Principles of Database Buffer Management

WOLFGANG EFFELSBERG
IBM Scientific Center, Heidelberg
AND
THEO HAERDER
University of Kaiserslautern

This paper discusses the implementation of a database buffer manager as a component of a DBMS. The interface between calling components of higher system layers and the buffer manager is described; the principal differences between virtual memory paging and database buffer management are outlined; the notion of referencing versus addressing of database pages is introduced; and the concept of fixing pages in the buffer to prevent uncontrolled replacement is explained.

Three basic tasks have to be performed by the buffer manager: buffer search, allocation of frames to concurrent transactions, and page replacement. For each of these tasks, implementation alternatives are discussed and illustrated by examples from a performance evaluation project of a CODASYL DBMS.

Categories and Subject Descriptors: D.4.2 [**Operating Systems**]: Storage Management—*storage hierarchies*; H.2.2 [**Database Management**]: Physical Design

General Terms: None

Additional Key Words and Phrases: Database, buffer management, locality, replacement algorithms

1. INTRODUCTION

Database management systems (DBMSs) use external magnetic devices (disks) for the storage of mass data. They offer low cost per bit and nonvolatility, which makes them indispensible in today's DBMS technology. However, under commercially available operating systems, data can only be manipulated (i.e., compared, inserted, modified, and deleted) in the main storage of the computer. Therefore, part of the database has to be loaded into a main storage area before manipulation and written back to disk after modification. A *database buffer* has to be maintained for purposes of interfacing main memory and disk.

Although several modern operating systems provide a main storage "cache" for their file systems, most DBMSs have their own buffer pools in the user address space—they do not use the OS file cache for various reasons (for a

Authors' addresses: W. Effelsberg, IBM Scientific Center, Tiergartenstrasse 15, 6900 Heidelberg, West Germany; T. Haerder, Fachbereich Informatik, University of Kaiserslautern, Postfach 3049, 6750 Kaiserslautern, West Germany.

detailed discussion, see, for example, [24]). In order to facilitate the exchange of data between disk and main storage, the database is divided into pages of equal size (generally 512 to 4096 bytes). The buffer consists of page frames of the same size. The number of frames in the buffer can be selected as a DBMS parameter, which remains constant during a DBMS session. Today buffer sizes vary from about 16 K to 12 M bytes. A typical buffer size may be assumed to be between 128 K and 256 K bytes.

Since a physical access to a database page on disk is much more expensive than an access to a database page in the buffer, the main goal of a database buffer manager is the minimization of physical I/O for a given buffer size. This goal has to be accomplished under certain restrictions resulting from the interface between the buffer manager and other DBMS components.

The purpose of this paper is to describe, in some detail, the main functions of a database buffer manager. In Section 2, its typical interface to the calling DBMS routines is investigated. Section 3 of the paper compares the applicability of different techniques for searching the buffer. Section 4 concentrates on the problem of allocating sufficient buffer space for concurrent transactions. In addition to the usual techniques, a new, page-type-oriented allocation algorithm is considered for use in the DBMS context. In Section 5, various page replacement algorithms are classified. The combination of classification criteria leads to the refinement of known algorithms. Section 6 presents an empirical study of the performance aspects of various buffer allocation algorithms in connection with page replacement algorithms. The results were gained using page reference strings of CODASYL DBMS applications. Section 7 describes some further buffer management problems related to a virtual OS environment and control of overload behavior. The final section summarizes the major aspects of DBMS buffer management.

2. INTERFACE AND OPERATIONS OF A DATABASE BUFFER MANAGER

In this paper the buffer manager is considered to be a component within the DBMS, having well-defined interfaces to other components. This section describes the interface between the buffer management component and the calling components of higher system layers, in order to provide a basic specification for the implementation of a buffer manager.

A request for a data object is performed as follows. The requestors have to be aware of the page boundaries and must use the DBMS catalog, index structures, and so on, to find the page numbers of pages they have to access. A page request P_i is issued by the FIX operator qualified by an optional update intent. As a result, P_i is located and fixed in the buffer to prevent replacement during its use. The address of the frame containing P_i is then passed back. Requestors can now execute machine instructions (i.e., COMPARE, MOVE, etc.) addressing data objects within P_i. Since only the requestors know when the addressing phase within P_i ends, they must call the buffer manager to again perform the UNFIX operation and make P_i eligible for replacement.

The addressability established by the FIX–UNFIX-mechanism is an important reason for choosing pages of equal size in a buffer with high replacement activity.

Since pages cannot be displaced deliberately, variable-size pages would cause heavy fragmentation problems.

Each request for a page is called a *logical reference*. For each logical reference the buffer manager has to perform the following actions:

- The buffer is searched and the page is located.
- If the page is not in the buffer, a buffer allocation strategy is used to determine the set of candidate buffer pages from which a "victim" for the requested page is to be taken. A page replacement algorithm then decides which of the buffer pages has to be replaced. Whenever the page selected for replacement has been modified, it has to be written back to disk before the new page is read into the buffer frame. Each access to a database page on disk is called a *physical reference*, and is one of the most expensive operations within a DBMS; it not only costs 25 to 50 ms of disk access time, but also involves 2000 to 5000 CPU instructions in most operating system environments.
- The requested page is fixed in the buffer by marking its buffer control block.
- The fact that the page has been referenced is recorded, since most replacement algorithms are based on the history of references to the buffer pages (e.g., LRU).
- Finally, the address of the buffer frame containing the requested page is passed to the calling DBMS component as a return parameter.

The sequence of logical references to database pages in temporal order (recorded as a sequence of page numbers) is called a *logical page reference string*. It describes the reference behavior of the DBMS, independent of the buffer size and replacement algorithm of the buffer manager. The logical reference behavior is determined by

- the types of transactions constituting the DBMS load (retrieval/update, direct/sequential access, etc.);
- the transaction mix (number and type of parallel transactions);
- the access path structures provided by the DBMS and its underlying data model, in the form selected in the internal database schema. For example, direct access on the external schema level may result in quite different logical page reference strings, depending on the existence of an index (e.g., a B*-tree) for the attribute specified in a query: Without an index, all records of a certain type (or all tuples of a relation) would have to be scanned, leading to a much longer reference string.

Logical references issued by a single transaction are independent of buffer size, whereas a global logical reference string characterizing the concurrent execution of multiple transactions is indirectly influenced by the size of the buffer. The global sequence of references is affected by intertransaction switches, which in turn are affected by specific transactions failing to find their data, which in turn are a function of buffer size. This kind of dependency is determined by many intrinsic implementation details of the DBMS and is very difficult to analyze.

Not every logical reference leads to a physical reference, but every physical reference is preceded by a logical reference. In contrast to logical references,

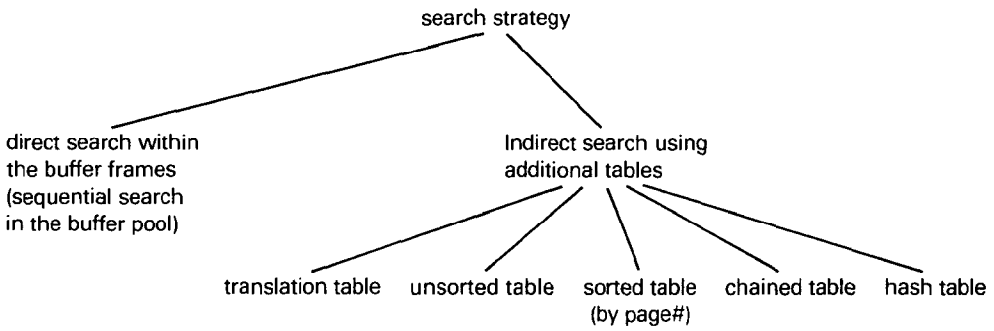


Fig. 1. Classification of search alogrithms for a database buffer manager.

physical references are strongly influenced by the size of the database buffer and the page replacement algorithm of the buffer manager. The same string of logical references can result in quite different physical reference strings under different replacement algorithms. Since physical references are expensive, the optimization of the page replacement algorithm is very important for the overall performance of the DBMS. Optimization means the minimization of the number of physical disk accesses for a typical transaction load, described by a logical reference string. As the characteristics of logical reference strings depend on the implementation details of a DBMS, the empirical results given in this paper should not be generalized. Our emphasis is on the basic principles of database buffer management and on the methods used in our evaluation rather than on the results themselves.

Having described the interface and operations of a database buffer manager, we now proceed to the implementation of single actions, as mentioned above—search within the buffer, buffer allocation, and page replacement.

3. SEARCHING THE BUFFER

Whenever a logical reference to a database page occurs, the buffer manager has to search the buffer. Since this is a frequent event, the search strategy implemented must be efficient. Figure 1 shows a classification scheme for possible search algorithms.

A *direct search* within buffer frames checks the page headers of all pages in the buffer sequentially. Since no assumptions are made concerning the ordering of buffer pages, the average number of pages searched in a buffer of size N will be $N/2$ in case of success and N in case of a fault. The main disadvantage of a direct search within buffer frames shows up when the DBMS is used under a virtual memory operating system. The buffer pool is then contained in the virtual address space of the DBMS. Searching page headers will result in the addressing of many distant parts of the virtual address space, causing frequent page faults and high paging overhead. Therefore, a DBMS running under a virtual memory operating system should use a table search technique, leading to higher locality in addressing behavior and thus reducing the page fault rate.

A *translation table* uses the page number as a displacement within the table. Hence, the table must provide D entries for a database containing D pages. It is

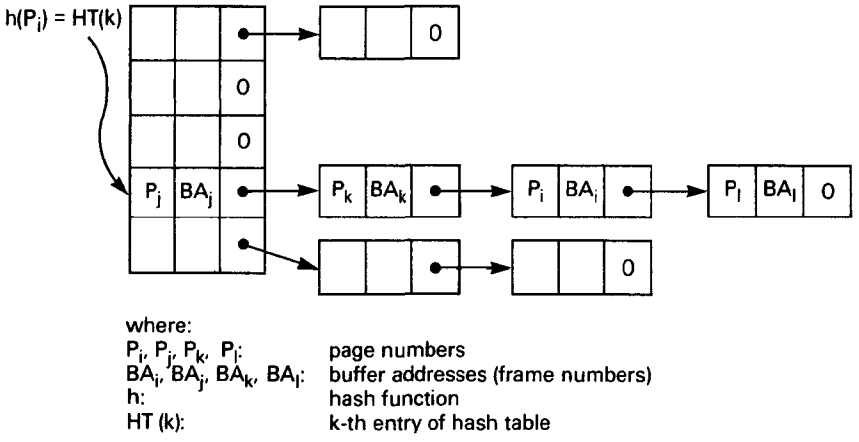


Fig. 2. Use of a hash table to improve buffer searching.

therefore restricted to very small databases. All other tables contain only N entries for a buffer of size N, independent of the size D of the database. The *unsorted and sorted tables* both require $N/2$ accesses on the average for a page found in the buffer. The sorted table reduces the number of accesses from N to $N/2$ for an unsuccessful sequential search and allows binary search techniques, but involves a much higher overhead when a table entry is inserted or deleted. By maintaining an index to the sorted table or by implementing the sorted table with a balanced binary tree, the search can be reduced to $\log_2 N$ accesses in either case; update costs, however, are even higher. A table with *chained* entries has two advantages over a compact table:

(1) update is less costly, since no entries have to be moved;
(2) the chaining sequence can be used to represent additional information. For example, table entries could be chained in LRU sequence, representing the replacement information for an LRU algorithm and speeding the buffer search when locality in the reference behavior is observed (e.g., when the probability of rereferencing recently used pages is high).

Since the most frequent operation in a page table is direct access using a page number, *hash* techniques can be used efficiently. The hash algorithm transforms a page number into a displacement within the page table, where the entry describing the page and its current position in the buffer can be found. Collisions can be resolved by chaining overflow entries to the "home" entry. With an appropriately sized hash table, the number n of entries searched per logical reference can be on the order of $1 < n < 1.2$. An example of such a hash table is given in Figure 2.

4. BUFFER ALLOCATION FOR CONCURRENT TRANSACTIONS

The buffer allocation algorithm of the buffer management component distributes the available buffer frames among the concurrent database transactions. It is closely related to the page replacement algorithm; in some cases, there is only one algorithm used both to distribute buffer frames to transactions and to make

replacement decisions (e.g., a global LRU algorithm). However, since the problem of allocating frames to transactions in an optimal way is logically different from the problem of selecting a page for replacement, buffer allocation algorithms are treated separately.

Before discussing specific algorithms, the reference behavior of database transactions has to be considered. In order to design optimal allocation and replacement algorithms, as much knowledge of the actual database reference characteristics as possible should be used. The following three basic properties of database reference strings distinguish them clearly from page reference strings of programs executing under a virtual memory operating system.

(1) Since database pages are a centralized resource shared by many users, the concurrent use of a page in the buffer by several transactions is quite frequent.

(2) Locality in the reference behavior of a DBMS is not necessarily due to the references of a single transaction; rather, the parallel execution of many transactions can increase the rereferencing probability across transaction boundaries (intertransaction locality, intratransaction sequentiality [18]).

(3) In some cases, the reference behavior of database transactions is predictable, being based on existing access path structures. Often, specific pages containing system tables, upper index levels, and so on, have a higher reference probability than do data pages. These identifiable, special-purpose pages can be treated in a special way when they are referenced.

Besides these general observations, it is important to know as much as possible about the reference behavior of the specific DBMS for which the buffer management component is to be implemented. On such a detailed level, different systems show different behavior. It is therefore more interesting to look at evaluation methods for reference strings than at the results for a specific DBMS in a specific database environment.

For storage allocation and page replacement algorithms, the most important property of a reference string is the *locality* of the reference behavior. *Locality* means that the probability of reference for recently referenced pages is higher than the average reference probability. If locality is observed in a reference string, most of the virtual memory allocation and replacement algorithms can be applied to buffer management; these algorithms were designed to keep the most recently referenced pages in main memory, since programs executing under virtual memory operating systems show high locality in their reference behavior [23].

Detailed information on locality is contained in an *LRU stack depth distribution* of the reference string, which shows the frequency of references to pages managed in the form of an LRU stack [22, 26]. The more the distribution is biased towards low stack depths, the higher is the locality in the string. Figure 3 shows two examples of LRU stack depth distributions calculated from the page reference strings of a CODASYL DBMS. The schema and transactions were taken from a school DB application. The schema consisted of 20 record types and 21 set types. The database contained approximately 330,000 record occurrences. The two reference strings discussed here correspond to session times of 30 to 40 minutes each; they contained 130,366 and 99,975 logical references. Figure 3a shows the stack depth distribution of a transaction load with a high percentage of short update transactions, whereas Figure 3b shows the distribution of a transaction

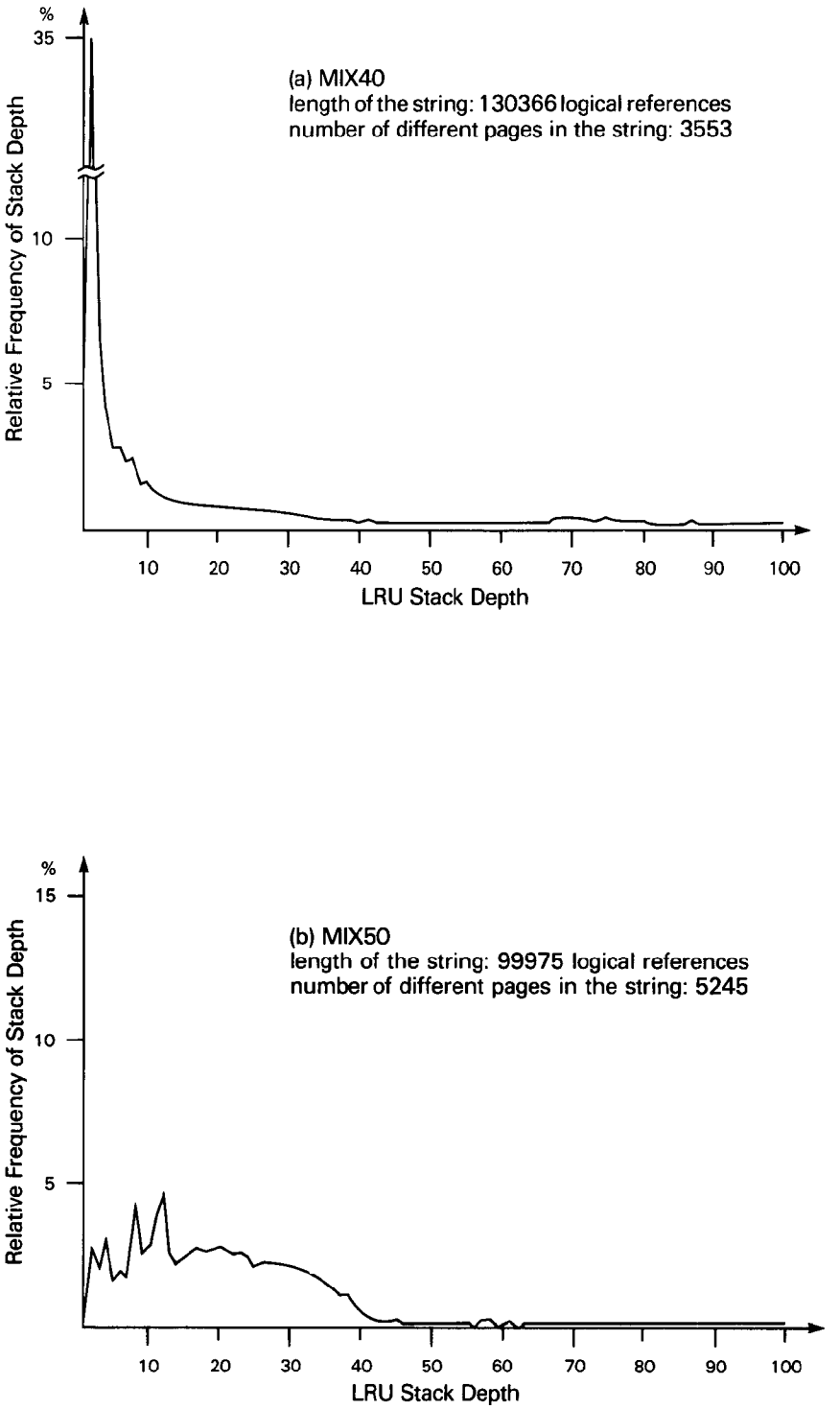


Fig. 3. LRU stack depth distributions of reference strings from a CODASYL DBMS.

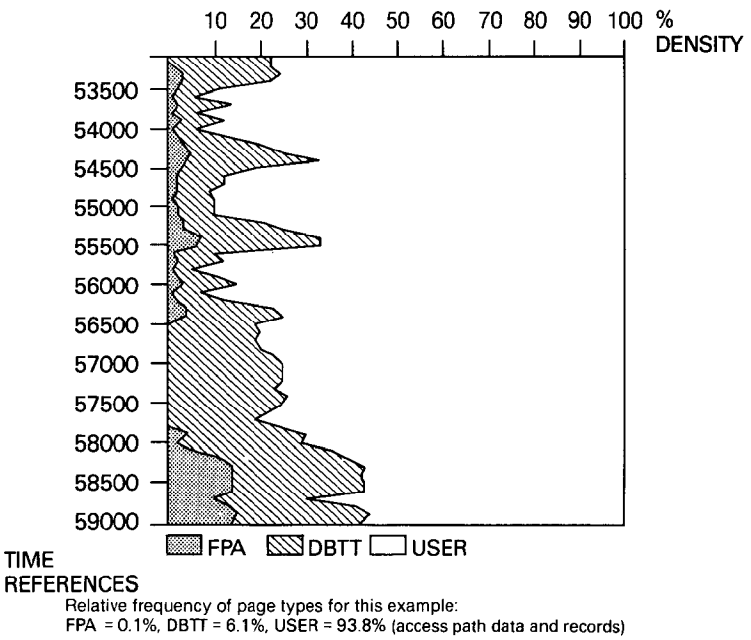


Fig. 4. Page reference density of a reference string from a CODASYL DBMS.

mix consisting of sequential retrieval transactions with only very few updates. In both cases, up to eight transactions ran concurrently. It is easily seen that the first mix (MIX40) had a much higher degree of locality than did the second (MIX50); both mixes are taken from [9].

In contrast to the reference behavior of programs under virtual memory operating systems, the highest reference probability is not found in stack depth 1 (containing the "youngest" page). This is due to the fact that the references observed are logical references to database pages and not addresses used by machine instructions. Since the "youngest" page in the buffer will be fixed in most cases, data objects within that page can be addressed without a new logical reference to the page. Another difference between the reference (or better: addressing) behavior of programs and the data reference behavior of database transactions is the probability of reference in stack positions 6 to 40 (approximately). For example, the first five stack positions may cover as much as 97 percent of all references of a program (data taken from [26]), whereas our MIX50 would find only 9.5 percent of its references in this range. Rereferencing in deeper stack positions (e.g., 6 to 40) is mainly caused by transactions being suspended for a certain time because they are blocked by concurrent transactions having exclusive access to the needed resources. Also, the DBMS stack depth distributions are not monotonically decreasing, for the same reason. Similar results have been reported by Fernandez, Lang, and Wood [10].

As mentioned before, the access path structures used by a specific DBMS lead to a higher reference probability for certain system pages, such as pages containing free-space data, address translation tables, root pages of B-trees, and so forth. An evaluation technique showing these effects is presented in Figure 4. The

reference density of pages of a certain type is defined as the *relative frequency* of references to such pages within a given reference interval. In our experiment, database pages were subdivided into FPA (free place administration), DBTT (database key translation table), and USER pages. In Figure 4, the percentage of references to these three classes of pages is shown over time. Although the number of system pages (FPA and DBTT) is much smaller than the number of USER pages in the database, the percentage of references to system pages can be as high as 45 percent. The considerable changes in the usage pattern depicted in Figure 4 are caused by changes in the transaction mix over the time period measured. Update transactions tend to access more system pages than do retrieval transactions.

Having found that there is locality in the reference behavior of database transactions, and that system pages have a much higher probability of reference than USER pages, we can now proceed to the discussion of buffer allocation algorithms.

4.1 Classification of Buffer Allocation Algorithms

Buffer allocation algorithms can be subdivided into local and global algorithms. An algorithm is local if it allocates buffer frames to a specific transaction without regarding the reference behavior of concurrent transactions. For a database buffer manager, local algorithms have to be supplemented with a mechanism for handling the allocation of buffer frames for shared pages, since concurrent access to the same database page is frequent. Local algorithms can be further subdivided into static and dynamic allocation. Under static allocation, the number of buffer frames belonging to a transaction remains constant during the lifetime of the transaction. A simple algorithm is the allocation of a fixed-size partition to each of the parallel transactions; more sophisticated algorithms could assign partitions of different sizes to different types of transactions, based on information known at the start time of a new transaction. Dynamic allocation assigns variable-size partitions to the transactions; each partition can grow and shrink according to the current reference behavior of the transaction. In contrast to local algorithms, global allocation algorithms consider not only the reference pattern of the transaction currently executing, but also the reference behavior of all other transactions. The allocation decision is based on data obtained from all transactions. In a DBMS context, a third class of allocation algorithms should also be considered. Whereas the terms "local" and "global" refer to transactions (processes) as owners of partitions, the database buffer could also be divided into parts containing a single type of page only. In our example, the buffer could have three partitions, for FPA, DBTT, and USER pages respectively. Again, partition sizes could be static or dynamic. A complete classification scheme is given in Figure 5.

The given classification scheme seems to reflect all buffer allocation algorithms that promise a successful application and are feasible with a reasonable amount of overhead.

The main disadvantage of static allocation (whether transaction oriented or page-type oriented) is its inflexibility in situations where the DBMS load changes frequently. Since the number of buffer frames allocated to a single transaction remains constant, static allocation is especially inefficient in an interactive

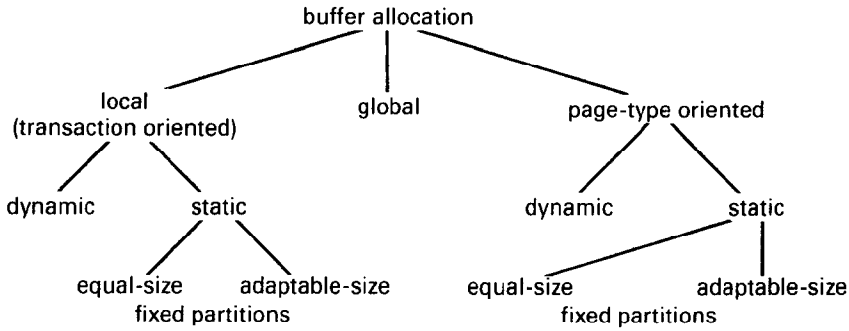


Fig. 5. Classification of buffer allocation algorithms.

environment where transactions can be blocked by long user think times. Because of its inflexibility, static allocation is not considered to be applicable in database buffer management.

When local dynamic allocation is applied, the size of the partition of a single transaction grows and shrinks with the transaction's changing need for buffer space. Only the current reference behavior of the transaction itself is taken into account by the allocation algorithm. When a buffer fault occurs, the allocation algorithm calculates the optimal partition size for the transaction. Depending on the reference history of the transaction, it may acquire an additional buffer frame, keep the partition size constant, or lose one or more frames. In the same way, partition sizes may vary under page-type-oriented dynamic allocation. For example, a database buffer could consist of a system part and a user part. New system pages would be placed in the system partition; new user pages in the user partition. The size of the partitions would vary with changing demand. With a fixed-size buffer, a second algorithm that selects pages for replacement has to be provided.

Whereas local algorithms consider only the reference behavior of a single transaction when calculating its optimal partition size, global algorithms take into account the reference behavior of all parallel transactions. All references to data pages are considered in the same way, independent of the transactions causing them. Since the DBMS buffer is considered to be of fixed size, global buffer allocation and the page replacement algorithm coincide. When a buffer fault occurs, one single algorithm decides which page has to be replaced; the decision is global. Depending on the owner of the corresponding buffer frame, the actual partition sizes of the transactions change automatically. (Page replacement algorithms are discussed in Section 5.)

Since static allocation is inefficient in a database environment and global allocation coincides with replacement algorithms, the only buffer allocation algorithms to be discussed in some detail here are local dynamic algorithms.

4.2 Dynamic Buffer Allocation Using Local Algorithms

The best-known dynamic storage allocation algorithm is Denning's working-set algorithm as defined in [6], which can be used to describe locality in the reference behavior of programs or database transactions. The working-set $W(t, \tau)$ of a transaction is defined as the set of pages referenced by the transaction during

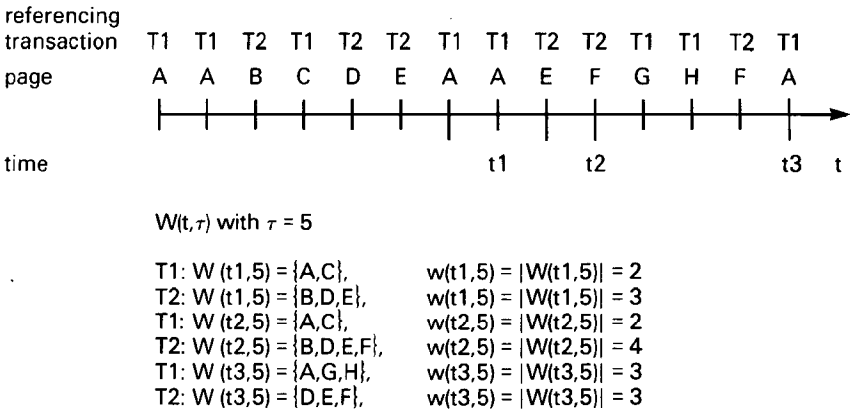


Fig. 6. Examples of working sets.

the last τ references at time t. τ is called window size. $w(t, \tau) = | W(t, \tau) |$ is the working-set size at time t. Figure 6 shows examples of working-sets and their sizes.

The average working-set size $\bar{w}(\tau)$ of a transaction can be used as a measure of locality. The higher the locality, the more references are made to elements previously referenced in the window of size τ, leading to a lower average working-set size $\bar{w}$.

Denning's working-set model can be used for the implementation of a buffer allocation algorithm, WS [17]. The basic principle is to keep the pages forming the working set of a transaction in the buffer, and to make all pages not belonging to any working set available for replacement. The window size τ has to be determined carefully so that the working set of a transaction contains just the minimum of pages needed for an efficient execution. In phases of high locality, the working set of a transaction shrinks, and the buffer frames freed will be available for reallocation to other transactions. A further refinement of the WS algorithm could be the assignment of different window sizes τ to different types of transactions, thereby establishing a priority system.

Note that the WS algorithm only decides whether or not a certain page in the buffer is available for replacement. It is irrelevant if the last reference to the page occurred $\tau + 1$ or $\tau + n$ ($n > 1$) references ago. Therefore, in addition to the buffer allocation algorithm WS, a page replacement algorithm is needed that selects one of the eligible pages for replacement when a buffer fault occurs. All replacement algorithms described in the next section can be used in combination with WS.

Implementing the WS algorithm means that whenever a buffer fault occurs, the working sets of all active transactions must be determined. Denning's proposals for an implementation were based on a hardware feature of virtual memory computers: Associated with each storage frame is a reference bit that is set by hardware whenever a page is referenced (addressed). Since a database buffer manager defines references to pages in a different way, these references have to be recorded by the software. A straightforward implementation is the following: Every active transaction has a reference counter TRC(T), which counts

TRC($T1$):	8	TRC($T2$):	6
LRC($T1$, A):	8	LRC($T2$, B):	1
LRC($T1$, C):	3	LRC($T2$, D):	2
LRC($T1$, G):	6	LRC($T2$, E):	4
LRC($T1$, H):	7	LRC($T2$, F):	6

Fig. 7. An implementation of the WS algorithm for buffer allocation. For $\tau = 5$, pages C and B are available for replacement.

all logical references of the transaction. Every page i in the buffer has, for every transaction using it, a field "last reference count" LRC(T, i). When transaction T references page i, TRC(T) is incremented and then copied into LRC(T, i). A buffer page i is available for replacement iff

$$\mathrm{TRC}(T) - \mathrm{LRC}(T, i) \geq \tau$$

for all transactions T using it. Figure 7 shows an example of this implementation, using the reference string of Figure 6.

Another dynamic storage allocation algorithm discussed in the literature is the page-fault-frequency algorithm (PFF). It uses the current interval between the last two page faults (which is related to the current page fault rate) for the allocation decision: As long as the actual page fault rate FA of a transaction is lower than a predefined maximum rate F, the transaction keeps its working set in the buffer (as under WS). When the actual fault rate FA is higher than F (determined by the fact that the interval between the last two page faults was less than $\tau' = 1/F$), a new buffer frame is allocated to the transaction, independent of its current working set. PFF is designed to guarantee a maximum fault rate of F for all transactions. (Further details may be found in the literature [5, 8]).

Further dynamic buffer allocation algorithms are proposed in the literature (e.g., a so-called WSCLOCK algorithm and an allocation based on a modified CLOCK algorithm; their complete descriptions and evaluations can be found in [1] and [4]).

In this section we have discussed the application of dynamic buffer allocation algorithms to transactions. Page-type-oriented dynamic allocations can be implemented in the same way. Hence, algorithms such as WS and PFF can be used with dynamic partitions in a transaction- or page-type-oriented database buffer.

5. REPLACEMENT ALGORITHMS FOR THE DATABASE BUFFER

If a logical reference to the buffer fails, a page in the buffer must be selected for replacement to make room for the requested page. Although a comparable problem arises in OS virtual memory management, there are some important differences:

- Any virtual memory page can be replaced at any time, because every reference is done by address translation hardware. However, in order to guarantee their addressability, some database pages can be fixed in the database buffer and are not eligible for replacement.
- A FIX–UNFIX interval of a database page, in which the calling system component issues n addressing operations to data objects within that page, is

treated as one database page reference. However, in OS memory management, every machine instruction touching the page is a page reference, resulting in substantially different replacement decisions.

The replacement algorithms presented here are considered to be orthogonal to the various allocation schemes presented in the preceding section. That is, they can be combined with each other in any manner. The set of candidate pages for replacement is determined by both the buffer allocation algorithm and the FIX–UNFIX mechanism. Replacement is needed in connection with

global allocation: within the entire buffer;
static allocation: within the respective partition;
dynamic allocation: in the set of eligible pages, that is, pages currently not contained in the working set of any transaction/page type.

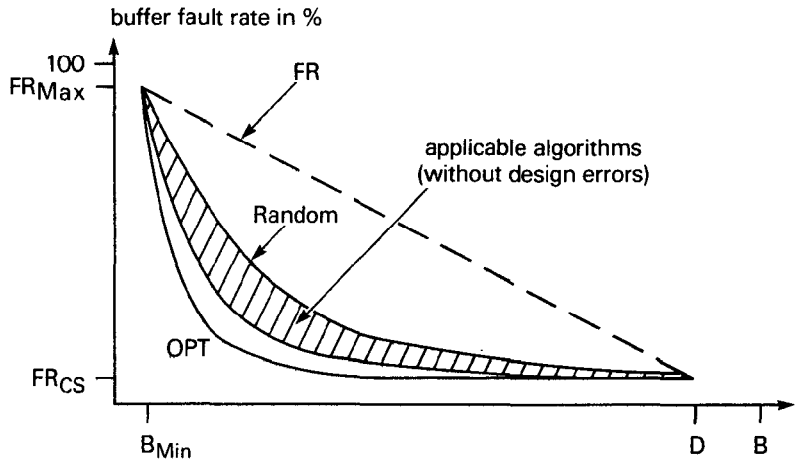
5.1 Fetching Pages

Replacement algorithms can be classified into prepaging and demand paging algorithms. Prepaging algorithms fetch not only the requested page, but also m additional pages that are physically close to the requested page, thus saving considerable access time, compared to $m + 1$ individual accesses. When sequential page references are expected, this algorithm can reduce the overall I/O costs substantially; prefetching unused pages, however, increases only the I/O overhead. Even worse, the replaced pages could be rereferenced. Unfortunately, physically contiguous pages are not necessarily the next ones referenced in logical sequence. In special cases, however, there may be some advantages in the exploitation of prepaging techniques. For detailed discussion, we refer readers to [16, 18, 21].

Demand paging algorithms fetch only the requested page when there is a page fault. Because of their importance for practical applications in buffer management, we classify and investigate them in a systematic manner.

In order to distinguish virtual memory page faults in the address space of the DBMS from page misses in the database buffer, we use the term *buffer fault* for the latter.

The goal of each replacement algorithm is the minimization of the buffer fault rate for a given buffer size and allocation. One can distinguish between algorithms that are practical and those that are not because they are based on knowledge of the future reference string. The latter are of theoretical interest. For example, Belady's well-known algorithm OPT [2,15], which replaces the buffer page with the longest future reference distance, can be used to derive a lower bound of the buffer fault rate for a given reference string. Clearly, the "absolute" upper bound for the buffer fault rate can be derived by the algorithm WORST, which replaces the buffer page with the shortest forward distance. An upper bound on the buffer fault rate for all practical replacement algorithms (which are assumed to be designed properly) should be achieved by the algorithm RANDOM, which does not use any knowledge about past reference behavior. Hence, for practical purposes, OPT and RANDOM are assumed to limit the realm of reasonable algorithms, giving a quality measure and some hints concerning the optimization potential with respect to a given replacement algorithm. In the following, we refer to practical algorithms as "applicable" algorithms.



B_{Min} = Minimal buffer size
D = Buffer holding entire database
FR_{CS} = Cold start buffer fault rate, i.e. the minimal fault rate for a given reference string due to the initially empty buffer (no. of different page/no. of references × 100%)

Fig. 8. Bounds of applicable algorithms given by RANDOM and OPT.

Figure 8 shows this relationship. With RANDOM referencing and RANDOM replacement, linear dependency of the buffer fault rate FR on buffer size B ($FR = 1 - B(1 - FR_{CS})/D$) could be expected. The locality of reference, however, causes the kind of nonlinear dependency for RANDOM replacement that is shown in Figure 8.

5.2 Systematic Description of Replacement Algorithms

Applicable algorithms replace the buffer page having the lowest probability of rereference. They usually rely on the characteristics of the past reference string in order to extrapolate future reference behavior. Their general assumption is that there is locality of reference; that is, recent reference behavior is a good indicator for the near future. Hence, the age and the references of a buffer page can be applied as suitable criteria to predict future reference behavior. By using logical references as units of time, the age of a page can be measured in an appropriate way.

We classify the various replacement algorithms by whether or not the following considerations are reflected in each algorithm's page selection decision:

- age since the first reference (fetch) to the page or since the last reference to the page;
- all references or only the most recent reference to the page.

Simple plausibility considerations lead to the conclusion that the exclusive use of only one of these criteria cannot guarantee an optimal replacement decision when there is locality of reference.

The algorithm FIFO (first-in, first-out) replaces the oldest buffer page. Independent of its reference frequency, the age of a page since the first reference is

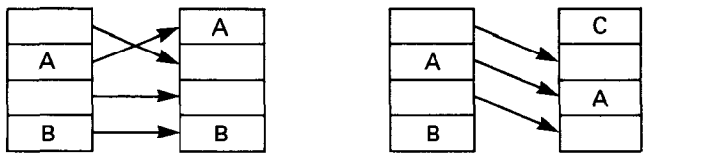


Fig. 9. The functioning of an LRU stack, buffer size = 4 pages.

the only decision criterion. Hence, FIFO is only appropriate for sequential access behavior. Figure 10a shows a common representation of FIFO, using a circular allocation of pages and a rotating pointer moved one step at every replacement. The pointer indicates the next page to be replaced.

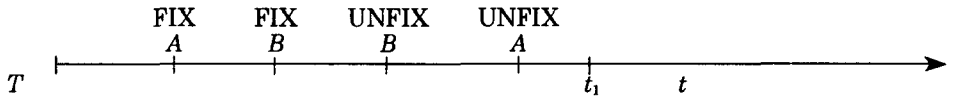
The algorithm LFU (least frequently used) uses only the second decision criterion, and replaces the buffer page with the lowest reference frequency. As shown in Figure 10c, reference counters (RC) are needed to record all references to a buffer page. When a page is fetched, the corresponding RC is initialized to 1; every rereference increments it by 1. When replacement is necessary, the buffer page with the smallest value of RC is chosen; a tie is resolved by some mechanism. In this strict LFU realization, the age of a page is not taken into account at all; pages with very high reference activity during a short interval can obtain such high RC values that they will never be displaced, even if they are never referenced again. For this reason, the pure LFU mechanism should not be implemented in a database environment. Using additional measures, the LFU concept can be made more appropriate, while losing its original characteristics.

All further algorithms to be discussed consider age as well as references. The widespread algorithm LRU replaces the buffer page that was least recently used, and can be explained easily by means of a so-called LRU stack, as shown in Figure 9.

The replacement decision is determined by which page is referenced and by the age of each buffer page since its most recent reference. The FIX mechanism for pages causes LRU to be optionally implemented by two versions, depending on how the term "used" is interpreted, as

- least recently referenced, or
- least recently unfixed.

The following scenario can help to clarify the difference.



At time t_1 with

- least recently referenced, page A is replaced;
- least recently unfixed, page B is replaced.

The version considering the UNFIX time is preferable in DBMS buffer management because FIX phases can last a very long time due to delays caused by a

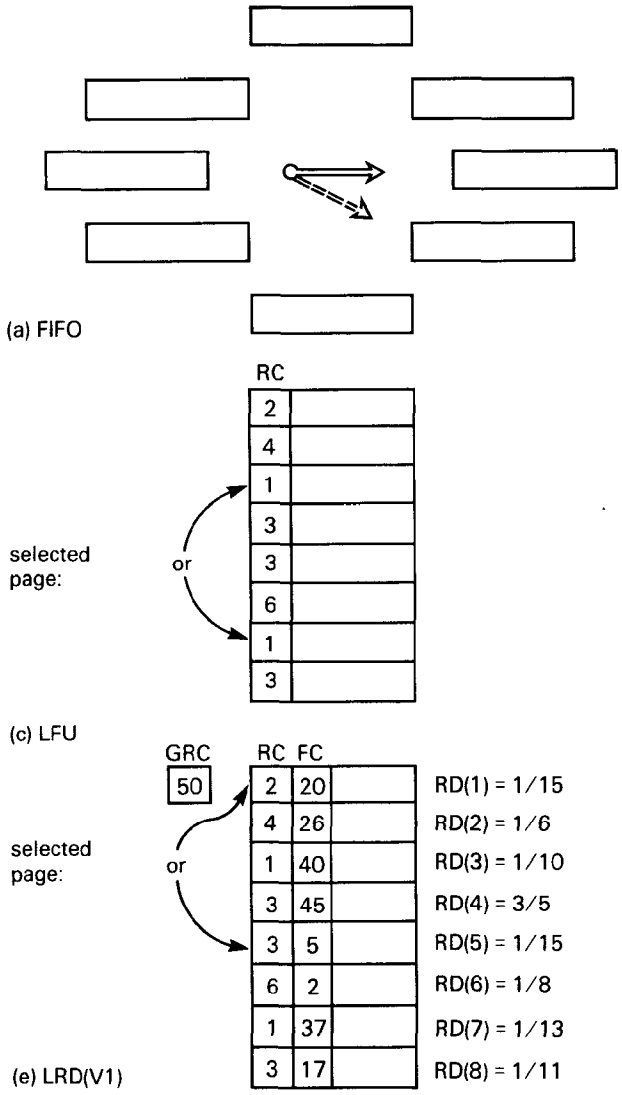


Fig. 10. Page replacement strategies (shown for 8 buffer pages).

transaction's blocking times and action interrupts. Thus, only this (UNFIX time) version guarantees the intended observation of the basic LRU idea.

The CLOCK algorithm attempts to simulate LRU behavior by means of a simpler implementation. As shown in Figure 10b, CLOCK is a modification of the FIFO mechanism (Figure 10a). A use-bit is added to every buffer page, indicating whether or not the page was referenced during the recent circulation of the selection pointer. The page to be replaced is determined by the stepwise examination of the use-bits. Encountering a 1-bit causes a reset to 0 and the move of the selection pointer to the next page. The first page found with a 0-bit is the victim for replacement. Another name for the CLOCK algorithm is

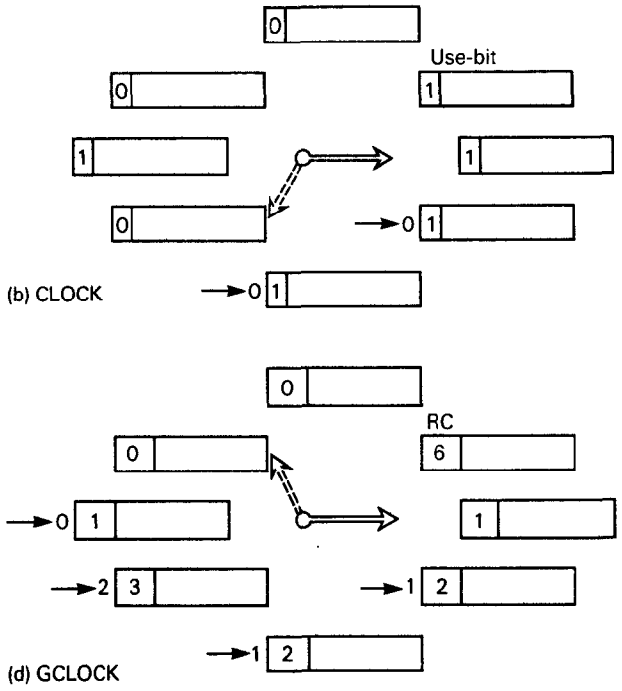


Fig. 10. (Con't.)

SECOND CHANCE, indicating that a page survives if rereferenced during a full circulation of the selection pointer.

In contrast to OS replacement algorithms, supported directly by special hardware features (e.g., use-bits), those for DBMS buffer management have to be fully accomplished in software. This necessity, however, offers more freedom with respect to the choice and evaluation of selection criteria. Thus, more complex algorithms are conceivable in a DBMS.

In combining the idea of LFU with the implementation of CLOCK, the basic version of GCLOCK (generalized CLOCK [21]) arises. The use-bit of a buffer page P_i is replaced by a reference counter (RC). References to P_i increment the corresponding counter RC(i). In the basic GCLOCK version, RC(i) is initialized to 1 upon first fetch of P_i and incremented by 1 at each rereference of P_i (see Figure 10d). When a buffer fault occurs, a circular search is initiated, decrementing stepwise the reference counters until the first with a value of 0 is found. This method accomplishes an essential improvement compared to a pure LFU algorithm. Nevertheless, this basic GCLOCK version tends to replace the youngest buffer pages, independent of their type and actual probability of rereference. To improve this undesired behavior, a number of variations can be introduced:

- initialize RC upon first fetch of a page with value >1;
- increment RC at each reference of a page versus set RC at each rereference to a fixed value;
- apply page-type or page-related weights.

The use of page weights (or virtual references) for the different page-types T_j (F_j for fetch and R_j for rereference) is appropriate to introduce knowledge about access paths and their traffic frequencies. For example, let T_1 be DBTT pages, T_2 FPA pages, T_3 index pages, and T_4 data pages. Then a fetch weight of 2 could be assigned to DBTT pages ($F_1 = 2$), a fetch weight of 1 to FPA and index pages ($F_2 = F_3 = 1$), while a weight of 0 could be assigned to data pages ($F_4 = 0$), expressing a low probability of rereference. A rereference could be treated by assigning the weights, $R = (2, 2, 2, 1)$.

This idea leads to the following versions V1 and V2 of GCLOCK, characterized by the way they handle the reference counter RC(i) related to page P_i of type T_j:

V1:	first reference (fetch)	: RC(i) := F_j
	each rereference	: RC(i) := RC(i) + R_j
V2:	first reference	: RC(i) := F_j
	each rereference	: RC(i) := R_j

When $F_j = 1$ and $R_j = 1$ for all j, V2 is equivalent to CLOCK, while V1 represents the basic version of GCLOCK. In V2, R_j should be $\geq F_j$; otherwise, an immediate rereference to a recently fetched page would decrease the value of the reference counter, an undesired effect.

In a real implementation, GCLOCK can be further refined—at the expense of increased overhead. It is generally possible to create a special version, called DGCLOCK, assigning dynamically calculated, page-related weights $F_j(t)$ and $R_j(t)$. Further implementation details such as threshold values or periodic decrease of the RCs are necessary to adapt a GCLOCK version to transitions in load characteristics.

GCLOCK represents a class of algorithms in which the different versions can be tailored to special applications and types of reference behavior by the appropriate choice of parameters. Its classification is difficult and necessarily fuzzy because of the variety of parameters involved.

The algorithms discussed (with the exception of FIFO) evaluate the age of a buffer page in some indirect way (via the latest reference). It appears to be promising to relate the actual number of references to a buffer page P_i counted in RC(i) to its age, defined as the number of elapsed references (to all buffer pages) since the first reference to P_i. The age of a page is measured in units of logical references, and can be determined as follows. Let the GRC (global reference counter) be the total number of logical references. For each buffer page P_i the time of its first reference (fetch) is FC(i). Hence, GRC–FC(i) is the reference interval of the age of P_i. Since both the age and RC are measured in units of logical references, they can be related to each other. By the use of simple division, the *reference density* RD(i) of P_i can be obtained. In our terminology, *reference frequency* always refers to an absolute number of references, whereas *reference density* means a frequency related to a reference interval (i.e., a relative frequency). This idea is materialized by the following algorithm (see Figure 10e):

$$\text{RD}(i) = \text{RC}(i)/(\text{GRC–FC}(i)) \qquad \text{where} \quad \text{GRC–FC}(i) \geq 1.$$

A buffer fault requires the determination of which buffer page has the lowest value for RD. GRC is incremented by the reference leading to the buffer fault,

before RD(i) is evaluated; a tie has to be resolved in some way. This algorithm, presented in its simplest version, can be generalized in various ways. Let us call the resulting class of algorithms LRD (least reference density) and the described version LRD(V1).

LRD(V1) determines the average reference density of a page. It assumes equidistant arrival of page references. High reference activity at the beginning of the reference interval keeps a page in the buffer much longer than desired, because the actual reference distribution within the interval is not known. The influence of older references on the selection decision—especially in the case of clustered arrivals—should be reduced. This goal, the reduction of the weight of references according to their actual age without the overhead of collecting additional page-related information, is achievable by the following LRD variant: After reference intervals of appropriate size, the reference counters RC of all buffer pages are reduced (e.g., by subtraction or division, using properly chosen constants). For example, the method to enforce some kind of "periodic aging" at the end of specific reference intervals IR could be chosen as follows:

LRD(V2): aging by subtraction:

$$\mathrm{RC}(i) = \begin{cases} \mathrm{RC}(i)-C1 & \text{if } \mathrm{RC}(i) - C1 \geq C2 \\ & \text{with } C1 > 0,\ C2 \geq 0 \\ C2 & \text{if } \mathrm{RC}(i) - C1 < C2 \end{cases}$$

aging by division:

$$\mathrm{RC}(i) = \mathrm{RC}(i)/C3 \quad \text{with } C3 > 1$$

$C1$, $C2$, and $C3$ are appropriately selected constants. The size of the reference interval IR for aging must also be selected carefully. In each algorithm counting reference frequencies, a number of modifications are conceivable (e.g., the use of page weights in case of fetch and/or rereference).

An overview of the discussed replacement algorithms is given in Figure 11, which attempts to classify them according to their parameters and they way age and reference are taken into consideration. Those algorithms that are candidates for an application in buffer management are emphasized.

Another important criterion to be considered in the replacement decision is the type of reference to a page, that is, whether a page is read only or modified. In general, it may be preferable to keep modified pages in the buffer longer (at least those with high probability of further updates), because their replacement is expensive (the page itself and the corresponding log information has to be written). On the other hand, overemphasizing this principle carries the danger of shrinking the active window for the read-only pages that are kept in the buffer. In a specific implementation, all algorithms have to be adapted to the particularities of the buffer interface (e.g., fixed pages are not displaceable and pages being modified have to be forced to disk at the end of the corresponding transaction when required by the logging mechanism).

With local or page-type-oriented buffer allocation, it is conceivable to combine various replacement algorithms, tailored to specific characteristics of the reference behavior. For example, four different reference types, apparently related to various page types, can be observed in the DBMS INGRES [24]. Hence, further

Name	*Parameter*	*Selection Criterion*
CLOCK	—	reference bit
LRU(UNFIX)	—	time (GRC) of last UNFIX
GCLOCK(V1)	Fj, Rj	value of reference counter RC
GCLOCK(V2)	Fj, Rj	value of reference counter RC
DGCLOCK	$Fj(t)$, $Rj(t)$	value of reference counter RC
LRD(V1)	RC, FC	reference density RD
LRD(V2)	IR, RC, $C1$, $C2$, or $C3$	value of manipulated RC

Consideration during selection decision		Age		
		No consideration	Since most recent reference	Since first reference
references	no consideration	RANDOM		FIFO
	most recent reference		LRU CLOCK GCLOCK(V2)	
	all references	LFU	GCLOCK(V1) DGCLOCK LRD(V2)	LRD(V1)

Fig. 11. Classification of replacement algorithms.

optimization should be possible by assigning appropriate replacement algorithms to a page-type-oriented buffer allocation scheme. Due to the nonprocedural requests of relational interfaces, transaction-oriented optimization of buffer replacement should be achievable because enough context information is given to the DBMS to predict a limited number of a transaction's future references [19].

6. EMPIRICAL STUDY OF REPLACEMENT STRATEGIES

As mentioned earlier, logical reference strings are DBMS-dependent; their characteristics are determined by implementation details, internal structure of system and user data, look-up sequences, and so forth. Therefore, it is impossible to derive general results from specific reference strings. Nevertheless, our empirical study attempts to evaluate the usefulness of buffer allocation and replacement strategies and aims to give an indication of how the various strategies compare to each other.

The large number of possible combinations of buffer allocation schemes and replacement algorithms prohibits an exhaustive study. Strategies such as GCLOCK and LRD offer especially many degrees of freedom, with their various versions and parameters. Optimal parameter values are difficult to determine and have to be tailored to a specific DBMS. Since such an optimizaton is not the goal of our study, we do not compare the effects of detailed parameter variations on these algorithms; rather, we compared various versions, with some simple

			MIX40	MIX50
Total number of pages (school-DB)			30,000	
Number of different pages in the string			3,553	5,245
Number of logical references			130,366	99,975
Number of page modifications			9,378	2,865
Number of pages being fixed[a]	maximum		11	10
	average		4.61	6.26
Percentage of references with FIX duration		= 1	71%	41%
		2–10	22%	41%
		>10	7%	18%
FIX duration (in logical references)	max.		1,786	—
	avg.		4.62	6.26
Percentage of pages of a given type		FPA	0.1%	
		DBTT	6.1%	
		USER	93.8%	
Percentage of references to page-types		FPA	0.9%	0.1%
		DBTT	9.4%	21.7%
		USER	89.7%	78.2%
Percentage of references to most frequently referenced pages ("hot spot" pages)		1.	12.6%	3.3%
		2.	9.8%	0.5%
		3.	4.8%	0.4%
Relative frequency distribution of references to the other pages		>0.9%	5	1
		0.9%–0.1%	195	293
		0.1%–0.03%	1,606	2,741
		<0.03%	1,747	2,208
Number of references to shared pages (concurrently fixed)[a]			1,175	359
cold start buffer fault rate in %			2.72	5.24
number of executed transactions	total		262	39
	parallel:	max.	8	8
		avg.	6.31	6.29

[a] measured with a buffer size of 128 pages

Fig. 12. Characteristics of the DB, transaction load and logical reference strings.

parameter combinations, in order to roughly evaluate their behavior and to give an idea of their relative usefulness.

The results of the empirical study reported in this section are based on two logical reference strings of different transaction loads, called MIX40 and MIX50 (see Section 4); their LRU stack-depth distributions are shown in Figure 3. Some additional characteristics of these strings are summarized in Figure 12.

A general buffer simulator consisting of PASCAL programs (and ASSEMBLER subroutines) of approximately 20,000 lines of code was written, providing various allocation schemes and replacement algorithms. The simulator is driven by a logical reference string; the buffer fault rate, as a function of buffer size, is produced as output. Optionally, a number of useful statistics (FIX duration, shared pages, etc.) can be obtained (see Figure 12). Emphasis is placed on the global buffer allocation scheme. Two local schemes are added for comparison. Page-type-oriented schemes are presented in a separate section.

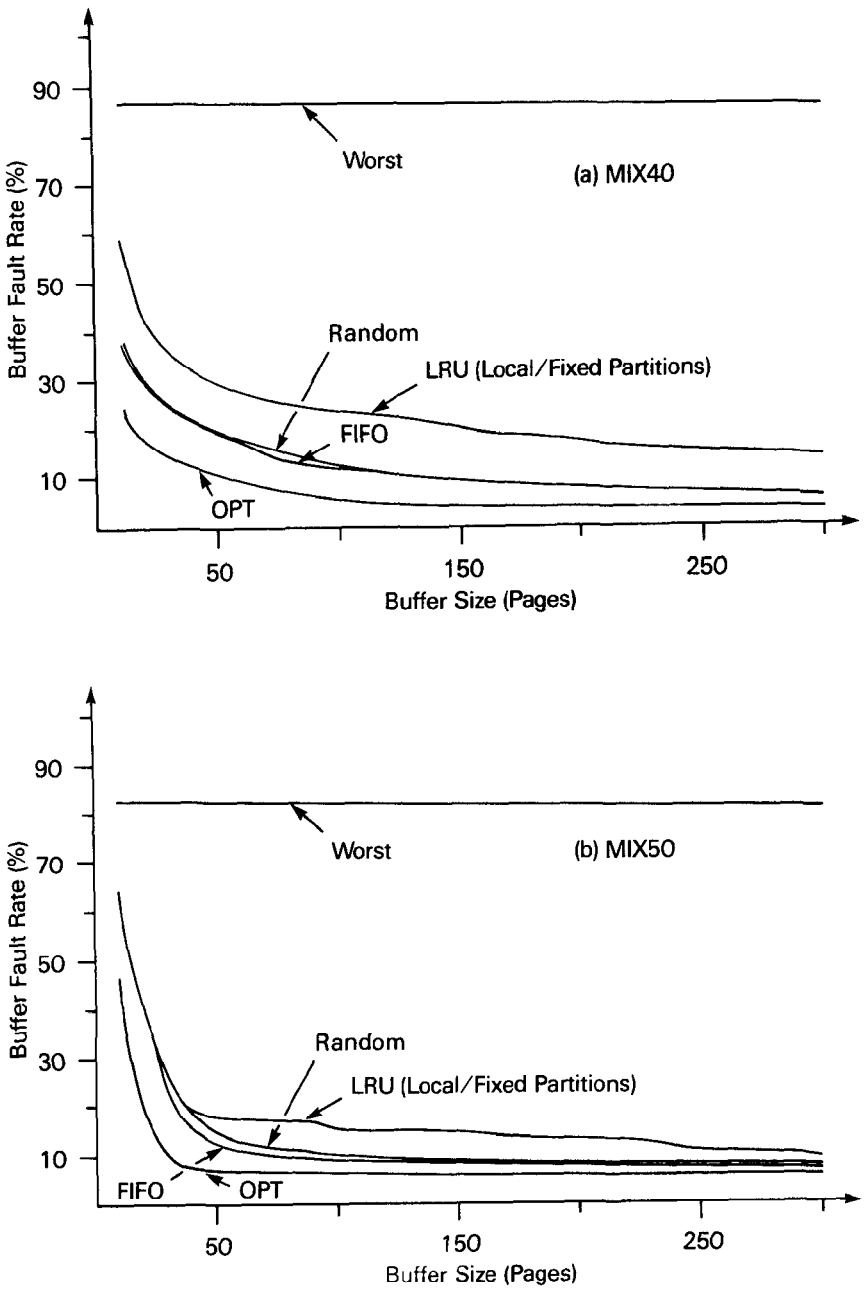


Fig. 13. The buffer fault rates of different replacement strategies.

6.1 Global Buffer Allocation

Figure 13 confirms the relationships illustrated in Figure 8. FIFO touches the bounds given by RANDOM, but fits into the expected range. The curves describing a local buffer allocation scheme with "almost" fixed partitions and LRU

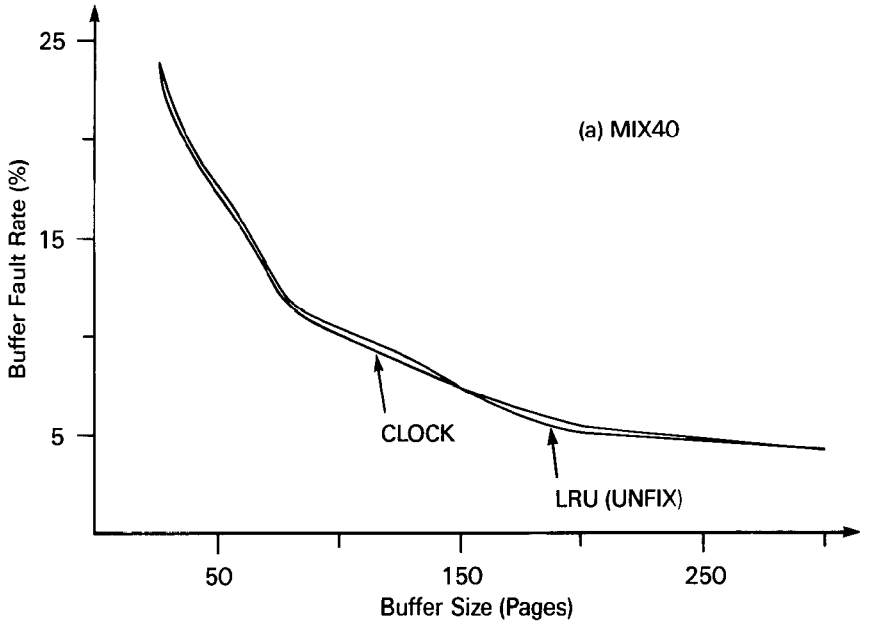


Fig. 14. The buffer fault rates of the CLOCK and LRU algorithms.

replacement are the simulation results of an existing buffer manager. Owing to a design error detected by these simulations, the dynamic partitioning mechanism behaved like a static one, resulting in a catastrophic buffer fault rate (often twice as high as RANDOM). This example can be taken as an illustration of the inflexibility of static buffer allocation strategies; it supports our judgment in Section 4.1. Having digested this unexpected surprise, we designed and simulated the strategy WORST to show the full range of "optimization" by using badly understood algorithms.

Figure 14 compares the buffer fault rates of the widely used LRU and CLOCK algorithms. LRU proved itself efficient in some operating systems as well as in DBMS buffer management (e.g., in System R). Figure 14 confirms the similarity of buffer fault rates between LRU and CLOCK, as stated elsewhere [20]. In our experiments, LRU (UNFIX) was slightly superior to LRU (REFERENCE) (not shown in Figure 14).

Figure 15 compares two GCLOCK algorithms. For GCLOCK(V2), the parameters are chosen in this example as follows:

- $F_3 = 5$ and $R_3 = 5$ for index pages,
- $F_j = 0$ and $R_j = 1$ for all other pages.

The DGCLOCK version applied in the simulation is used to illustrate the degrees of freedom of the GCLOCK technique, rather than as a proposal for implementation. The weight W_i (W_i is the fetch weight and also the rereferencing weight) assigned to page P_i is calculated dynamically, depending on the absolute number of its buffer faults BF_i and the average number of buffer faults $\overline{BF}$ over all pages: $\overline{BF}$ = (total number of buffer faults so far)/(number of distinct pages referenced).

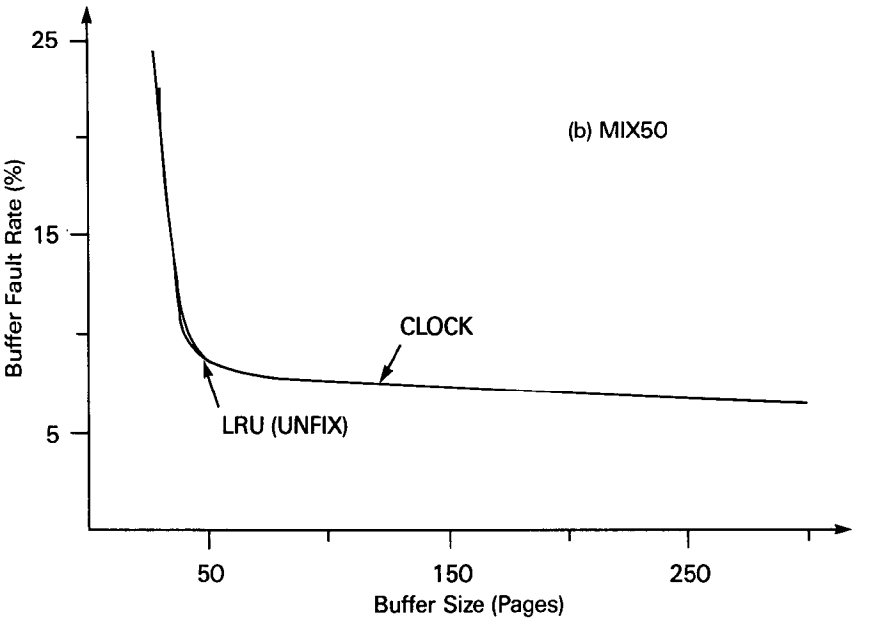


Fig. 14. (Con't.)

Hence, the weight is calculated as follows:

$$W_i = \begin{cases} BF_i - \overline{BF} & \text{if } BF_i > \overline{BF} \\ 0 & \text{otherwise} \end{cases}$$

As shown in Figure 15, a general recommendation cannot be made; DGCLOCK delivers very good results in a certain range, and worse results in others. According to the comparisons in Figures 17a and 18a, DGCLOCK is superior to all other replacement algorithms for a buffer size of 100 to 200 pages.

The behavior of two LRD algorithms is shown in Figure 16. LRD(V1) corresponds to the respective algorithm in Section 5.2. The parameters of LRD(V2) are chosen in this example as follows:

- reference interval IR (determining periodic aging) is 10;
- divisor constant $C3 = 2$.

Looking at Figure 16, the sophisticated algorithm LRD(V2) displays its superiority with respect to the simple version LRD(V1) over wide ranges; by better tailoring its parameters to the specific reference characteristics, further optimization seems to be achievable. The use of LRD(V2), based on the subtraction method, in the DBMS ADABAS underlines its value for DBMS buffer management.

The "best" algorithms of each class investigated in our empirical study are summarized with a wider buffer size range in order to facilitate comparison among all results. Unfortunately, the various curves had to be divided up into Figures 17 and 18 to avoid an overloaded representation, and at least allow for a clear separation of the inefficient algorithms. Comparable algorithms deserve

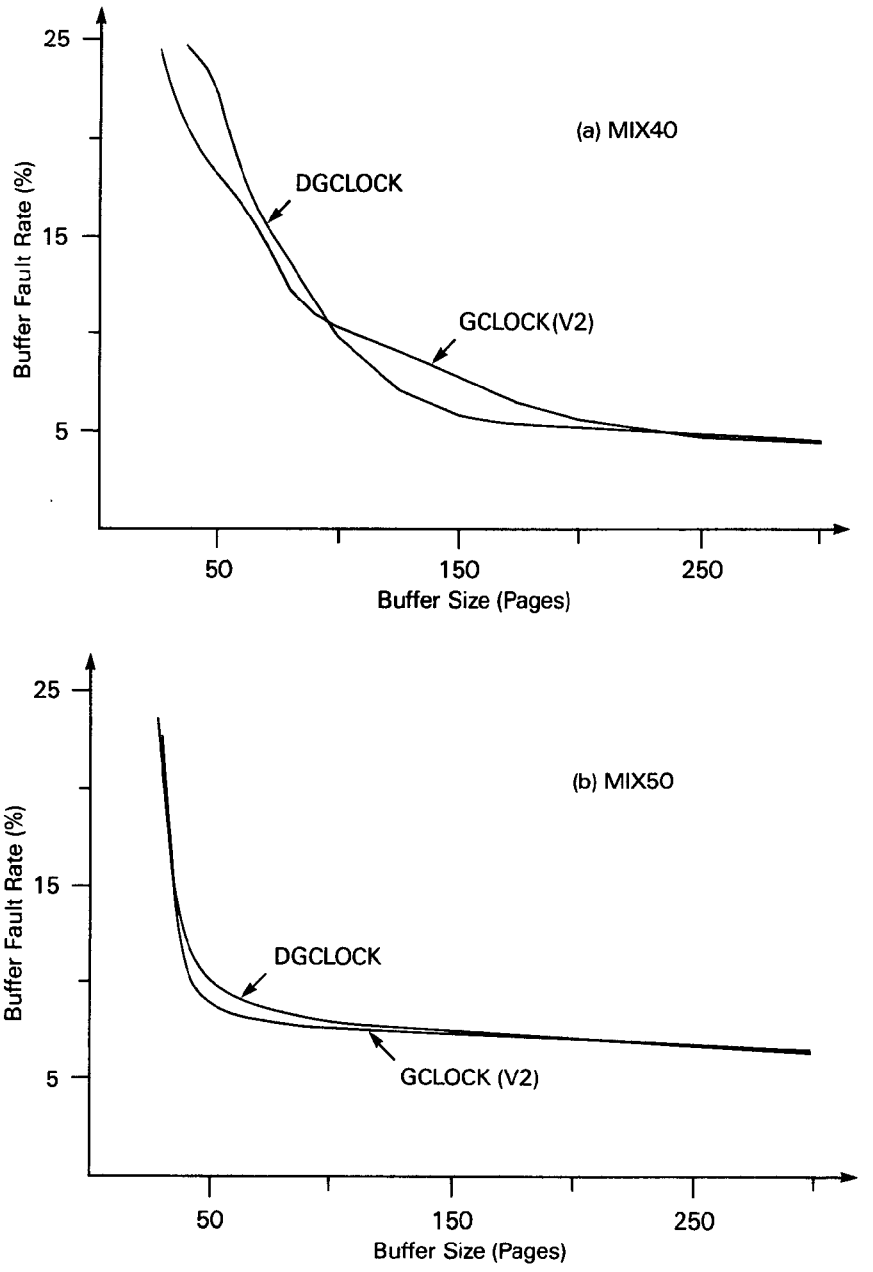


Fig. 15. The buffer fault rates of GCLOCK strategies.

further investigation when applied to a specific DBMS. As shown in Figure 18, we include a local buffer allocation algorithm with dynamic partitions—a working set with LRU replacement—as a comparable candidate. The following reasons, however, seem to weigh against the application of local allocation in DBMS:

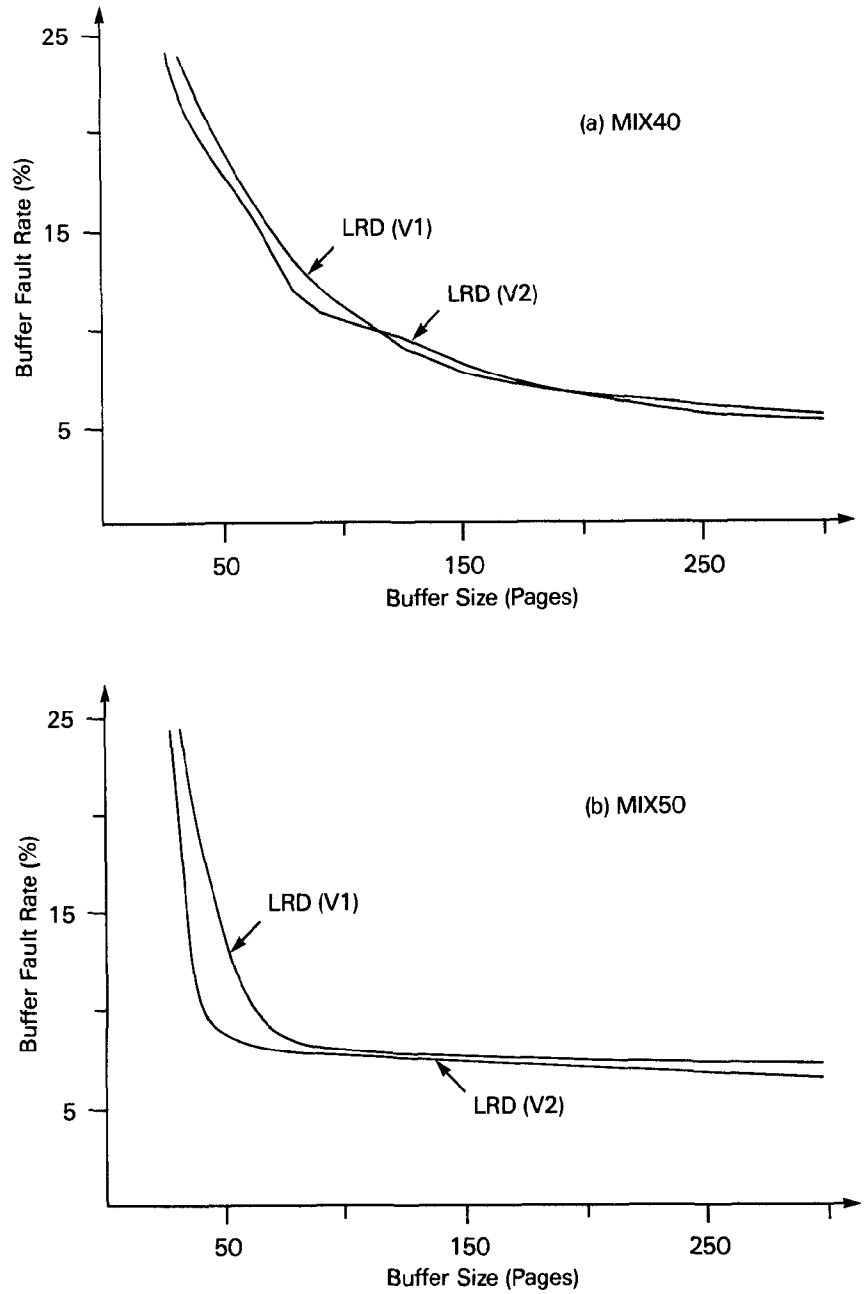


Fig. 16. The buffer fault rates of LRD strategies.

- Such algorithms try to keep a transaction's working set in the buffer, supposing a continuous sequence of operations issued by the transaction. For batch transactions calling the DBMS at intervals of several milliseconds, this assumption is valid. With multistep terminal transactions, groups of a few

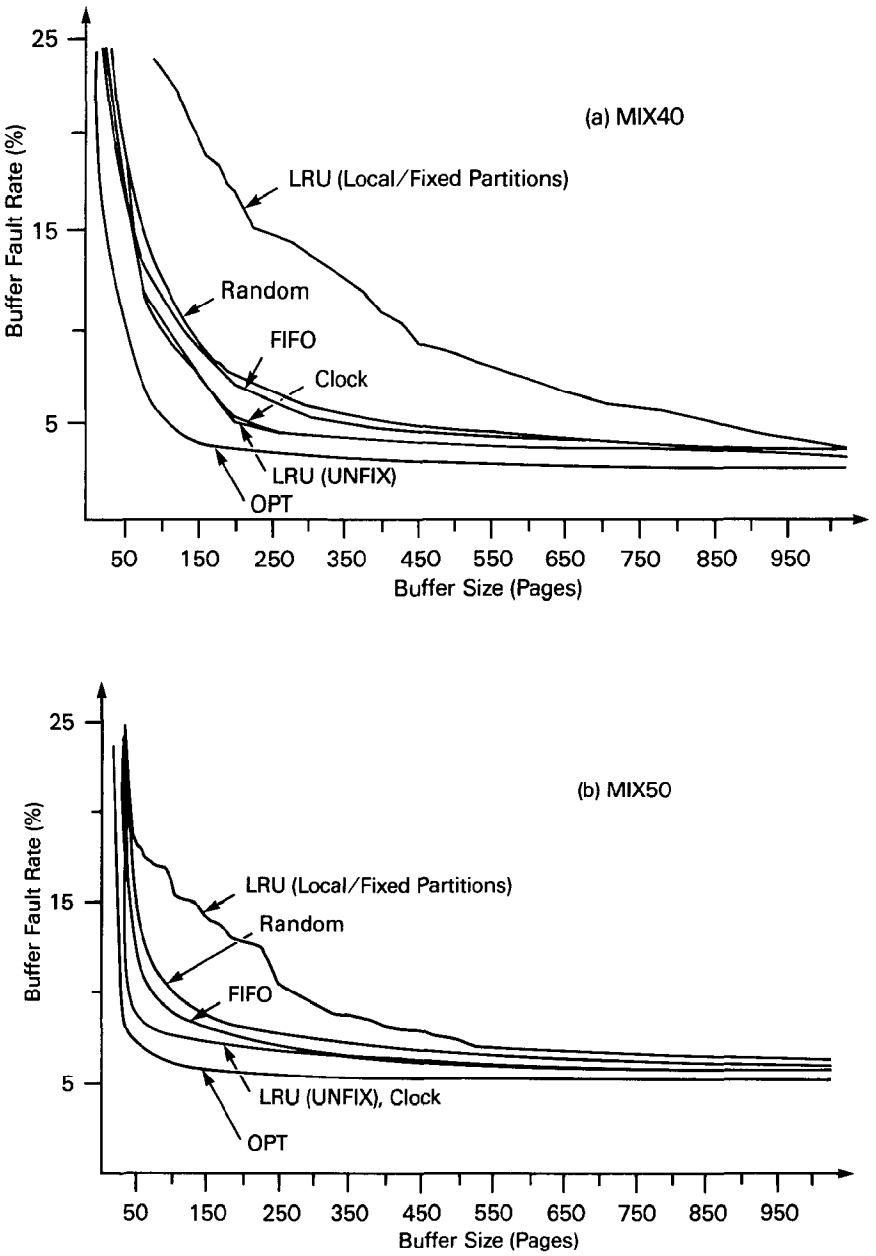


Fig. 17. A comparison of different replacement strategies.

references interrupted by delays of seconds or minutes (the think time of the interactive user) must be expected, making considerable inefficiencies probable.

– Extra overhead has to be paid for handling the allocation of buffer frames for shared pages.

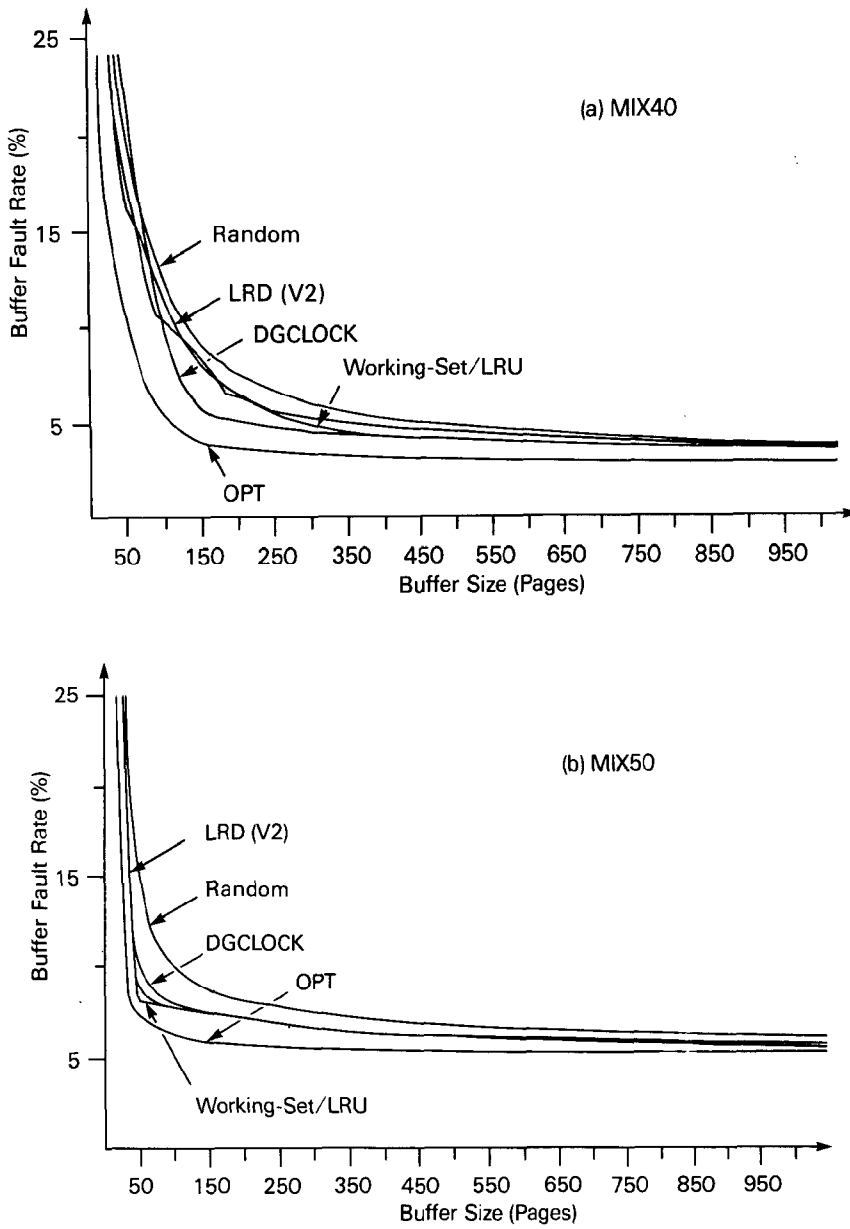


Fig. 18. A comparison of different replacement strategies (continuation).

– The complexity of such algorithms (i.e., the determination of the dynamic partitions) increases with the number of active transactions, provoking substantial overhead in realistic applications.

Comparison of buffer fault rates for OPT and the best applicable algorithms clearly indicates that further optimization with better tailored algorithms may

be worthwhile for restricted buffer sizes (less than 200 pages (400 K bytes)). In the range greater than 200 pages, the best algorithms come fairly close to OPT, so that additional efforts are not justified by the potential gain—at least in our applications, where we had an average degree of parallelism of less than 7. It is, however, conceivable that the range of buffer sizes, in which further optimization efforts pay-off, is enlarged under transaction loads having a higher number of concurrent transactions [13].

6.2 Page-Type-Oriented Buffer Allocation

The disadvantages of local buffer allocation do not apply to page-type-oriented buffer allocation. The allocation of dynamic partitions by means of a WS or PFF algorithm permits flexible and fast adaptation to changes in reference behavior to the various page types. As compared to global allocation, the selective use of a particular replacement algorithm on the set of eligible pages of a specific type is considered to be an extra advantage. Hence, concepts of this kind allow allocation and replacement algorithms to be tailored to the various types of reference behavior.

An analysis of page-type-related reference behavior revealed the following characteristics, considering three different types of pages containing system data (DBTT/FPA), access path data (tables, pointer-arrays, B*-trees, etc.), and records. Here pages containing access path data are summarized as TABLE pages, whereas pages storing data records are called USER pages.

page types		distinct pages referenced (in %) MIX40	MIX50	number of references (in %) MIX40	MIX50
system data	(SYSTEM)	4.9	5.0	10.3	21.8
access paths	(TABLE)	39.6	39.5	22.5	33.9
records	(USER)	55.5	55.5	67.2	44.3

When only two partitions were considered, access path data and records were put together into a single partition (called USER).

Static partition allocation is straightforward; replacement is always done in the partition where the buffer fault occurs. The dynamic partition mechanism works as follows: With N as the total number of buffer frames, at most $N_p = 0.8\ N$ pages were allocated to partitions (working sets) at a time. Different τs were assigned to determine the partition sizes dynamically, according to the following ratio:

$$\begin{aligned} &\text{two partitions:} && \tau_S/\tau_U = 15/85 \\ &\text{three partitions:} && \tau_S/\tau_T/\tau_U = 10/50/40. \end{aligned}$$

N_p is used to determine the various τs directly; for instance, for two partitions,

$$\tau_S = 15/100 * N_p, \qquad \tau_U = 85/100 * N_p,$$

with a suitable lower limit of each τ. For three partitions, a similar assignment was chosen. Hence, the number of buffer pages eligible for replacement was $N_{av} \geq 0.2 * N$.

The influence of partition size with static buffer allocation is evaluated in Figure 19. The behavior of two static partitions for system and user data is shown

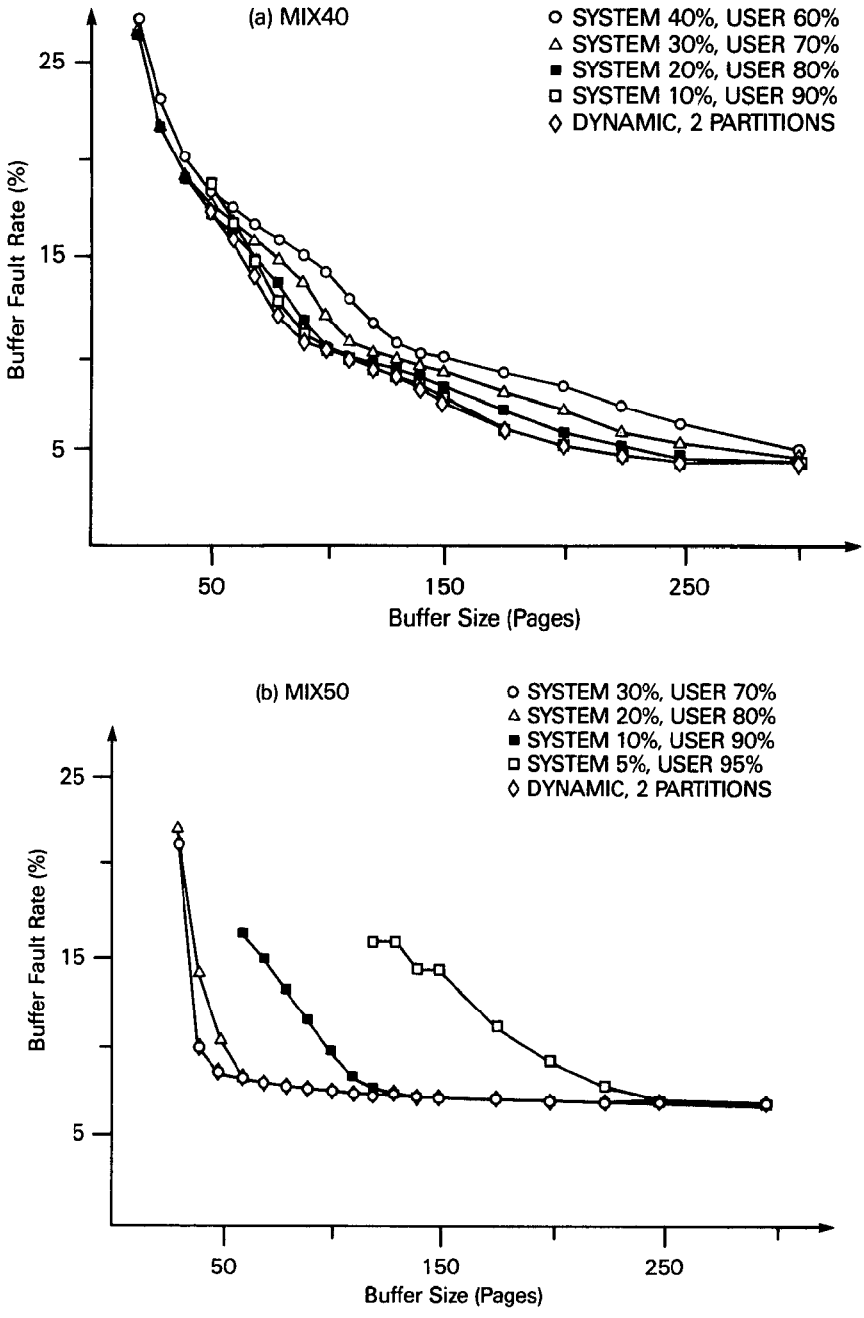


Fig. 19. The buffer fault rate for page-type-oriented allocation schemes with two partitions.

for various ratios of partition sizes and LRU replacement in each partition. The different curves confirm the critical nature of the choice of partition size and the superiority of dynamic allocation. Furthermore, the results indicate the danger of congestion with small buffer partitions, due to pages being fixed.

With static partition schemes, the ratio of actual partition references seems to be a good indicator of appropriate relative partition sizes. The results for the dynamic scheme are approached according to the closeness of the partition size ratio to the reference ratio. Due to different reference ratios, their relative sequence is inverted (when comparing MIX40 to MIX50).

The results presented in Figure 20 are derived with three partitions. The user partition of the previous allocation (Figure 19) was divided into access path pages and other data pages. LRU replacement was applied in each partition. The comparison of MIX40 and MIX50 also reveals the strong dependence of page-type-related reference behavior and partition size with respect to the buffer fault rate obtained. The closer the partition size ratio approximates the page-type reference ratio, the better the results. This causes a different approximation sequence of the various curves for MIX40 and MIX50. As in Figure 19, the results indicate the danger of mistaken partition choices with static schemes, and recommend dynamic partitioning for general use.

The results of the static scheme having a partition size ratio of 10/20/70 for MIX40, which closely resembles the fractions of its actual page-type references, indicate that the parameters of our dynamic scheme (10/50/40) are not appropriately chosen for the respective mix, because the static scheme produces the smallest buffer fault rate in the range of 100 to 160 buffers. This example shows that dynamic schemes of the given type are at least weakly dependent on their initialization parameters. An adaptation of these parameters to the actual reference behavior should lead to an overall superiority of the dynamic scheme.

For our DBMS, three dynamic partitions seem to be an appropriate choice. The buffer fault rate obtained was slightly better than the corresponding rates for two dynamic partitions and for global LRU.

Further refinement is possible by explicitly regarding the locality and type of reference behavior. For example, typical mixes for our DBMS can be characterized as follows:

DBTT/FPA partition:	random access to pages for which there is a high probability of rereference;
index partition:	random access to pages with a nonzero probability of cyclical rereference of some pages (e.g., index roots) or sequential reference cycles during tree traversal (e.g., B*-tree);
data partition:	random or sequential access to pages that will be rereferenced with a lower probability than DBTT/FPA or index pages.

The most important design decision is the determination of working-set parameters. A large τ value for the data partition provokes a large, but useless, partition to be kept as a working set, because locality of reference on data pages is assumed to be very low. Only a small τ value (or none) should be assigned to the data partition, while sufficiently large τs, which can be chosen as a function of buffer size, are needed for the remaining partitions in order to guarantee sizable working sets.

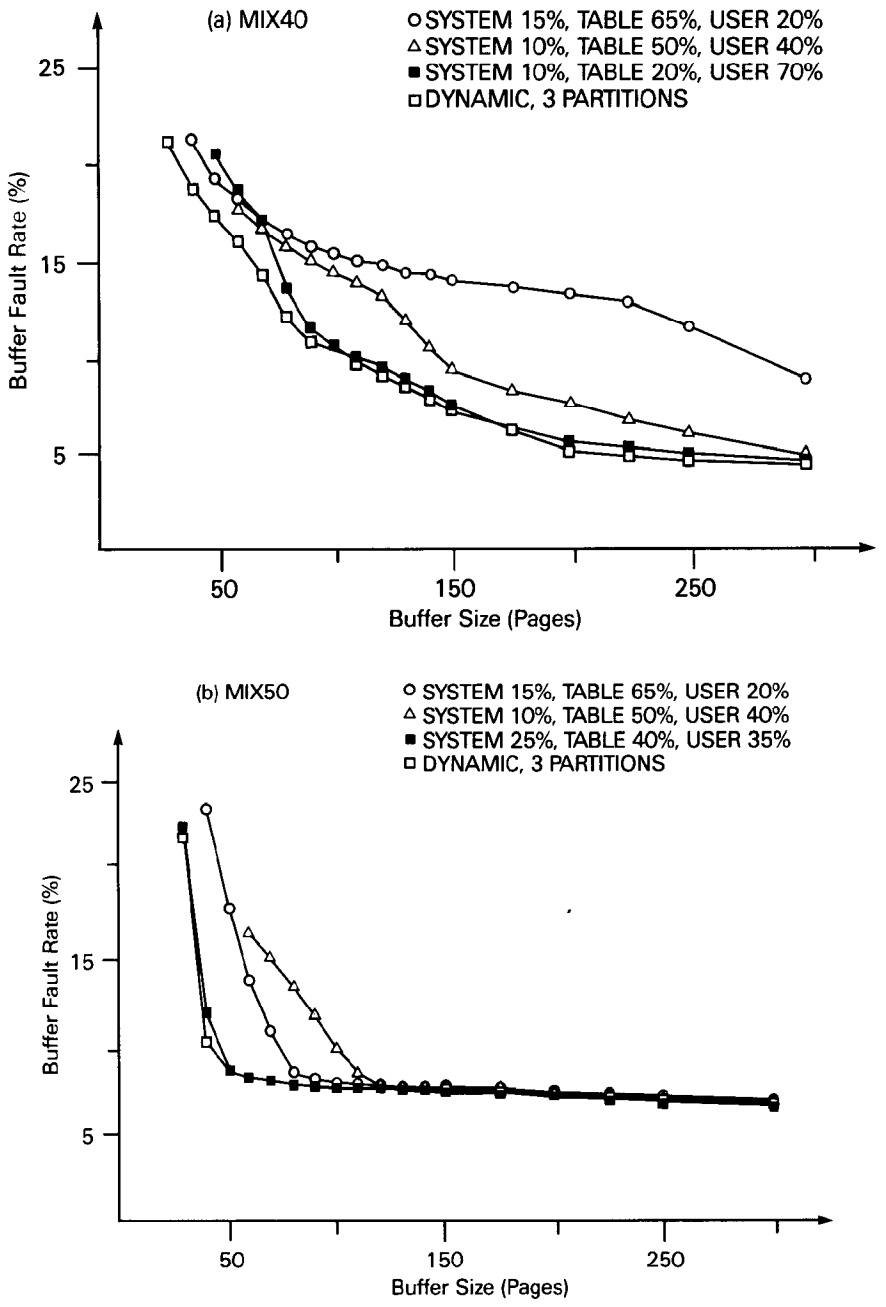


Fig. 20. The buffer fault rate for page-type-oriented allocation schemes with three partitions.

Replacement should be treated as follows: For pages of DBTT/FPA type, LRU or CLOCK algorithms seem to be appropriate. Although these algorithms work well in cases of locality of a given type, they are bad for the other situations. If the reference cycle is larger than the number of buffer frames for index pages,

an LRU-type replacement is the worst possible strategy for the index partition. LRD or GCLOCK algorithms with page weights for the roots and upper-level index pages are better suited to choose the best replacement decision. For data pages, "toss immediately" or FIFO could be used.

From the results derived in our experiments, it is at least debatable whether or not page-type-oriented schemes should be chosen. Even with a dynamic partitioning scheme, the parameters have to be determined to approximate the page-type reference ratio. In addition, the choice of tailored replacement algorithms becomes complex and susceptible to wrong assumptions. Hence, there is no convincing evidence that they are distinctly superior to global schemes.

7. PROBLEMS RELATED TO DBMS BUFFER MANAGEMENT

7.1 DBMS Buffer Management Under a Virtual OS

Embedding a DBMS and an OS environment, in which it is usually treated like a normal application program, can result in aggravating effects on buffer management. If the DBMS runs in a virtual address space, program code as well as the DBMS buffer are paged by OS memory management, unless they are made resident in main memory. While the replacement of buffer pages is done by the DBMS according to logical references, paging of main memory frames is performed by independent OS algorithms based on the addressing behavior within the main memory frames. In such an environment, the following kinds of faults can occur [20]:

Page faults. The required page is contained in the DBMS buffer but is not currently in main memory. It has to be read by the OS from the paging device. A page fault can be provoked by a logical reference, as well as by an addressing operation during the FIX phase.

Buffer faults. The requested page is not found within the buffer. The buffer page selected for replacement, however, is resident, allowing a direct exchange of pages.

Double-page faults. A logical reference to a database page fails and the buffer page to be replaced is not in main memory. In this case the corresponding buffer frame has to be transferred from the paging device, before the replacement of the selected page and the read operation for the new page can be issued.

When the OS itself is running in a virtual environment by use of a hypervisor, it is conceivable that the pathological situation resulting from distributed and uncoordinated resource management is extended to triple-page faults [12].

The frequency of the various faults essentially determines whether or not the buffer manager becomes the bottleneck of the entire DBMS. The analysis of the double-paging problem is a difficult and complex task; for further discussion, we refer readers to [3, 9, 10, 14, 20, 25].

7.2 The Overload Behavior of Buffer Management

Because the DBMS can keep several pages per transaction in FIX status, it is possible that a shortage of buffer frames will occur (a resource deadlock); an additional page is requested, yet no page in the buffer can be replaced if all are

flagged with FIX status. This situation is especially threatening with small buffer sizes. A solution to the problem is to undo the current operation of a transaction, thereby freeing buffer frames occupied by its (the transaction's) associated pages.

With a given buffer size, the number of available frames per transaction decreases as the number of active transactions increases. Unless there is a very high degree of intertransaction locality, the relative frequency of logical references leading to physical I/O grows with the number of parallel transactions. Although the cost of an I/O access remains constant, the total overhead is increased drastically by the increasing relative frequency of page replacement. In this case, so-called *thrashing* [7] can occur—a system state in which almost no useful work is done. To limit the danger of thrashing, a number of measures are proposed [9, 11]:

- optimization of the replacement algorithm,
- reduction of the costs for replacing a page,
- program restructuring to optimize its reference behavior.

These measures serve to reduce the system overhead and to safely increase the number of concurrent transactions that can be processed without provoking the thrashing phenomenon. No guarantee is given that thrashing cannot occur if the system's concurrency is further extended. Hence, thrashing can be prevented, while achieving optimal throughput, only by dynamically limiting the number of transactions. This implies the close cooperation of transaction and buffer management in a DBMS in order to perform effective scheduling of transactions and accurate load control. The same demand is stated for OS: "Memory management and process scheduling must be closely related activities." [6]

8. CONCLUSIONS

We have explained the interface requirements of a DBMS buffer manager and introduced the concept of fixing pages in a buffer to prevent their uncontrolled replacement. The spectrum of possible strategies for searching the buffer was then discussed; hash techniques on buffer information tables with overflow chaining are recommended as the most efficient implementation alternative for the buffer search function.

Initial experiments have shown that locality in DBMS reference behavior is much less significant than locality in the reference behavior of programs under virtual memory operating systems. This motivates the thorough analysis of buffer allocation and page replacement algorithms with respect to DBMS characteristics.

We have classified different buffer allocation algorithms and explained their relationship to page replacement algorithms. Buffer allocation and page replacement are considered to be orthogonal, allowing the combination of each allocation algorithm with an arbitrary replacement algorithm. Since the buffer manager is implemented in software, it is not restricted to the use of hardware flags available in a specific virtual machine architecture. This leads to much more freedom in the design of replacement algorithms. Specifically, we have investigated new ways to combine the age of a page in the buffer, information about recent

references, and information about page contents (page type) into new replacement criteria.

In order to evaluate the performance of various allocation and replacement algorithms in a DBMS environment, we have conducted an empirical study, using two reference strings from CODASYL DBMS applications. A comparison of local and global allocation algorithms shows that local allocation with dynamic partitions leads to buffer fault rates very similar to global LRU replacement; however, we recommend that local allocation algorithms in an on-line transaction processing environment not be used, because user think times would freeze pages in the buffer for long periods of time.

With a global allocation scheme, the adaptation of the buffer contents to the particular reference behavior is left entirely to the replacement algorithm. LRU and CLOCK indicate a satisfactory overall behavior; nevertheless, it could be shown that the LRD and GCLOCK algorithms are also good candidates. Since none of these global schemes requires explicit buffer allocation, they are easy to understand and implement. In addition to global and local algorithms, we have investigated algorithms using page-type information. Different working-set sizes can be assigned to various page types to reflect their specific kind of reference behavior.

As a general conclusion, we were able to show the optimization potential of some of the new algorithms. Since they are parameterized, they can be tailored to a specific DBMS and application environment. The basic trade-off is the conceptual simplicity of the old algorithms versus a potential improvement in performance with the new algorithms.

The following problems are likely to be of interest for future research:

- How can knowledge about the application program be made available for the prediction of future reference behavior? Some means have to be introduced to allow the buffer manager to accept "advice" from the query interpretation or compilation process in order to use the context information of high-level database languages.
- How does the DBMS-specific locality depend on the degree of concurrency of transactions? What is the minimum buffer requirement, as a function of the degree of concurrency?
- How can two-level storage management be generalized for DBMS use? What additional gain can be expected when another level is introduced in the storage hierarchy?

ACKNOWLEDGMENTS

We would like to thank Mary Loomis and Andreas Reuter for their many helpful comments and discussions of this paper. We are grateful to Michael Brunner and Paul Hirsch for their support of the empirical study. The helpful comments of the referees are gratefully acknowledged.

REFERENCES

1. BABAOGLU, O., AND JOY, W. Converting a swap-based system to do paging in an architecture lacking page-referenced bits. In *Proceedings 8th Symposium on Operating Systems Principles. SIGOPS 15*, 5 (Dec. 1981), 78–86.

2. BELADY, L.A. A study of replacement algorithms for virtual storage computers. *IBM Syst. J. 5*, 2 (1966), 78–101.
3. BRICE, R.S., AND SHERMAN, S.W. An extension of the performance of a database manager in a virtual memory system using partially locked virtual buffers. *ACM Trans. Database Syst. 2*, 2 (1977), 196–207.
4. CARR, R.W., AND HENNESSY, J.L. WSCLOCK—a simple and effective algorithm for virtual memory management. In *Proceedings 8th Symposium on Operating Systems Principles. SIGOPS 15*, 5 (Dec. 1981), 87–95.
5. CHU, W.W., AND OPDERBECK, H. Program behavior and the page fault frequency replacement algorithm. *Computer 9*, 11 (1976), 29–38.
6. DENNING, P.J. The working set model for program behavior. *Commun. ACM 11*, 5 (1968), 323–333.
7. DENNING, P.J. Thrashing: Its causes and prevention. In *AFIPS Conference Proceedings, Vol. 33, FJCC*, 1968, 915–922.
8. DENNING, P.J. Working sets past and present. *IEEE Trans. Softw. Eng. SE-6*, 1 (1980), 64–84.
9. EFFELSBERG, W. Buffer management in database systems. Dissertation, Fachbereich Informatik, Technische Hochschule Darmstadt, 1981, (in German).
10. FERNANDEZ, E.B., LANG, T., AND WOOD, C. Effect of replacement algorithms on a paged buffer database system. *IBM J. Res. Dev. 22*, 2 (1978), 185–196.
11. FERRARI, D. The improvement of program behavior. *Computer 9*, 11 (1976), 39–47.
12. HAERDER, T. Embedding a database system in an operating system environment. In *Datenbanktechnologie*, J. Niedereichholz, Ed. Proceedings II/79 of the German Chapter of the ACM, Teubner-Verlag, Stuttgart, 1979, 9–24, (in German).
13. HOWARD, J.H. Virtual memory buffering. IBM Res. Rep., San Jose, Calif., 1980, (in preparation).
14. LANG, T., WOOD, C., AND FERNANDEZ, E.B. Database buffer paging in virtual storage systems. *ACM Trans. Database Syst. 2*, 4 (1977), 339–351.
15. MATTSON, R.L., GECSEI, J., SLUTZ, D.R., AND TRAIGER, I.L. Evaluation techniques for storage hierarchies. *IBM Syst. J. 9*, 2 (1970), 78–117.
16. REITER, A. A study of buffer management policies for data management systems. Tech. Summary Rep. No. 1619, Mathematics Research Center, Univ. of Wisconsin, Madison, Mar. 1976.
17. RODRIGUEZ-ROSELL, J., AND DUPUY, J-P. The design, implementation, and evaluation of a working set dispatcher. *Commun. ACM 16*, 4 (1973), 247–253.
18. RODRIGUEZ-ROSELL, J. Empirical data reference behavior in database systems. *Computer 9*, 11 (1976), 9–13.
19. SACCO, G.M., AND SCHKOLNICK, M. A mechanism for managing the buffer pool in a relational database system using the hot set model. In *Proceedings 8th Conference on Very Large Data Bases*, (Mexico, 1982).
20. SHERMAN, S.W., AND BRICE, R.S. Performance of a database manager in a virtual memory system. *ACM Trans. Database Syst. 1*, 4 (1976), 317–343.
21. SMITH, A.J. Sequentiality and prefetching in database systems. *ACM Trans. Database Syst. 3*, 3 (1978), 223–247.
22. SPIRN, J. Distance string models for program behavior. *Computer 9*, 11 (1976), 14–20.
23. SPIRN, J.R., AND DENNING, P.J. Experiments with program locality. In *AFIPS Conference Proceedings, Vol. 41, FJCC*, 1972, 611–621.
24. STONEBRAKER, M. Operating system support for database management systems. *Commun. ACM 24*, 7 (1981), 412–418.
25. TUEL, W.G. An analysis of buffer paging in virtual storage systems. *IBM J. Res. Dev. 20*, 5 (1976), 518–520.
26. TURNER, R., AND STRECKER, B. Use of the LRU stack depth distribution for simulation of paging behavior. *Commun. ACM 20*, 11 (1977), 795–798.

Received March 1982; revised September 1983; accepted May 1984