University of California at Berkeley by the National Science Foundation grant MCS75-03839-A01 and at the Lawrence Berkeley Laboratory by the Basic Energy Sciences Division of the Department of Energy.

Authors' addresses: P. Hawthorn, Computer Science and Applied Mathematics Department, Lawrence Berkeley Lab, Berkeley, CA. 94720; M. Stonebraker, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720.

CR categories: 3.70, 3.72, 4.41, 6.20

Section I. Introduction

A. Background

Two technological advances that may have substantial impact on data management system performance are low cost, fast processors and extended storage devices such as CCDs, bubbles and semiconductor memories. Low cost processors should lead to machines with distributed intelligence; extended storage devices should lead to systems with improved I/O performance due to buffering techniques. What is not clear is the most effective use of these advances: what CPU functions to distribute; and when and how to buffer I/O.

The problem of determining how to buffer I/O is essentially the problem of determining 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 by obtaining a large percentage of hits in the buffer. Therefore, in order to effectively use the increased buffer size made possible by extended storage devices, there must exist sequentiality and/or locality of access in the data.

If the I/O references are known to be sequential, the data can be stored sequentially on the disk and read into main memory or an extended storage device an entire track or cylinder at a time, thus dramatically reducing access time. This read ahead tactic is often employed by operating systems when processing sequential files. If there is locality of reference in the I/O requests, then data pages should be buffered in faster memory and a replacement algorithm used. Such pages should be treated in much the same way as program pages are handled in virtual memory machines [DENN68].

There are several studies of data management system I/O reference patterns which deal with buffer replacement algorithms [SHER76, TUEL76, LANG77], but do not directly address the question of under what conditions there is locality or sequentiality in the I/O references. In studies of an IMS trace [RODR76], sequentiality was found in the data references, and in [SMIT76] read-ahead strategies to take advantage of that sequentiality were analyzed. The trace analyzed in [RODR76] also exhibited some inter-query locality of reference.

CPU usage patterns have not been previously considered in published performance evaluations of data management systems. It is often assumed that the CPU time in a data management system is insignificant compared to the I/O time [YAO78]. However, the CPU can become the bottleneck in data management systems. This may become a critical problem if I/O time is reduced through the use of extended storage devices or read ahead tactics. One method of lessening a CPU bottleneck is to distribute the processing of the queries through the use of intelligent terminals and/or back end machines. We will indicate under what circumstances one data base management system is CPU bound and what forms of distributed processing appear to be beneficial.

B. Description of this study

The performance of the data management system INGRES [STON76] was analyzed to determine its CPU usage and data reference patterns. The study was done using a software trace facility and benchmark query streams.

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. Therefore, INGRES, the benchmark programs, and the results are explained in detail. The description of INGRES and the trace are in Section II.

In Section III the benchmark query streams are described. We did not attempt to trace a "typical user" for a "typical week" but, instead, constructed a series of benchmarks. This technique was used because the goal was to identify and analyze the performance patterns of general types of queries, not to determine the mixture of these queries in any one user's query stream. Three sets of benchmarks were developed from user query streams. These were overhead-intensive, data-intensive, and multi-relation queries. Since all updates in INGRES are physically implemented by a "Retrieve" to isolate what actions to take followed by lower level shuffling, we treated only "Retrieves" in this study.

Section III also contains the results of the analysis of the benchmarks. It is shown that the performance patterns of the overhead-intensive queries differ greatly from the data-intensive queries. There is locality of reference to INGRES system files for overhead-intensive queries; moreover, they are CPU bound. It is shown that data-intensive queries exhibit a high degree of sequentiality, and cyclic sequentiality. In addition there is intraquery locality of reference in INGRES only when processing aggregate functions or multi-relation queries. The CPU usage patterns of INGRES are analyzed to determine the most efficient distribution of the functions of the CPU. Lastly, Section IV summarizes our conclusions.

Section II. INGRES

The following is a brief description of INGRES. Complete descriptions of the INGRES data management system are contained in [STON76, WONG76, EPST77].

INGRES supports three structures for a relation: it may be an unordered collection of tuples (heap), it may be hashed on any domain or any combination of domains (hash); or it may be stored indexed sequential on any domain or combination of domains (ISAM). Secondary indices are allowed on any domain or any combination of domains.

A query such as

```
retrieve (employee.name) where employee.empnum = 1234
```

results in the searching of the employee relation for all tuples with empnum = 1234. In the above query, if the employee relation is stored as a heap, the entire relation will be scanned, each tuple tested for empnum = 1234, and the "name" domain in the qualifying tuples returned to the user. The relation is read in logically sequential order, one page at a time.

If the relation is hashed on empnum, the hash function (which is selected automatically by INGRES and has a bucket size equal to one UNIX page) is used to determine the main page the tuples satisfying the qualification would fall on. That page is read, then each tuple on that page is tested and the data in the qualifying tuples returned to the user.

The case where more tuples hash to a location than can fit on one page is handled by chaining overflow pages. If there are overflow pages, they are also read as above.

If the relation is ISAM, the index associated with the relation is first searched to determine the main page that the tuples would fall on, and then processing continues as in a hashed relation.

If the relation is not structured on empnum, but has a secondary index on empnum, the secondary index is searched to find the logical page number and offset within the page of the qualifying tuples. All pages containing a qualifying tuple are read, and the data in the qualifying tuples returned to the user. Note in this case that the secondary index identifies qualifying tuples. Hence, unlike ISAM and hash, a data page does not have to be exhaustively searched.

B. Significant Considerations about the INGRES Environment

The process structure

Because DEC PDP-11 computers have an address space limitation of 64K bytes per process, INGRES is separated into 5 processes. These processes are the terminal monitor, parser, decomposition (decomp), single relation query processor (one variable query processor, OVQP), and the database utilities (DBU). INGRES is an interpretative system. The terminal monitor fields the user's query and passes it to the parser. The parser parses the query and passes it to the decomposition process, which decomposes the query into single relation queries, which are executed by OVQP. The database utilities include such functions as creating relations and modifying storage structures. The processes communicate by using UNIX protocols.

This five process structure has considerable impact on the performance of INGRES. The CPU time per query is increased by the cost of interprocess communication (approximately 0.2 sec. per query for the five processes). Both CPU and I/O time are increased by the same validity checking which must be done independently by more than one process. This extra cost is unavoidable in a 16 bit machine.

It is estimated that the CPU time of INGRES, since it is interpreted rather than compiled, is substantially greater than the CPU time of a compiled system. If the system were compiled, almost all of the cost in the parser and decomp could be paid at compile time rather than runtime. Additionally, the OVQP time might be reduced by a factor of two or three.

Terminal Monitor

INGRES supports two interfaces to users: a stand alone terminal monitor for interactively formulating and executing queries and a programming language interface, EQUEL [ALLM76]. In effect, the latter interface replaces the terminal monitor with a custom host language program. Production use of INGRES involves both executing terminal commands through the terminal monitor and running EQUEL programs.

The benchmarks which follow use the terminal monitor as they result from scripts of terminal commands. However, it should be noted that the terminal monitor was designed to provide a large class of services and is not particularly efficient. The single feature of user defined macros can require as much as 2 seconds per command in the terminal monitor. This cost is not present for EQUEL applications. In fact, the time spent in the EQUEL host program can be easily kept under 100 msec per query.

UNIX

The costs of using the UNIX operating system for I/O are that pages are constrained to be 512 bytes long, and logically sequential pages in a file are not necessarily stored physically sequential. There is a facility for defining page sizes longer than 512 bytes, but the operating system will, in fact, store the larger pages as two or more disjoint 512-byte pages. A page is read into UNIX process space, then copied into INGRES process space, which also increases the CPU time.

The performance data which follows reflects all these costs. Note very clearly that some of the costs are inevitable (address space), some reflect design decisions (e.g. interpretation), and some reflect the choice of the user program which "drives" INGRES (terminal monitor rather than an EQUEL program).

C. The software trace

Software probes placed in the operating and data management systems consist of calls to a system trace subroutine with the relevant data as parameters. The subroutine reads a clock with 0.0001 second resolution and appends that time to the data. It then writes the package to one of 3-1K byte buffers, checking to see if the buffer is full. If it is, the buffer is written to tape. The trace adds about 10% to the normal running time of the query stream. For this report, the probes placed in INGRES traced the logical page numbers and relation names for each read and write, and the total CPU time for each process. The probes placed in the operating system traced the disk read and write time. The trace information was then analyzed to produce the following results.

Section III. Performance Analysis

A. Overview

Three sets of benchmark programs were developed: sets of overhead-intensive, data-intensive, and multi-relation queries. Overhead-intensive and data-intensive are general types of queries which appear in other data management systems. The differentiation is similar in concept to the differentiation 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 the 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.

The performance pattern expected of an overhead-intensive query was that it would contain no locality of reference within the query, because by definition it references little data. It was not known initially whether the overhead was CPU or I/O bound.

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 analysis 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.

This situation arises in poorly designed databases and in data management systems, such as INGRES, that support ad hoc queries.

The performance pattern expected of data-intensive queries was that they would be I/O bound, and that there would be little re-referencing of data.

Data-intensive ("long") queries and overhead-intensive ("short") queries exist in other data management systems. The third set of benchmarks, multi-relation queries, are specific to relational systems. Due to the particular implementation of INGRES, the results from these queries may not generalize to other relational systems. They are queries which reference more than one relation, and, because of the INGRES implementation of them, were expected to show significant locality of I/O references.

B. Results

The benchmarks were run single-user on a DEC PDP 11/70. This is a 16-bit minicomputer, with a 2K byte cache, and will perform 3.15 million register increments per second. The databases were on a disk with an average access time of .030 seconds per 512-byte block.

1) Overhead-intensive queries

Three query streams were developed to determine the performance patterns for overhead-intensive queries. The three benchmarks were taken from a collection of user queries from an application used by the UC Berkeley EECS Department. The application is course and room scheduling, where 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 application programmer used the INGRES macro facility to define a query:

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

The terminal operator merely specified a query number, course number, and professor's name, and the above query was executed with the appropriate substitutions made (namely, course number for courseNN, professor's name for name). In the script used to form the benchmark Short1, the above query was duplicated, exactly as the user wrote it, with 76 professor's names substituted for "name" and any one of eight courses (picked at random) substituted for courseNN. Since the names were all unique, and the courses were picked at random, the queries are guaranteed to generate random references to data. This was done to study the performance patterns of short, random queries as would normally be found in this application.

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, as it is simply a way to get around the absence of a "Retrieve Unique" command for INGRES. 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 will show the following:

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

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

a) I/O reference patterns

Table 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 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 measured 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 then 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 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.

In INGRES a "transaction" is a single query, so we may not refer to the "destroy - retrieve into - print" set as a transaction. We will call it a query set. 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 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 often. 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.

i) sequentiality of reference

The high percentages of sequential references we see in Table 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 whenever the queries were run on newly modified system relations. Therefore, the sequentiality observed in Table 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.

ii) locality of reference

A commonly accepted measure of locality is the hit-ratio curve. The hit ratio curves for the overhead-intensive queries are presented in Figures 1 and 2.

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 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 2, which shows the same curves, but with the references to the system relations removed. The line for short1 in Figure 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, which is not surprising since the query script that made up the benchmark was chosen to be random. 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 references are provided because the references are random.

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 (i.e. there is more activity for a plane about to leave than one scheduled for next week).

Therefore, for overhead-intensive queries, the only reliable locality of reference appears to be the references to the system relations. It is clear that caching the system relations would be very beneficial. However, this results directly from the interpretive nature of INGRES and would not be applicable to a compiled system.

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.

b) CPU usage patterns

Table 2 contains the CPU usage patterns for overhead-intensive queries. The CPU time for each process is given in Table 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 2. Overhead-intensive queries: CPU usage patterns
all times are in seconds

query stream	monitor	parser	decomp	OVQP total	dp	DBU	total	total - monitor
short1	1.91	.30	.21	.52	(.16)	4.48	7.42	5.51
short2	1.75	.98	.11	.38	(.15)		3.22	1.47
short3	.88	.26	.05	.28	(.12)		1.47	.59

(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 occurs because there are fewer domains in Short3. 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.

conclusion:

Except for short1, which is probably not a general query stream, the terminal monitor requires the largest percentage of the overhead time - 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 found in Table 2 is, for all three cases, greater than the total I/O time per query found in Table 1. We therefore conclude that INGRES is always CPU bound when handling overhead-intensive queries. Moreover, this statement is true even if terminal monitor time is zero, as noted in column 9, which represents a "super efficient" EQUEL program. In such a circumstance, INGRES will execute about 1.6 queries from short3 per second and use less than half the available disk transfers.

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 or compilation of queries. In the overhead-intensive queries in the benchmark query streams, the relocation 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.

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.

a) I/O reference patterns

Table 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 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.

i) 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.

ii) locality of reference

In figures 3 and 4 we present the hit ratio curves for these benchmarks. Note that long1 is a gently rising curve in both figures, while long2 is nearly flat. The rising nature of long1 appears to be due entirely to the INGRES implementation of aggregate functions where random re-referencing of (relatively small) temporaries is taking place.

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 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.

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 much like the locality found in program references (figure 7). The buffer size needed to take advantage of this type of locality can be relatively small and is related to the number of values for which aggregates are simultaneously being performed. The second type of locality, the cyclic sequentiality (as seen in the curves for long3) would require a buffer size equal to the size of the relation being repeatedly scanned. Since this can be very large, it may not be cost effective to deal with this situation by "blind buffering". Rather, data read-ahead may be especially attractive here. It may be best to simply ignore the cyclic sequential case, and only do the read-ahead that the large amount of sequentiality mandates.

b) CPU usage

Table 4. Data-intensive queries: CPU usage patterns
all times are in seconds

query stream	monitor	parser	decomp	OVQP total	dp	DBU	total	total - monitor
long1	1.7	.4	.8	7.1	(6.9)	5.8	15.8	14.1
long2	2.0	.9	.26	11.23	(11.0)		14.39	12.4
long3	1.3	.3	.07	3.73	(3.5)		5.4	4.1

(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 queries 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.

conclusion:

We note that in all cases the data processing time (dp) in OVQP is the greatest single item of CPU time, and that I/O time is approximately equal to CPU time. It should be consequently noted that both improved buffering and distributing the processing toward the data would be required to improve performance for data intensive queries. Because INGRES interprets queries CPU time is probably higher than compiled systems. On the other hand, our configuration contains only one disk controller. A multiprogrammed benchmark with multiple I/O controllers could cut the effective I/O time substantially. Hence, both I/O and CPU time are higher than might be the case in an alternate architecture. This lends credence to the possibility that the above conclusion is not specific to our environment. Hence, it is possible that designers of data base machines (e.g. RAP, DIRECT) should pay careful attention to staging of data as well as to creating a distributed processing environment.

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, read ahead policies 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 cyclic sequentiality 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 in a cyclic sequential fashion. 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. It should be clearly noted that all 20 queries could have been processed in parallel with one sequential pass of temp2. Unfortunately, the current implementation does not support such a strategy.

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.

a) I/O reference patterns

Table 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 rooms 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 5 and 6 show the hit-ratio curves for the multi-relation queries. 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.

Hence, this information can be used advantageously in a buffering or read ahead policy.

b) CPU usage

Table 6. Multi-relation queries: CPU usage patterns
all times are in seconds

query stream	monitor	parser	decomp	OVQP total	dp	DBU	total	total - monitor
multi1	1.0	.25	.52	4.33	(4.2)	1.15	7.25	6.3
rooms	1.37	.81	.71	705.75	(705.4)	1.86	709.5	708.1

(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.

conclusion:

It might be claimed that the above numbers simply reflect the INGRES algorithms for handling multi-variable queries and would not be indicative of referencing patterns of the other approaches (e.g. [BLAS76]). We might note that other algorithms often include sorting. Many sorting algorithms involve a limited kind of cyclic sequentiality of referencing, at least to write data, then read it in again. Hence, our results may be representative of such algorithms also.

The CPU time per query in Table 6 is greater than the I/O time per query (Table 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, or an effective read ahead policy would appear very beneficial.

Section IV. Summary

A) Sequentiality of data reference

Comparing the hit ratio curves in figures 1-6 and the typical hit ratio curve for program references in figure 7, we note that INGRES data references are in most cases very different from program references. This difference results from sequentiality of reference. This sequentiality can be used to increase performance by placing pages which are logically sequential in a file physically sequential on the disk. Then, if a relation is being read sequentially, when one page on a disk track is read, several pages or even the remainder of the track can also be read. It is even possible to read an entire cylinder at a time if an extended storage device is available. The amount which should be prefetched depends on available buffer space and whether the data base system is I/O bound. In the best case read ahead would reduce the time to fetch a disk block from about 30 msec to less than 3 msec; an order of magnitude improvement in I/O system performance.

Clearly, it is only wise to read ahead if one knows the additional data will be used. Therefore the system handling I/O and buffer management must have the information that the data is being accessed sequentially. In an analysis of IMS data references, [SMIT76], it is recommended the number of blocks to be read ahead be based on the number of blocks previously read sequentially. This is a guess, and necessary in IMS because the level of the user interface to the data management system is one record at a time. But in INGRES and other high-level systems the user interface is at least one relation at a time. There is no need to guess. INGRES knows when it must read all or a part of a file sequentially. It can either pass that information to the operating system through a new read command (read sequentially) or do its own I/O management so that read-ahead can be selectively invoked.

B) Locality

Locality of reference (as opposed to sequentiality) was found in references to the system catalogs and in processing aggregate functions. Moreover, cyclic sequentiality was found in the processing of multirelation queries. The locality of reference for system relations can be utilized by permanently storing them on an extended storage device. For a long time operating systems have used the technique of keeping hierarchies of directories in main memory and on fixed head disks. Alternately, INGRES could be redesigned to do minimal run time checking.

The cyclic sequential locality of reference in multi-relation queries can be used either by implementing a cache or by invoking a read ahead policy. Lastly, the processing of aggregate functions can be expedited only by a buffer cache. INGRES is aware of when a multi-relation query or aggregate function is being processed and can signal the operating system that cyclic sequential or random referencing will be taking place.

C) Distributed processing

In INGRES the performance bottleneck for overhead-intensive queries and for many data-intensive queries is the CPU. Distributed processing is one solution to the CPU-bound problem. Several data management machines have been designed which include distributed processing. We have shown that for data-intensive queries, distributing the processing toward the data certainly will result in performance improvement and that for overhead-intensive queries distributing the processing toward the user will result in performance improvement.

Although these two solutions do not appear to be in conflict, they are. If distributing the processing toward the data requires overhead that increases the processing time of the overhead-intensive query, and if a substantial portion of the use of the system will be overhead-intensive queries, the performance will degrade.

D) Conclusion

The data usage and I/O reference patterns found in the processing of queries by INGRES show that the performance of INGRES may be improved dramatically by using some combination of extended memory, read ahead and multiple processors.

For overhead-intensive queries, this may be done by:

1) distributing processing at the terminal monitor level

2) Using extended storage (or main memory) to cache temporary and system relations

For data-intensive queries, this may be done by:

1) distributing the processing at the data level

2) using main memory or an extended storage device to implement large, read-ahead buffers to take advantage of the sequential reading of data.

3) using main memory or an extended storage device to cache the temporary relations formed and referenced during the processing of aggregate functions.

For multi-relation queries:

1) distributing the processing at the data level.

2) caching the relations to be cyclically re-referenced in an extended storage device or invoking read ahead.

INGRES, in its present form, is CPU bound most of the time. Therefore, the benefits received by improving I/O speed would not appreciably effect the query response time unless the CPU time is decreased. This can be done by distributing the processing or compiling a part of the query processing.

Generalization to other systems:

The INGRES system, which is forced into a multi-process structure and which supports a complex terminal monitor, may have a higher overhead than most data-management systems. However, any system that supports a highly functional user interface may benefit from putting as much of the interface as possible into intelligent terminals. Those systems that do run-time checking of system catalogs may benefit from caching them in extended storage.

In the case of data-intensive queries, sequentiality of access has been found in another data management system [RODG76], and may be present in any data management systems that support such facilities as report generators. Therefore the caching of read-ahead data blocks appears to be a good technique for any system which

1) has space available in main memory or an extended storage device.

2) has the data organized on the disks to take advantage of sequential reads, so that actual I/O time is saved by reading several disk blocks at once.

3) can communicate with the operating system that data is to be read sequentially.

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.

[BLAS76] Blasgen, M.W. and Eswaren, K.P., "On the Evaluation of Queries in a Relational Data Base System," IBM Research Report RJ-1745, Aprill, 1976.

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

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

[EPST77] Epstein, R., "Creating and Maintaining a Database Using INGRES," Electronics Research Laboratory, University of California, Berkeley, Ca., Memo #M77-71, Dec. 1977.

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

[LANG77] Lang, Thomas, Wood, Christopher and Fernandez, Eduardo f., "Database Buffer paging in Virtual Storage Systems," TODS, Vol. 2, No. 4, December, 1977.

[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.

[RODR76] Rodriguez-Rosell, Juan, "Empirical Data Reference Behavior in Data Base Systems," Computer, Nov., 1976, Pages 9-13.

[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," Communications ACM 17, 7 , July,1974.

[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 Reséarch Report RJ 1743, March 19, 1976.

[STON76] Stonebraker, M. et. al., "The Design and Implementation of INGRES," TODS, Vol 1, No. 3, September 1976.

[TUEL76] Tuel, W. G. Jr., "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.

Overhead intensive queries

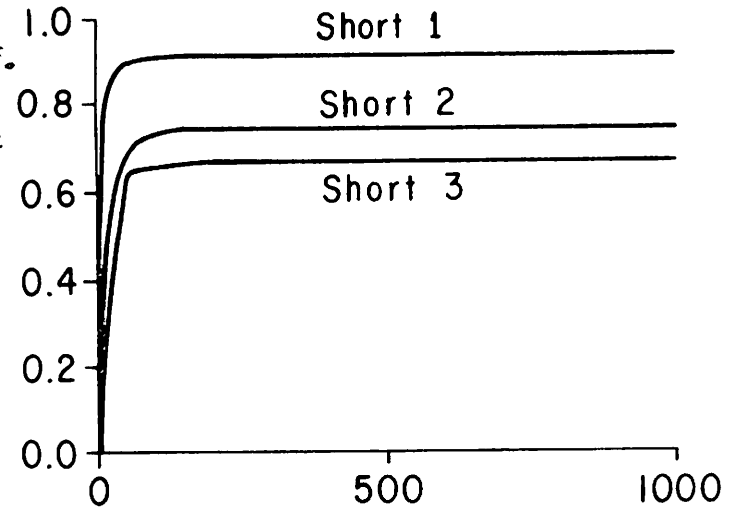

Fig. 1

Overhead intensive queries

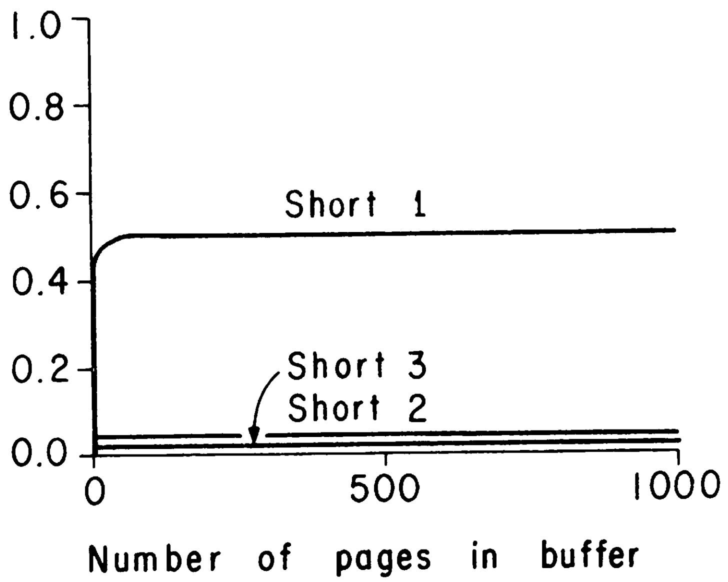

Fig. 2

Data - intensive queries

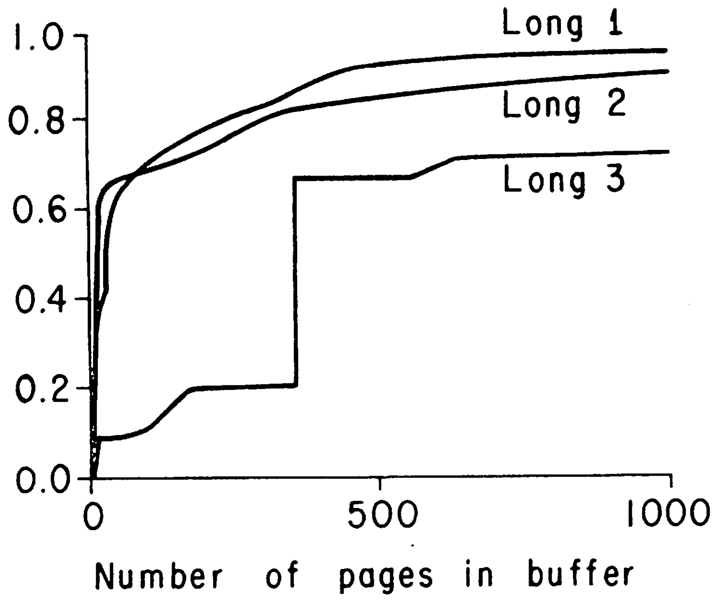

Fig. 3

Data - intensive queries
system references removed

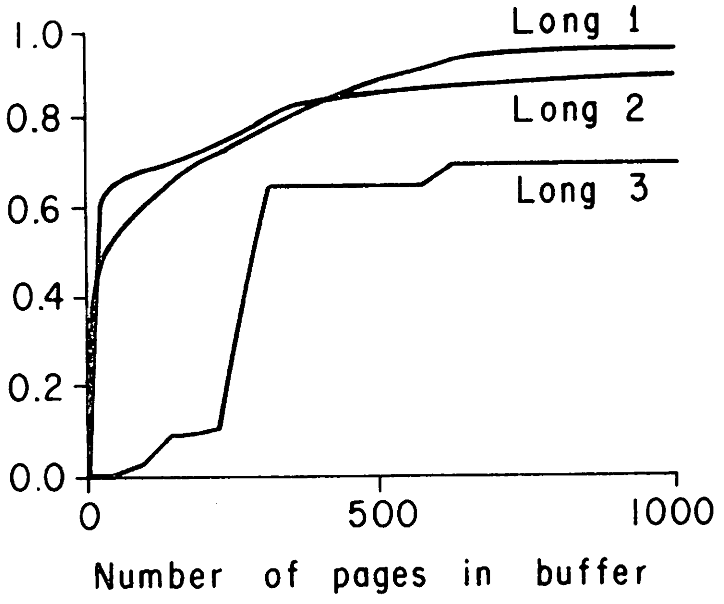

Fig. 4

Multi-relation queries

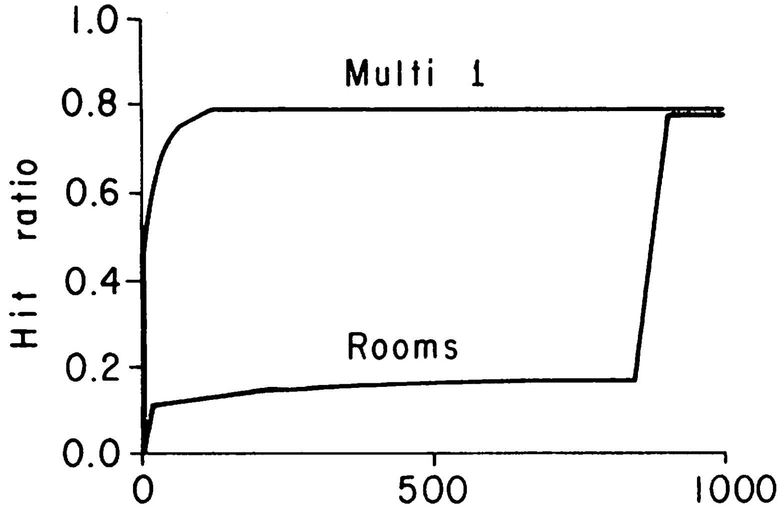

Fig. 5

Multi- relation queries
system references removed

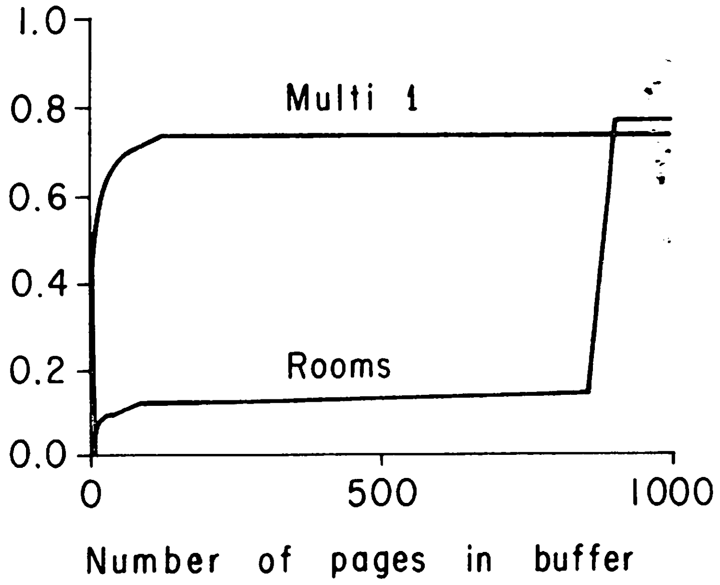

Fig. 6

Typical hit ratio curve
for program references

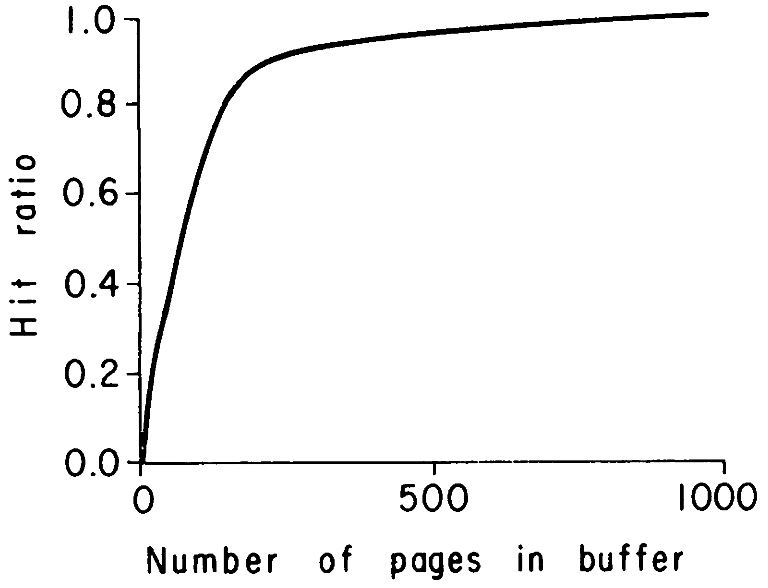

Fig. 7