An Evaluation of Buffer Management Strategies for Relational Database Systems

Hong-Tai Chou[†]
David J. DeWitt

Computer Sciences Department
University of Wisconsin

ABSTRACT

In this paper we present a new algorithm, DBMIN, for managing the buffer pool of a relational database management system. DBMIN is based on a new model of relational query behavior, the **query locality set model** (QLSM). Like the hot set model, the QLSM has an advantage over the stochastic models due to its ability to predict future reference behavior. However, the QLSM avoids the potential problems of the hot set model by separating the modeling of reference behavior from any particular buffer management algorithm. After introducing the QLSM and describing the DBMIN algorithm, we present a performance evaluation methodology for evaluating buffer management algorithms in a multiuser environment. This methodology employed a hybrid model that combines features of both trace driven and distribution driven simulation models. Using this model, the performance of the DBMIN algorithm in a multiuser environment is compared with that of the hot set algorithm and four more traditional buffer replacement algorithms.

1. Introduction

In this paper we present a new algorithm, DBMIN, for managing the buffer pool of a relational database management system. DBMIN is based on a new model of relational query behavior, the **query locality set model** (QLSM). Like the hot set model [Sacc82], the QLSM has an advantage over the stochastic models due to its ability to predict future reference behavior. However, the QLSM avoids the potential problems of the hot set model by separating the modeling of reference behavior from any particular buffer management algorithm. After introducing the QLSM and describing the DBMIN algorithm, the performance of the DBMIN algorithm in a multiuser environment is compared with that of the hot set algorithm and four more traditional buffer replacement algorithms.

A number of factors motivated this research. First, although Stonebraker [Ston81] convincingly argued that conventional virtual memory page replacement algorithms (e.g. LRU) were generally not suitable for a relational database environment, the area of buffer management has, for the most part, been ignored (contrast the activity in this area with that in the concurrency control area). Second, while the hot set results were encouraging they were, in our opinion, inconclusive. In particular, [Sacc82] [Sacc85] presented only limited simulation results of the hot set algorithm. We felt that extensive, multiuser tests of the hot set algorithm and conventional replacement policies would provide valuable insight into the effect of the buffer manager on overall system performance.

In Section 2, we review earlier work on buffer management strategies for database systems. The QLSM and DBMIN algorithm are described in Section 3. Our multiuser performance evaluation of alternative buffer replacement policies is presented in Section 4. Section 5 contains our conclusions and suggestions for future research.

2. Buffer Management for Database Systems

While many of the early studies on database buffer management focused on the double paging problem [Fern78] [Lang77] [Sher76a] [Sher76b] [Tuel76], recent research efforts have been focused on finding

[†]Author's current address is: Microelectronics and Computer Technology Corporation, 9430 Research Blvd., Echelon Bldg. #1, Austin, TX 78759.

buffer management policies that "understand" database systems [Ston81] and know how to exploit the predictability of database reference behavior. We review some of these algorithms in this section.

2.1. Domain Separation Algorithms

Consider a query that randomly accesses records through a B-tree index. The root page of the B-tree is obviously more important than a data page, since it is accessed with every record retrieval. Based on this observation, Reiter [Reit76] proposed a buffer management algorithm, called the **domain separation** (DS) algorithm, in which pages are classified into types, each of which is separately managed in its associated domain of buffers. When a page of a certain type is needed, a buffer is allocated from the corresponding domain. If none are available for some reason, e.g. all the buffers in that domain have I/O in progress, a buffer is borrowed from another domain. Buffers inside each domain are managed by the LRU discipline. Reiter suggested a simple type assignment scheme: assign one domain to each non-leaf level of the B-tree structure, and one to the leaf level together with the data. Empirical data[1] showed that this DS algorithm provided 8-10% improvement in throughput when compared with an LRU algorithm.

The main limitation of the DS algorithm is that its concept of domain is static. The algorithm fails to reflect the dynamics of page references as the importance of a page may vary in different queries. It is obviously desirable to keep a data page resident when it is being repeatedly accessed in a nested loops join. However, it is not the case when the same page is accessed in a sequential scan. Second, the DS algorithm does not differentiate the relative importance between different types of pages. An index page will be over-written by another incoming index page under the DS algorithm, although the index page is potentially more important than a data page in another domain. Memory partitioning is another potential problem. Partitioning buffers according to domains, rather than queries, does not prevent interference among competing users. Lastly, a separate mechanism needs to be incorporated to prevent thrashing since the DS algorithm has no built-in facilities for load control.

Several extensions to the DS algorithm have been proposed. The group LRU (GLRU) algorithm, proposed by Hawthorn [Nybe84], is similar to DS, except that there exists a fixed priority ranking among different groups (domains). A search for a free buffer always starts from the group with the lowest priority. Another alternative, presented by Effelsberg and Haerder [Effe84], is to dynamically vary the size of each domain using a working-set-like [Denn68] partitioning scheme. Under this scheme, pages in domain i which have been referenced in the last τ_i references are exempt from replacement consideration. The "working set" of each domain may grow or shrink depending on the reference behavior of the user queries. Although empirical data indicated that dynamic domain partitioning can reduce the number of page faults (of the system) over static domain partitioning, Effelsberg and Haerder concluded that there is no convincing evidence that the page-type-oriented schemes[2] are distinctly superior to global algorithms, such as LRU and CLOCK.

2.2. "New" Algorithm

In a study to find a better buffer management algorithm for INGRES [Ston76], Kaplan [Kapl80] made two observations from the reference patterns of queries: the priority to be given to a page is not a property of the page itself but of the relation to which it belongs; each relation needs a "working set". Based on these observations, Kaplan designed an algorithm, called the "new" algorithm, in which the buffer pool is subdivided and allocated on a per-relation basis. In this "new" algorithm, each active relation is assigned a resident set which is initially empty. The resident sets of relations are linked in a priority list with a global free list on the top. When a page fault occurs, a search is initiated from the top of the priority list until a suitable buffer is found. The faulting page is then brought into the buffer and added to the resident set of the relation. The MRU discipline is employed within each relation. However, each relation is entitled to one active buffer which is exempt from replacement consideration. The ordering of relations is determined, and may be adjusted subsequently, by a set of heuristics. A relation is placed near the top if its pages are unlikely to be reused. Otherwise, the relation is protected at the bottom. Results from Kaplan's simulation experiments suggested that the "new" algorithm performed much better than the UNIX buffer manager. However, in a trial implementation [Ston82], the "new" algorithm failed to improve the performance of an experimental version of INGRES which uses an LRU algorithm.

The "new" algorithm presented a new approach to buffer management, an approach that tracks the locality of a query through relations. However, the algorithm itself has several weak points. The use of MRU is justifiable only in limited cases. The rules suggested by Kaplan for arranging the order of relations on the priority list were based solely on intuition. Furthermore,

[1] In Reiter's simulation experiments, a shared buffer pool and a workload consisting of 8 concurrent users were assumed.

[2] The DS algorithm is called a page-type-oriented buffer allocation scheme in [Effe84].

under high memory contention, searching through a priority list for a free buffer can be expensive. Finally, extending the "new" algorithm to a multi-user environment presents additional problems as it is not clear how to establish priority among relations from different queries that are running concurrently.

2.3. Hot Set Algorithm

The hot set model proposed by Sacco and Schkolnick [Sacc82] is a query behavior model for relational database systems that integrates advance knowledge on reference patterns into the model. In this model, a set of pages over which there is a looping behavior is called a **hot set**. If a query is given a buffer large enough to hold the hot sets, its processing will be efficient as the pages referenced in a loop will stay in the buffer. On the other hand, a large number of page faults may result if the memory allocated to a query is insufficient to hold a hot set. Plotting the number of page faults as a function of buffer size, we can observe a discontinuity around the buffer size where the above scenario takes place. There may be several such discontinuities in the curve, each is called a **hot point**.

In a nested loops join in which there is a sequential scan on both relations, a hot point of the query is the number of pages in the inner relation plus one. The formula is derived by reserving enough buffers to hold the entire inner relation, which will be repeatedly scanned, plus one buffer for the outer relation, which will be scanned only once. If, instead, the scan on the outer relation is an index scan, an additional buffer is required for the leaf pages of the index. Following similar arguments, the hot points for different queries can be determined.

Applying the predictability of reference patterns in queries, the hot set model provides a more accurate reference model for relational database systems than a stochastic model. However, the derivation of the hot set model is based partially on an LRU replacement algorithm, which is inappropriate for certain looping behavior. In fact, the MRU (Most-Recently-Used) algorithm, the opposite to an LRU algorithm, is more suited for cycles of references [Thor72], because the most-recently-used page in a loop is the one that will not be re-accessed for the longest period of time. Going back to the nested loops join example, the number of page faults will not increase dramatically when the number of buffers drops below the "hot point" if the MRU algorithm is used. In this respect, the hot set model does not truly reflect the inherent behavior of some reference patterns, but rather the behavior under an LRU algorithm.

In the hot set (HOT) algorithm, each query is provided a separate list of buffers managed by an LRU discipline. The number of buffers each query is entitled to is predicted according to the hot set model. That is, a query is given a local buffer pool of size equal to its **hot set size**. A new query is allowed to enter the system if its hot set size does not exceed the available buffer space.

As discussed above, the use of LRU in the hot set model lacks a logical justification. There exist cases where LRU is the worse possible discipline under tight memory constraint. The hot set algorithm avoids this problem by always allocating enough memory to ensure that references to different data structures within a query will not interfere with one another. Thus it tends to over-allocate memory, which implies that memory may be under-utilized. Another related problem is that there are reference patterns in which LRU does perform well but is unnecessary since another discipline with a lower overhead can perform equally well.

3. The DBMIN Buffer Management Algorithm

In this section, we first introduce a new query behavior model, the **query locality set model** (QLSM), for database systems. Using a classification of page reference patterns, we show how the reference behavior of common database operations can be described as a composition of a set of simple and regular reference patterns. Like the hot set model, the QLSM has an advantage over the stochastic models due to its ability to predict future reference behavior. However, the QLSM avoids the potential problems of the hot set model by separating the modeling of reference behavior from any particular buffer management algorithm.

Next we describe a new buffer management algorithm termed DBMIN based on the QLSM. In this algorithm, buffers are allocated and managed on a **per file instance** basis. Each file instance is given a local buffer pool to hold its **locality set**, which is the set of the buffered pages associated the file instance. DBMIN can be viewed as a combination of a working set algorithm [Denn68] and Kaplan's "new" algorithm in the sense that the locality set associated with each file instance is similar to the working set associated with each process. However, the size of a locality set is determined in advance, and needs not be re-calculated as the execution of the query progresses. This predictive nature of DBMIN is close to that of the hot set algorithm. Similar to the WS and the hot set algorithm[3], DBMIN uses a dynamic partitioning scheme, in

[3] The issue of memory partitioning was not clearly addressed in [Sacc82]. However, it was later shown in [Sacc85] how dynamic memory partitioning can be achieved by decomposing a query into sub-evaluation plans, each of which is independently characterized by the hot set model.

which the total number of buffers assigned to a query may vary as files (relations) are opened and closed.

3.1. The Query Locality Set Model

The QLSM is based on the observation that relational database systems support a limited set of operations and that the pattern of page references exhibited by these operations are very regular and predictable. In addition, the reference pattern of a database operation can be decomposed into the composition of a number of simple reference patterns. Consider, for example, an index join with an index on the joining attribute of the inner relation. The QLSM will identify two locality sets for this operation: one for the sequential scan of the outer relation and a second for the index and data pages of the inner relation. In this section, we present a taxonomy for classifying the page reference patterns exhibited by common access methods and database operations[4]

Sequential References

In a sequential scan, pages are referenced and processed one after another. In many cases, a sequential scan is done only once without repetition. For example, during a selection operation on an unordered relation, each page in the file is accessed exactly once. A single page frame provides all the buffer space that is required. We shall refer to such a reference pattern as **straight sequential** (SS).

Local re-scans may be observed in the course of a sequential scan during certain database operations. That is, once in a while, a scan may back up a short distance and then start forward again. This can happen in a merge join [Blas77] in which records with the same key value in the inner relation are repeatedly scanned and matched with those in the outer relation. We shall call this pattern of reference **clustered sequential** (CS). Obviously, records in a cluster (a set of records with the same key value) should be kept in memory at the same time if possible.

In some cases, a sequential reference to a file may be repeated several times. In a nested loops join, for instance, the inner relation is repeatedly scanned until the outer relation is exhausted. We shall call this a **looping sequential** (LS) pattern. The entire file that is being repeatedly scanned should be kept in memory if possible. If the file is too large to fit in memory, an MRU replacement algorithm should be used to manage the buffer pool.

[4] A similar analysis of query reference behavior was independently derived in [Sacc85].

Random References

An **independent random** (IR) reference pattern consists a series of independent accesses. As an example, during an index scan through a non-clustered index, the data pages are accessed in a random manner. There are also cases when a locality of reference exists in a series of "random" accesses. This may happen in the evaluation of a join in which a file with a non-clustered and non-unique index is used as the inner relation, while the outer relation is a clustered file with non-unique keys. This pattern of reference is termed **clustered random** (CR). The reference behavior of a CR reference is similar to that of a CS scan. If possible, each page containing a record in a cluster should be kept in memory.

Hierarchical References

A hierarchical reference is a sequence of page accesses that form a traversal path from the root down to the leaves of an index. If the index is traversed only once (e.g. when retrieving a single tuple), one page frame is enough for buffering all the index pages. We shall call this a **straight hierarchical** (SH) reference. There are two cases in which a tree traversal is followed by a sequential scan through the leaves: **hierarchical with straight sequential** (H/SS), if the scan on the leaves is SS, or **hierarchical with clustered sequential** (H/CS), otherwise. Note that the reference patterns of an H/SS reference and an H/CS reference are similar to those of an SS reference and a CS reference, respectively.

During the evaluation of a join in which the inner relation is indexed on the join field, repeated accesses to the index structure may be observed. We shall call this pattern of reference as **looping hierarchical** (LH). In an LH reference, pages closer to the root are more likely to be accessed than those closer to the leaves. The access probability of an index page at level i, assuming the root is at level 0, is inversely proportional to the ith power of the fan-out factor of an index page. Therefore, pages at an upper level (which are closer to the root) should have higher priority than those at a lower level. In many cases, the root is perhaps the only page worth keeping in memory since the fan-out of an index page is usually high.

3.2. DBMIN - A Buffer Management Algorithm Based on the QLSM

In the DBMIN algorithm, buffers are allocated and managed on a **per file instance** basis[5]. The set of

[5] Active instances of the same file are given different buffer pools, which are independently managed. However, as we will explain later, all the file instances share the same copy of a buffered page whenever possible through a global table mechanism.

buffered pages associated with a file instance is referred to as its **locality set**. Each locality set is separately managed by a discipline selected according to the intended usage of the file instance. If a buffer contains a page that does not belong to any locality set, the buffer is placed on a global free list. For simplicity of implementation, we restrict that a page in the buffer can belong to at most one locality set. A file instance is considered the owner of all the pages in its locality set. To allow for data sharing among concurrent queries, all the buffers in memory are also accessible through a global buffer table. The following notation will be used in describing the algorithm:

N - the total number of buffers (page frames) in the system;

l_{ij} - the maximum number of buffers that can be allocated to file instance **j** of query **i**;

r_{ij} - the number of buffers allocated to file instance **j** of query **i**.

Note that **l** is the desired size for a locality set while **r** is the actual size of a locality set.

At start up time, DBMIN initializes the global table and links all the buffers in the system on the global free list. When a file is opened, its associated locality set size and replacement policy are given to the buffer manager. An empty locality set is then initialized for the file instance. The two control variables **r** and **l** associated with the file instance are initialized to 0 and the given locality set size, respectively.

When a page is requested by a query, a search is made to the global table, followed by an adjustment to the associated locality set. There are three possible cases:

(1) **The page is found in both the global table and the locality set:** In this case, only the usage statistics need to be updated if necessary as determined by the local replacement policy.

(2) **The page is found in memory but not in the locality set:** If the page already has an owner, the page is simply given to the requesting query and no further actions are required. Otherwise, the page is added to the locality set of the file instance, and **r** is incremented by one. Now if **r** > **l**, a page is chosen and released back to the global free list according to the local replacement policy, and **r** is set to **l**. Usage statistics are updated as required by the local replacement policy.

(3) **The page is not in memory:** A disk read is scheduled to bring the page from disk into a buffer allocated from the global free list. After the page is brought into memory, proceed as in case 2.

Note that the local replacement policies associated with file instances do not cause actual swapping of pages. Their real purpose is to maintain the image of a query's "working set". Disk reads and writes are issued by the mechanism that maintains the global table and the global free list.

The load controller is activated when a file is opened or closed. Immediately after a file is opened, the load controller checks whether $\sum_i \sum_j l_{ij} < N$ for all active queries **i** and their file instances **j**. If so, the query is allowed to proceed; otherwise, it is suspended and placed at the front of a waiting queue. When a file is closed, buffers associated with its locality set are released back to the global free list. The load controller then activates the first query on the waiting queue if this will not cause the above condition to be violated.

What remains to be described is how the QLSM is used to select local replacement policies and estimate sizes for the locality sets of each file instance.

Straight Sequential (SS) References

For SS references the locality set size is obviously 1. When a requested page is not found in the buffer, the page is fetched from disk and overwrites whatever is in the buffer.

Clustered Sequential (CS) References

For CS references, if possible, all members of a cluster (i.e. records with the same key) should be kept in memory. Thus, the locality set size equals the number of records in the largest cluster divided by the blocking factor (i.e. the number of records per page). Provided that enough space is allocated, FIFO and LRU both yield the minimum number of page faults.

Looping Sequential (LS) References

When a file is being repeatedly scanned in an LS reference pattern, MRU is the best replacement algorithm. It is beneficial to give the file as many buffers as possible, up to the point where the entire file can fit in memory. Hence, the locality set size corresponds to the total number of pages in the file.

Independent Random (IR) References

When the records of a file are being randomly accessed, say through a hash table, the choice of a replacement algorithm is immaterial since all the algorithms perform equally well [King71]. Yao's formula [Yao77], which estimates the total number of pages referenced **b** in a series of **k** random record accesses, provides an (approximate) upper bound on the locality set size. In those cases where page references are sparse, there is no need to keep a page in memory after its initial reference. Thus, there are two reasonable sizes for the locality set, **l** and **b**, depending on the likelihood that each page is re-referenced. For exam-

ple, we can define $r = \frac{k\ b}{b}$ as the **residual value** of a page. The locality set size is 1 if $r \sim \beta$, and **b** otherwise; where β is the threshold above which a page is considered to have a high probability to be re-referenced.

Clustered Random (CR) References

A CR reference is similar to that of a CS reference. The only difference is, in a CR reference, records in a "cluster" are not physically adjacent, but randomly distributed over the file. The locality set size in this case can be approximated by the number of records in the largest cluster[6].

Straight Hierarchical (SH), H/SS, and H/CS References

For both SH and H/SS references each index page is traversed only once. Thus the locality set size of each is 1 and a single buffer page is all that is needed. The discussion on CS references is applicable to H/CS references, except that each member in a cluster is now a key-pointer pair rather than a data record.

Looping Hierarchical (LH) References

In an LH reference, an index is repeatedly traversed from the root to the leaf level. In such a reference, pages near the root are more likely to be accessed than those at the bottom [Reit76]. Consider a tree of height **h** and with a fan-out factor **f**. Without loss of generality, assume the tree is complete, i.e. each non-leaf node has **f** sons. During each traversal from the root at level 0 to a leaf at level **h**, one out of the f^i pages at level i is referenced. Therefore pages at an upper level (which are closer to the root) are more important than those at a lower level. Consequently, an ideal replacement algorithm should keep the active pages of the upper levels of a tree resident and multiplex the rest of the pages in a scratch buffer. The concept of "residual value" (defined for the IR reference pattern) can be used to estimate how many levels should be kept in memory. Let b_i be the number of pages accessed at level i as estimated by Yao's formula. The size of the locality set can be approximated by $(1 + \sum_{i=1}^{j} b_i) + 1$, where j is the largest i such that $\frac{k\ b_i}{b_i} > \beta$. In many cases, the root is perhaps the only page worth keeping in memory, since the fan-out of an index page is usually high. If this is true, the LIFO algorithm and 3-4 buffers may deliver a reasonable level of performance as the root is always kept in memory.

[6] A more accurate estimate can be derived by applying Yao's formula to calculate the number of distinct pages referenced in a cluster.

4. Evaluation of Buffer Management Algorithms

In this section, we compare the performance of the DBMIN algorithm with the hot set algorithm and four other buffer management strategies in a multiuser environment. The section begins by describing the methodology used for the evaluation. Next, implementation details of the six buffer management algorithms tested are presented. Finally, the results of some of our experiments are presented. For a more complete presentation of our results, the interested reader should examine [Chou85].

4.1. Performance Evaluation Methodology

There were three choices for evaluating the different buffer management algorithms: direct measurement, analytical modeling, and simulation. Direct measurement, although feasible, was eliminated as too computationally expensive. Analytic modeling, while quite cost-effective, simply could not model the different algorithms in sufficient detail while keeping the solutions to the equations tractable. Consequently, we choose simulation as the basis for our evaluation.

Two types of simulations are widely used [Sher73]: **trace driven simulations** which are driven by traces recorded from a real system, and **distribution driven simulations** in which events are generated by a random process with certain stochastic structure. A trace driven model has several advantages, including creditability and fine workload characterization which enables subtle correlations of events to be preserved. However, selecting a "representative" workload is difficult in many cases. Furthermore, it is hard to characterize the interference and correlation between concurrent activities in a multiuser environment so that the trace data can be properly treated in an altered model with a different configuration. To avoid these problems, we designed a **hybrid simulation model** that combines features of both trace driven and distribution driven models. In this hybrid model, the behavior of each individual query is described by a trace string, and the system workload is dynamically synthesized by merging the trace strings of the concurrently executing queries.

Another component of our simulation model is a simulator for database systems which manages three important resources: CPU, an I/O device, and memory. When a new query arrives, a load controller (if it exists) decides, depending on the availability of the resources at the time, whether to activate or delay the query. After a query is activated, it circulates in a loop between the CPU and an I/O device to compete for resources until it finishes. After a query terminates, another new query is generated by the workload model. An active query, however, may be temporarily suspended by the load

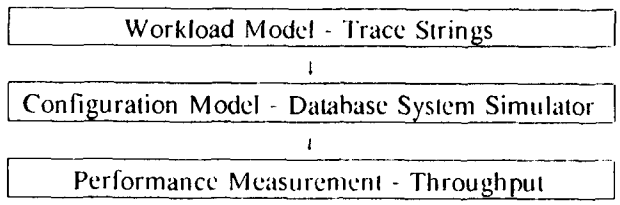
controller when the condition of over-loading is detected.

Although the page fault rate is frequently used to measure the performance of a memory management policy, minimizing the number of page faults in a multi-programmed environment does not guarantee optimal system behavior. Thus, throughput, measured as the average number of queries completed per second, was chosen as our performance metric. In the following sections, we shall describe three key aspects of the simulation model (Figure 1): workload characterization, configuration model, and performance measurement.

4.1.1. Workload Synthesis

The first step in developing a workload was to obtain single-query trace strings by running queries on the Wisconsin Storage System[7] (WiSS) [Chou83]. While WiSS supports a number of storage structures and their related scanning operations, WiSS does not directly support a high-level query interface; hence, the test queries were "hand coded". A synthetic database [Bitt83] with a well-defined distribution structure, was used in the experiments. Several types of events were recorded (with accurate timing information) during the execution of each query, including page accesses, disk I/O's, and file operations (i.e. opening and closing of files).

A trace string can be viewed as an array of event records, each of which has a tag field that identifies the type of the event. There are six important event types:



A Simulation Model for Database Systems
Figure 1

page read, page write, disk read, disk write, file open, and file close. Disk read and write events come in pairs bracketing the time interval of a disk operation[8]. The corresponding record formats in the trace string are:

Page read and write

page read / write	file ID	page ID	time stamp

Disk read and write

disk read / write	file ID	page ID	time stamp

File open

file open	file ID	locality set size	replacement policy

File close

file close	file ID

The time stamps originally recorded were real (elapsed) times of the system. For reasons to be explained later, **disk read** and **write** events were removed from the trace strings, and the time stamps of other events were adjusted accordingly. In essence, the time stamps in a modified trace string reflect the virtual (or CPU) times of a query.

Since accurate timing, on the order of 100 microseconds, is required to record the events at such detailed level, the tracing were done on a dedicated VAX-11/750 under a very simple operating kernel, which is designed for the CRYSTAL multicomputer system [DeWi84]. To reduce the overhead of obtaining the trace strings, events were recorded in main memory and written to a file (provided by WiSS) after tracing had ended.

In the multiuser benchmarking methodology described in [Bora84], three factors that affect throughput in a multiuser environment were identified: the number of concurrent queries[9], the degree of data sharing, and the query mix.

The number of concurrent queries (NCQ) in each of our simulation runs was varied from 1 to 32. To study the effects of data sharing, 32 copies of the test database were replicated. Each copy was stored in a separate portion of the disk. Three levels of data sharing were defined according to the average number of concurrent queries accessing a copy of the database:

(1) full sharing, all queries access the same database;

(2) half sharing, every two queries share a copy of the database; and

(3) no sharing, every query has its own copy.

[7] WiSS provides RSS-like [Astr76] capabilities in the UNIX environment.

[8] The version of WiSS used for gathering the trace strings does not overlap CPU and I/O execution.

[9] The term, multiprogramming level (MPL), was used in [Bora84]. However, since it is desirable to distinguish the external workload condition from the internal degree of multiprogramming, "number of concurrent queries" (NCQ) is used here instead. Using our definitions, MPL $\leq$ NCQ under a buffer manager with load control.

The approach to query mix selection used in [Bora84] is based on a dichotomy on the consumption of two system resources, CPU cycles and disk bandwidth. For this study, this classification scheme was extended to incorporate the amount of main memory utilized by the query (Table 1). After some initial testing, six queries were chosen as the base queries for synthesizing the multiuser workload (Table 2). The CPU and disk consumptions of the queries were calculated from the single-query trace strings, and the corresponding memory requirements were estimated by the hot set model (which are almost identical to those from the query locality set model). Table 3 contains a summary description of the queries.

At simulation time, a multiuser workload is constructed by dynamically merging the single-query trace strings according to a given probability vector, which describes the relative frequency of each query type. The trace string of an active query is read and processed, one event at a time, by the CPU simulator when the query is being served by the CPU. For a page read or write event, the CPU simulator advances the query's CPU time (according to the time stamp in the event record), and forwards the page request to the buffer manager. If the requested page is not found in the buffer, the query is blocked while the page is being fetched from the disk. The exact ordering of the events from the concurrent queries are determined by the behavior of the simulated system and the time stamps recorded in the trace strings.

Query Number	CPU Usage (seconds)	Number of Disk IO's	Hot Set Size (4K-pages)
I	.53	17	3
II	.67	99	3
III	2.95	53	5
IV	3.09	120	5
V	3.47	55	17
VI	3.50	138	24

Representative Queries
Table 2

4.1.2. Configuration Model

Three hardware components are simulated in the model: a CPU, a disk, and a pool of buffers. A round-robin scheduler is used for allocating CPU cycles to competing queries. The CPU usage of each query is determined from the associated trace string, in which detailed timing information has been recorded. In this respect, the simulator's CPU has the characteristics of a VAX-11/750 CPU. The simulator's kernel schedules disk requests on a first-come-first-serve basis. In addition, an auxiliary disk queue is maintained for implementing delayed asynchronous writes, which are initiated only when the disk is about to become idle.

The disk times recorded in the trace strings tend to be smaller than what they would be in a "real" environment for two reasons: (1) the database used in the tracing is relatively small; and (2) disk arm movements are

Query Type	CPU Demand	Disk Demand	Memory Demand
I	Low	Low	Low
II	Low	High	Low
III	High	Low	Low
IV	High	High	Low
V	High	Low	High
VI	High	High	High

Query Classification
Table 1

Query #	Query Operators	Selectivity	Access Path of Selection	Join Method	Access Path of Join
I	select(A)	1%	clustered index	-	-
II	select(B)	1%	non-clustered index	-	-
III	select(A) join B	2%	clustered index	index join	clustered index on B
IV	select(A') join B	10%	sequential scan	index join	non-clustered index on B
V	select(A) join B'	3%	clustered index	nested loops	sequential scan over B'
VI	select(A) join A'	4%	clustered index	hash join	hash on result of select(A)

A,B:10K tuples; A':1K tuples; B':300 tuples; 182 bytes per tuple.

Description of Base Queries
Table 3

usually less frequent on a single user system than in a multiuser environment. Furthermore, requests for disk operations are affected by the operating conditions and the buffer management algorithm used. Therefore, the disk times recorded were replaced by a stochastic disk model, in which a random process on disk head positions is assumed. In the disk simulator, the access time of a disk operation is calculated from the timing specifications of a Fujitsu Eagle disk drive [Fuji82]. On the average, it takes about 27.6 ms to access a 4K page.

The buffer pool is under the control of the buffer manager using one of the buffer management algorithms. However, the operating system can fix a buffer in memory when an I/O operation is in progress. The size of the buffer pool for each simulation run is determined by the formula:

$$8 \cdot \frac{\sum_i p_i t_i h_i}{\sum_i p_i t_i}$$

where p_i is the ith element of the query mix probability vector and t_i and h_i are the CPU usage and the hot set size of query i, respectively. The intent was to saturate the memory at a load of eight concurrent queries so that the effect of overloading on performance under different buffer management algorithms could be observed.

4.1.3. Statistical Validity of Performance Measurements

Batch means [Sarg76] was selected as the method for estimating confidence intervals. The number of batches in each simulation run was set to 20. Analysis of the throughput measurements indicates that many of the confidence intervals fell within 1% of the mean. For those experiments in which thrashing occurred, the length of a batch was extended to ensure that all confidence intervals were within 5% of the mean.

4.2. Buffer Management Algorithms

Six buffer management algorithms, divided into two groups, were included in the experiments. The first group consisted of three simple algorithms: RAND, FIFO, and CLOCK. They were chosen because they are typical replacement algorithms and are easy to implement. It is interesting to compare their performance with that of the more sophisticated algorithms to see if the added complexity of these algorithms is warranted. Beside DBMIN, WS (the working set algorithm), and HOT (the hot set algorithm) were included in the second group. WS is one of the most efficient memory policies for virtual memory systems [Denn78]. It is intriguing to know how well it performs when applied to database systems. The hot set algorithm was chosen to represent the algorithms that have previously been proposed for database systems.

All the algorithms in the first group are global algorithms in the sense that the replacement discipline is applied globally to all the buffers in the system. Common to all three algorithms is a global table that contains, for each buffer, the identity of the residing page, and a flag indicating whether the buffer has an I/O operation in progress. Additional data structures or flags may be needed depending on the individual algorithm. Implementations of RAND and FIFO are typical, and need no further explanation. The CLOCK algorithm used in the experiments gives preferential treatment to dirty pages, i.e. pages that have been modified. During the first scan, an unreferenced dirty page is scheduled for writing, whereas an unreferenced clean page is immediately chosen for replacement. If no suitable buffer is found in the first complete scan, dirty and clean pages are treated equally during the second scan. None of the three algorithms has a built-in facility for load control. However, we will investigate later how a load controller may be incorporated and what its effects are on the performance of these algorithms.

The algorithms in the second group are all local policies, in which replacement decisions are made locally. There is a local table associated with each query or file instance for maintaining its resident set. Buffers that do not belong to any resident set are placed in a global LRU list. To allow for data sharing among concurrent queries, a global table, similar to the one for the global algorithms, is also maintained by each of the local algorithms in the second group. When a page is requested, the global table is searched first, and then the appropriate local table is adjusted if necessary. As an optimization, an asynchronous write operation is scheduled whenever a dirty page is released back to the global free list. All three algorithms in the second group base their load control on the (estimated) memory demands of the submitted queries. A new query is activated if there is sufficient free space left in the system. On the other hand, an active query is suspended when over-commitment of main memory has been detected. We adopted the deactivation rule implemented in the VMOS operating system [Foge74] in which the faulting process (i.e. the process that was asking for more memory) is chosen for suspension[10]. In the following section, we discuss implementation decisions that are pertinent to each individual algorithm in the second group.

[10] We also implemented the deactivation rule suggested by Opderbeck and Chu [Opde74] which deactivates the process with the least accumulated CPU time. However, no noticeable differences in performance were observed.

Working Set Algorithm

To make WS more competitive, a two-parameter WS algorithm was implemented. That is, each process is given one of the two window sizes depending on which is more advantageous to it. The two window sizes, τ_1 = 10ms and τ_2 = 15ms, were determined from an analysis of working set functions on the single-query trace strings. Instead of computing the working set of a query after each page access, the algorithm re-calculates the working set only when the query encounters a page fault or has used up its current time quantum.

Hot Set Algorithm

The hot set algorithm was implemented according to the outline described in [Sacc82]. The hot set sizes associated with the base queries were hand-calculated according to the hot set model (see Table 2 above). They were then stored in a table, which is accessible to the buffer manager at simulation time.

DBMIN Algorithm

The locality set size and the replacement policy for each file instance were manually determined. They were then passed (by the program that implemented the query) to the trace string recorder at the appropriate points when the single-query trace strings were recorded. At simulation time, the DBMIN algorithm uses the information recorded in the trace strings to determine the proper resident set size and replacement discipline for a file instance at the time the file is opened.

4.3. Simulation Results

Although comparing the performance of the algorithms for different query types provides insight into the efficiency of each individual algorithm, it is more interesting to compare their performance under a workload consisting of a mixture of query types[11]. Three query mixes were defined to cover a wide range of workloads:

M1 - in which all six query types are equally likely to be requested;

M2 - in which one of the two simple queries (I and II) is chosen half the time;

M3 - in which the two simple queries have a combined probability of 75%.

The specific probability distributions for the three query mixes is shown in Table 4.

[11] The performance of the single query type tests are contained in [Chou85]. In general, the behavior of the algorithms for these tests are similar to the three mixes.

Query Mix	Type I	Type II	Type III	Type IV	Type V	Type VI
M1	16.67	16.67	16.67	16.67	16.66	16.66
M2	25.00	25.00	12.50	12.50	12.50	12.50
M3	37.50	37.50	6.25	6.25	6.25	6.25

(in %)

Composition of Query Mixes
Table 4

The first set of tests were conducted without any data sharing between concurrently executing queries. In Figure 2, the throughput for the six buffer management algorithms is presented for each mix of queries. In each graph, the x axis is the number of concurrent queries (NCQ) and the y axis is the throughput of the system measured in queries per second. The presence of thrashing for the three simple algorithms is evident[12]. A relatively sharp degradation in performance can be observed in most cases. RAND and FIFO yielded the worst performance, although RAND is perhaps more stable than FIFO in the sense than its curve is slightly smoother than that of FIFO. Before severe thrashing occurred, CLOCK was generally better than both RAND and FIFO.

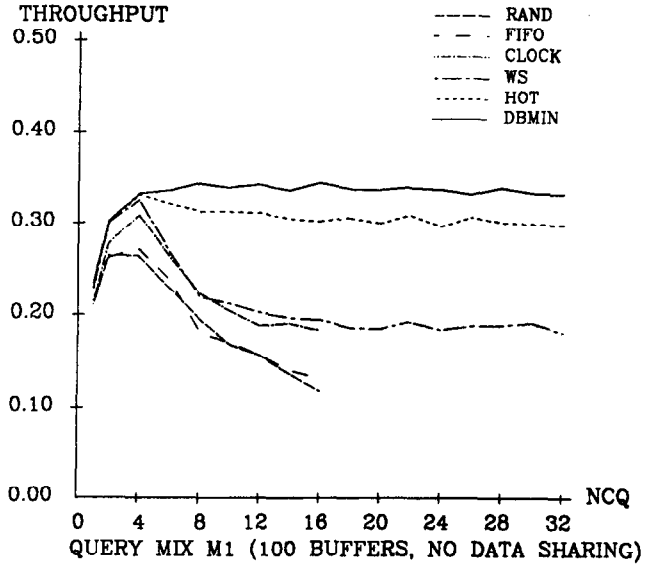
WS did not perform well because it failed to capture the main loops of the joins in queries V and VI. Its performance improved as the frequency of queries V and VI decreased. The efficiency of the hot set algorithm was close to that of DBMIN. When the system was lightly loaded, DBMIN was only marginally better than the rest of the algorithms. However, as the number of concurrent queries increased to 8 or more, DBMIN provided more throughput than the hot set algorithm by 7 to 13%[13] and the WS algorithm by 25 to 45%.

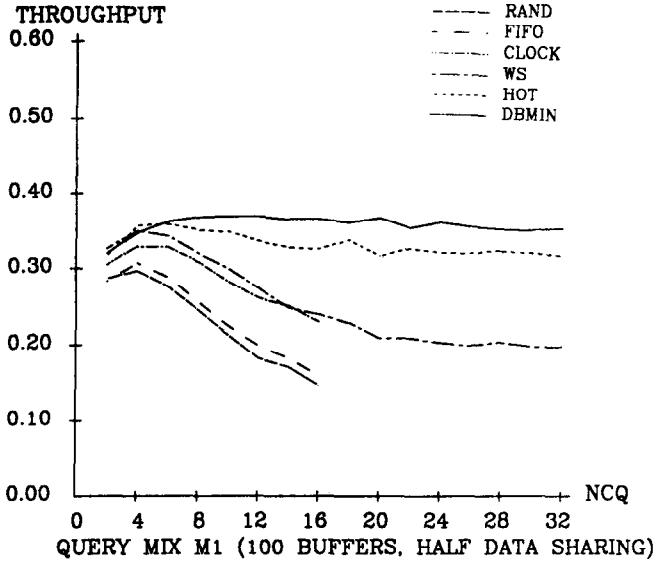
Effect of Data Sharing

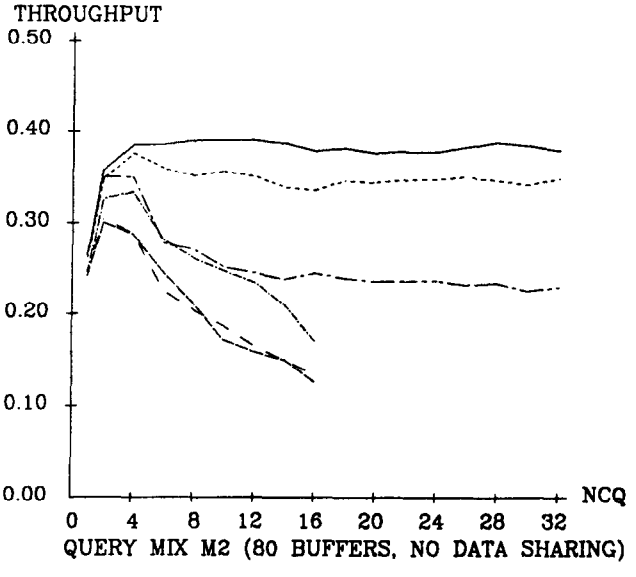
To study the effects of data sharing on the performance of the algorithms, two more sets of experiments, each with a different degree of data sharing, were conducted. The results are plotted in Figures 3 and 4. It can be observed that, for each of the algorithms, the throughput increases as the degree of data sharing increases. This reinforces the view that allowing for data sharing among concurrent queries is important in a multi-programmed database system [Reit76] [Bora84].

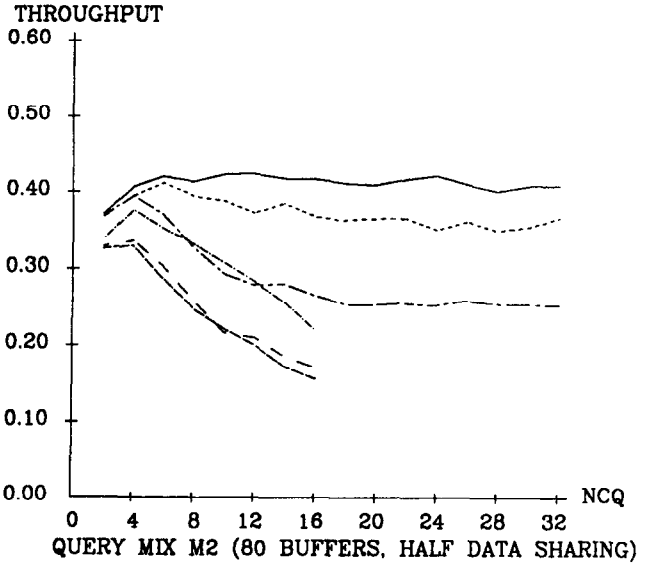
[12] Data points for the three simple algorithms were gathered only up to 16 concurrent queries as it is very time-consuming to gather throughput measurements with a ±5% confidence interval when the simulated system is trapped in a thrashing state.

[13] The percentages of performance difference were calculated relative to the better algorithm.









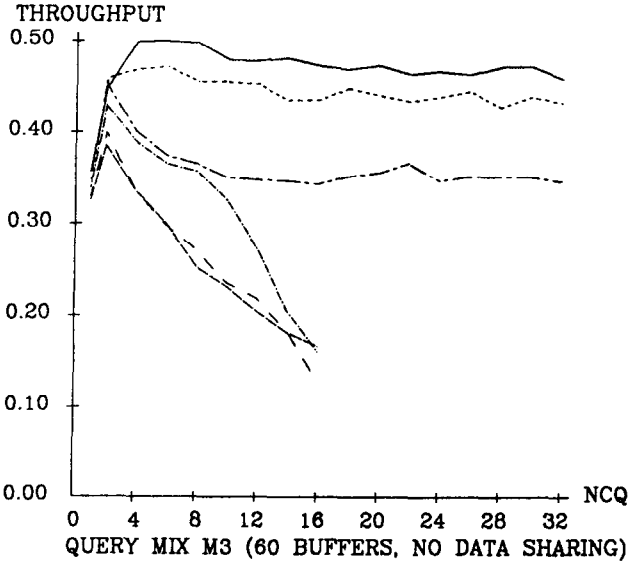


Figure 2

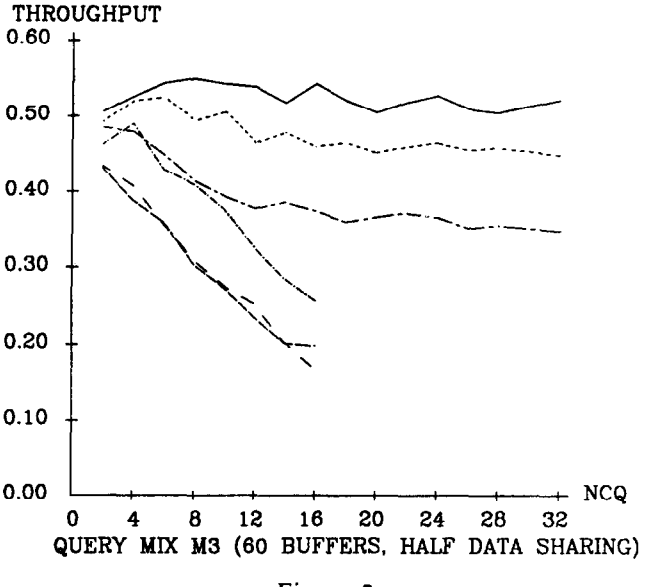


Figure 3

The relative performance of the algorithms for half data sharing is similar to that for no data sharing. However, it is not the case for full data sharing. For query mixes M1 and M2, the efficiencies of the different algorithms were close. Because every query accessed the same copy of the database, it was easy for any algorithm to keep the important portion of the database in memory. With no surprises, RAND and FIFO performed slightly worse than other algorithms due to their inherent deficiency in capturing locality of reference. For query mix M3, however, the performance of the different algorithms again diverged. This may be attributed to the fact that small queries dominated the performance for query mix M3. The "working" portion of the database becomes less distinct as many small queries are entering and leaving the system. (In contrast, the larger queries, which intensively access a limited set of pages over a relatively long period of time, played a more important role for query mixes M1 and M2.) Therefore, algorithms that made an effort to identify the localities performed better than those that did not.

Effect of Load Control

As was observed in the previous experiments, the lack of load control in the simple algorithms had led to thrashing under high workloads. It is interesting to find out how effective those algorithms will be when a load controller is incorporated. The "50% rule" [Lero76], in which the utilization of the paging device is kept busy about half the time, was chosen partly for its simplicity of implementation and partly because it is supported by empirical evidence [Denn76].

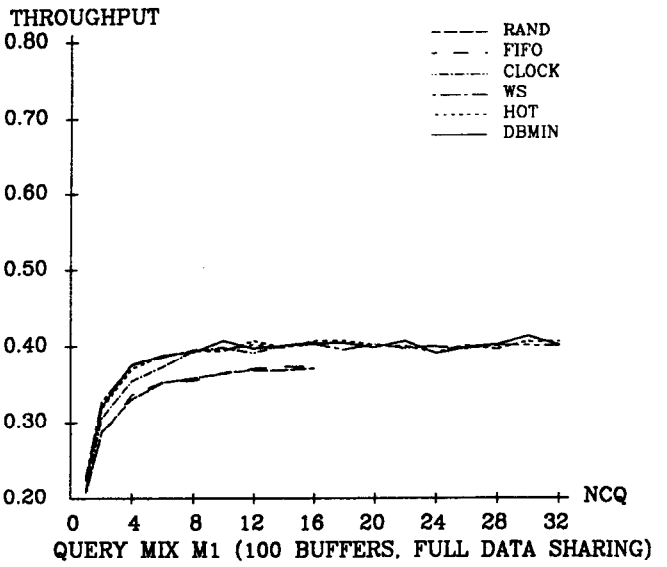
A load controller which is based on the "50% rule" usually consists of three major components:

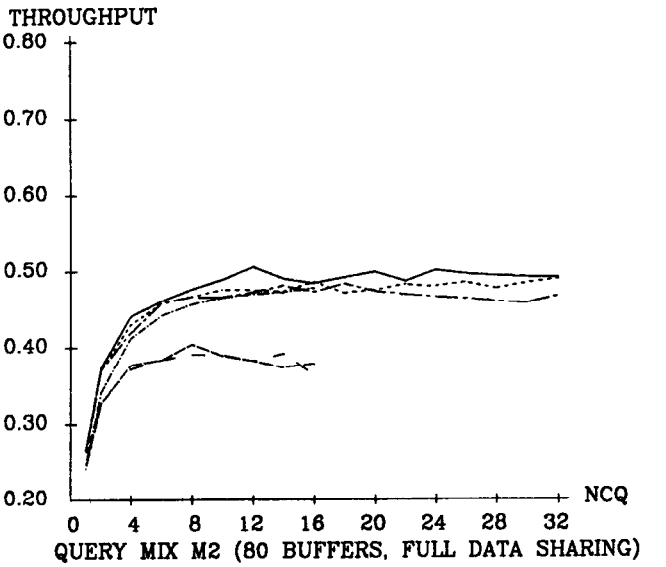
(1) an **estimator** that measures the utilization of the device,

(2) an **optimizer** that analyzes the measurements provided by the estimator and decides what load adjustment is appropriate, and

(3) a **control switch** that activates or deactivates processes according to the decisions made by the optimizer.

In Figure 5, the effects of a load control mechanism on the three simple buffer management algorithms is shown. A set of initial experiments established that throughput was maximized with a disk utilization of 87%. With load control, every simple algorithm in the experiments out-performed the WS algorithm. The performance of the CLOCK algorithm with load control came very close to that of the hot set algorithm. However, the results should not be interpreted literally. There are several potential problems with such a load





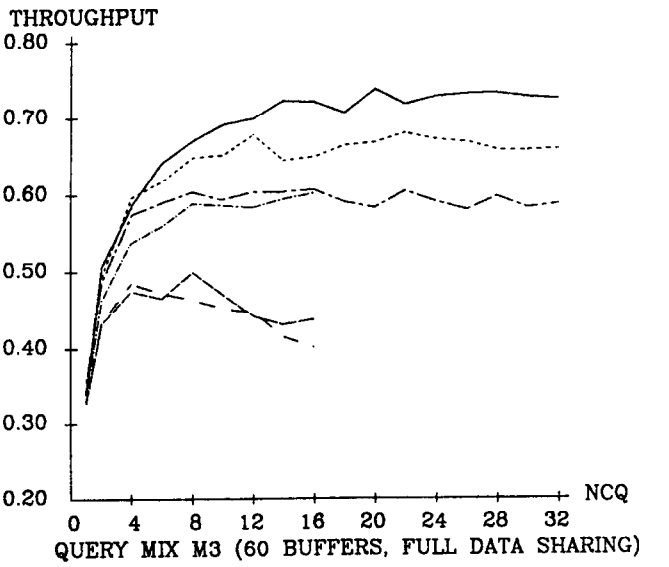


Figure 4

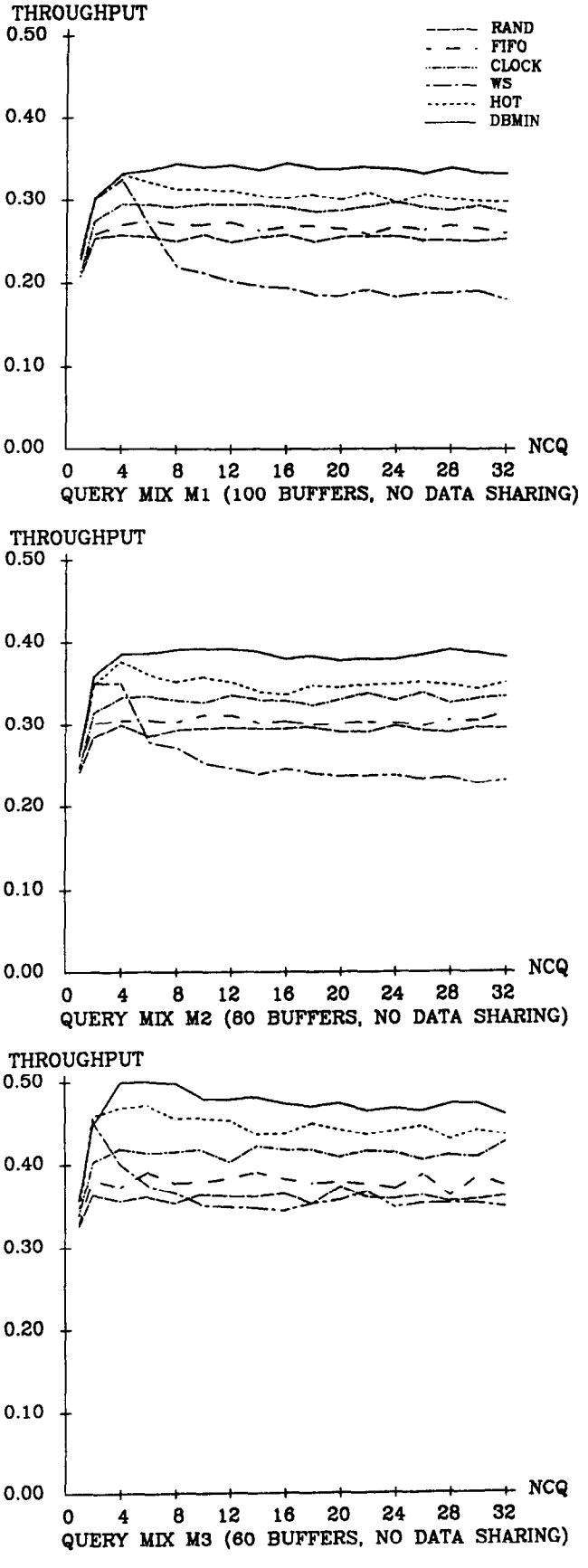


Figure 5

control mechanism which arise from the feedback nature of the load controller:

(1) Run-time overhead can be expensive if sampling is done too frequently. On the other hand, the optimizer may not respond fast enough to adjust the load effectively if analyses of the measurements are not done frequently enough.

(2) Unlike the predictive load controllers, a feedback controller can only respond after an undesirable condition has been detected. This may result in unnecessary process activations and deactivations that might otherwise be avoided by a predictive load control mechanism.

(3) A feedback load controller does not work well in an environments with a large number of small transactions which enter and leave the system before their effects can be assessed. This effect can be seen in Figure 5 as the percentage of small queries increases. Note that the so-called "small queries" (i.e. queries I and II) in our experiments still retrieve 100 tuples from the source relation. The disadvantages of a feedback load controller are likely to become even more apparent in a system with a large number of single-tuple queries.

5. Conclusions

In this paper we presented a new algorithm, **DBMIN**, for managing the buffer pool of a relational database management system. DBMIN is based on a new model of relational query behavior, the **query locality set model** (QLSM). Like the hot set model, the QLSM allows a buffer manager to predict future reference behavior. However, unlike the hot set model, the QLSM separates the modeling of referencing behavior from any particular buffer management algorithm. The DBMIN algorithm manages the buffer pool on a per file basis. The number of buffers allocated to each file instance is based on the locality set size of the file instance and will varies depending on how the file is being accessed. In addition, the buffer pool associated with each file instance is managed by a replacement policy that is tuned to how the file is being accessed.

We also presented a performance evaluation methodology for evaluating buffer management algorithms in a multiuser environment. This methodology employed a hybrid model that combines features of both trace driven and distribution driven simulation models. Using this model, we compared the performance of six buffer management algorithms. Severe thrashing was observed for the three simple algorithms: RAND, FIFO, and CLOCK. Although the introduction of a feedback load controller alleviated the problem, it created new potential problems. As expected, the three more sophisticated algorithms - WS, HOT, and

DBMIN - performed better than the simple algorithms. However, the WS algorithm did not perform as well as "advertised" for virtual memory systems [Denn78]. The last two algorithms, HOT and DBMIN, were successful in demonstrating their efficiency. In comparison, DBMIN provided 7 to 13% more throughput than HOT over a wide range of operating conditions for the tests conducted.

In [Chou85] we also examined the overhead associated with each of the WS, HOT, and DBMIN algorithms. Based on our analysis, the cost of the WS algorithm is higher than that of HOT unless the page fault rate is kept very low. In comparison, DBMIN is less expensive than both WS and HOT as less usage statistics need to be maintained.

Acknowledgements

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

6. References

[Astr76] Astrahan, M. M., et. al. System R: A Relational Approach to Database Management, ACM Transactions on Database Systems, vol. 1, no. 2, June 1976.

[Bitt83] Bitton, Dina, David J. DeWitt, and Carolyn Turbyfill, Benchmarking Database Systems: A Systematic Approach, Proceedings of the Ninth International Conference on Very Large Data Bases, November 1983.

[Blas77] Blasgen, M. W. and K. P. Eswaran, Storage and Access in Relational Data Base, IBM System Journals, no. 4, pp. 363-377, 1977.

[Bora84] Boral, Haran and David J. DeWitt, A Methodology For Database System Performance Evaluation, Proceedings of the International Conference on Management of Data, pp. 176-185, ACM, Boston, June 1984.

[Chou83] Chou, Hong-Tai, David J. DeWitt, Randy H. Katz, and Anthony C. Klug, Design and Implementation of the Wisconsin Storage System, Computer Sciences Technical Report #524, Department of Computer Sciences, University of Wisconsin, Madison, November 1983.

[Chou85] Chou, Hong-Tai, Buffer Management in Database Systems, Ph.D. Thesis, University of Wisconsin, Madison, 1985.

[DeWi84] DeWitt, David J., Raphael Finkel, and Marvin Solomon, The CRYSTAL Multicomputer: Design and Implementation Experience, Computer Sciences Technical Report #553, Department of Computer Sciences, University of Wisconsin, Madison, September 1984.

[Denn68] Denning, Peter J., The Working Set Model for Program Behavior, Communications of the ACM, vol. 11, no. 5, pp. 323-333, May 1968.

[Denn76] Denning, Peter J., Kevin C. Kahn, Jacques Leroudier, Dominique Potier, and Rajan Suri, Optimal Multiprogramming, Acta Informatica, vol. 7, no. 2, pp. 197-216, 1976.

[Denn78] Denning, Peter J., Optimal Multiprogrammed Memory Management, in Current Trends in Programming Methodology, Vol.III Software Modeling, ed. Raymond T. Yeh, pp. 298-322, Prentice-Hall, Englewood Cliffs, 1978.

[Effe84] Effelsberg, Wolfgang and Theo Haerder, Principles of Database Buffer Management, ACM Transactions on Database Systems, vol. 9, no. 4, pp. 560-595, December 1984.

[Fern78] Fernandez, E.B., T. Lang, and C. Wood, Effect of Replacement Algorithms on a Paged Buffer Database System, IBM Journal of Research and Development, vol. 22, no. 2, pp. 185-196, March 1978.

[Foge74] Fogel, Marc H., The VMOS Paging Algorithm, a Practical Implementation of the Working Set Model, ACM Operating System Review, vol. 8, January 1974.

[Fuji82] Fujitsu, Limited, M2351A/AF Mini-Disk Drive CE manual, 1982.

[Kapl80] Kaplan, Julio A., Buffer Management Policies in a Database Environment, Master Report, UC Berkeley, 1980.

[King71] King, W. F. III, Analysis of Demand Paging Algorithms, in Proceedings of IFIP Congress (Information Processing 71), pp. 485-490, North Holland Publishing Company, Amsterdam, August 1971.

[Lang77] Lang, Tomas, Christopher Wood, and Ieduardo B. Fernandez, Database Buffer Paging in Virtual Storage Systems, ACM Transactions on Database Systems, vol. 2, no. 4, December, 1977.

[Lero76] Leroudier, J. and D. Potier, Principles of Optimality for Multi-Programming, Proceedings of the international Symposium on Computer Performance Modeling, Measurement, and Evaluation, ACM SIGMETRICS (IFIP WG. 7.3), pp. 211-218, Cambridge, March 1976.

[Nybe84] Nyberg, Chris, Disk Scheduling and Cache Replacement for a Database Machine, Master Report, UC Berkeley, July, 1984.

[Opde74] Opderbeck, Holger and Wesley W. Chu, Performance of the Page Fault Frequency Replacement Algorithm in a Multiprogramming Environment, in Proceedings of IFIP Congress, Information Processing 74, pp. 235-241, North Holland Publishing Company, Amsterdam, August 1974.

[Reit76] Reiter, Allen, A Study of Buffer Management Policies For Data Management Systems, Technical Summary Report # 1619, Mathematics Research Center, University of Wisconsin-Madison, March, 1976.

[Sacc82] Sacco, Giovanni Maria and Mario Schkolnick, A Mechanism For Managing the Buffer Pool In A Relational Database System Using the Hot Set Model, Proceedings of the 8th International Conference on Very Large Data Bases, pp. 257-262, Mexico City, September 1982.

[Sacc85] Sacco, Giovanni Maria and Mario Schkolnick, Buffer Management in Relational Database Systems, To appear in ACM Transactions on Database Systems.

[Sarg76] Sargent, Robert G., Statistical Analysis of Simulation Output Data, Proceedings of ACM Symposium on Simulation of Computer Systems, August 1976.

[Sher73] Sherman, Stephen W. and J.C. Browne, Trace Driven Modeling: Review and Overview, Proceedings of ACM Symposium on Simulation of Computer Systems, pp. 201-207, June 1973.

[Sher76a] Sherman, Stephen W. and Richard S. Brice, I/O Buffer Performance in a Virtual Memory System, Proceedings of ACM Symposium on Simulation of Computer Systems, pp. 25-35, August, 1976.

[Sher76b] Sherman, Stephen W. and Richard S. Brice, Performance of a Database Manager in a Virtual Memory System, ACM Transactions on Database Systems, vol. 1, no. 4, December 1976.

[Ston76] Stonebraker, Michael, Eugene Wong, and Peter Kreps, The Design and Implementation of INGRES, ACM Transactions on Database Systems, vol. 1, no. 3, pp. 189-222, September 1976.

[Ston81] Stonebraker, Michael, Operating System Support for Database Management, Communications of the ACM, vol. 24, no. 7, pp. 412-418, July 1981.

[Ston82] Stonebraker, Michael, John Woodfill, Jeff Ranstrom, Marguerite Murphy, Marc Meyer, and Eric Allman, Performance Enhancements to a Relational Database System, Initial draft of a paper which appeared in TODS, vol. 8, no. 2, June, 1983.

[Thor72] Thorington, John M. Jr. and David J. IRWIN, An Adaptive Replacement Algorithm for Paged Memory Computer Systems, IEEE Transactions on Computers, vol. C-21, no. 10, pp. 1053-1061, October 1972.

[Tuel76] Tuel, W. G. Jr., An Analysis of Buffer Paging in Virtual Storage Systems, IBM Journal of Research and Development, pp. 518- 520, September, 1976.

[Yao77] Yao, S.B, Approximating Block Accesses in Database Organizations, Communications of the ACM, vol. 20, no. 4, pp. 260-261, April 1977.