Extensible Buffer Management of Indexes†

Chee Yong Chan Beng Chin Ooi Hongjun Lu
Department of Information Systems and Computer Science
National University of Singapore
Lower Kent Ridge Road
Singapore 0511

Abstract

Most extensible database systems support addition of new indexes or new data types. However, the reference patterns exhibited by these new indexes may not be efficiently supported by existing buffer replacement strategies. In this paper, we propose a new mechanism that allows an index method to pass replacement hints to the buffer manager by assigning priority values to the buffer pages to reflect the desired replacement criteria. The proposed approach provides more flexible control over the replacement criteria as different semantics can be encoded using priority values which can also be changed dynamically. Buffer replacement thus becomes extensible since it is possible to customize a strategy to exploit knowledge about the reference pattern of the application. This extensibility also facilitates the design and fine tuning of better replacement strategies. The approach is illustrated with a hierarchical index example. Experimental results show that a customized priority-based replacement strategy outperforms the commonly used LRU strategy.

1. Introduction

Emerging applications such as CAD/CAM and GIS have motivated research into extensible database management systems which aim to achieve two goals [CaH90]. The first important objective is to support addition of new features like new data types, complex objects and access methods to provide greater modeling power and performance. Secondly, extensible systems also seek to facilitate the incorporation of new technologies. Existing prototypes of such systems (e.g. GENESIS [BBG88], EXODUS [CDG90], Starburst [HCL90], DASDBS [SPS90], POSTGRES [StR86], etc) provide extensibility to varying degrees — query language extensions, query processing, storage and access methods, etc. A significant parameter affecting database performance is the I/O cost. Various approaches have been used to minimize the number of disk accesses: clustering of related objects that are frequently retrieved together on disk; using efficient index methods to provide fast access paths to the data; and effective buffering of data pages to reduce page faults. Although most of the prototypes provide support for new data types and index structures, little work has been done in supporting these new index structures, in particular, taking advantage of the knowledge of the reference behaviour of the index structures to optimize the performance of the buffer manager.

New index structures such as spatial indexes (e.g. R-trees [Gut84]) and complex object indexes (e.g. H-trees [LOL92]) are commonly used in applications where data are multi-dimensional or where object-oriented concepts are supported. These indexes exhibit new reference behaviours which may not be efficiently supported by existing buffer replacement strategies. Most of these indexes are hierarchical and the leaf pages cannot be ordered easily. As such, unlike conventional B^+-trees, these leaf nodes are not linked sequentially. Queries that involve the search over a range of values are common. Range queries on such indexes typically involve more index page accesses since each leaf page access is preceded by one or more index page accesses. Thus the resulting page fault rate can be extremely high if the index pages are not buffered efficiently.

† This work was in part supported by NUS Research Grant RP910694.

Proceedings of the 18th VLDB Conference
Vancouver, British Columbia, Canada 1992

For explanation purposes, we consider a variant of the B^+-tree in which the leaf pages are not sequentially linked. For a given range query, the index node that contains the entire query range such that no other nodes in the subtree rooted at that node contain that range can be uniquely identified for the index. We call such a node the *anchor node* of the query. To answer a range query basically involves a depth-first traversal of the subtree rooted at the anchor node.

Let Ref(N) denotes the page reference string of a depth-first traversal of an index subtree rooted at node N and $N_1, N_2, ..., N_s$ be the child nodes of the index node, N. Therefore, Ref(N) = <N, Ref(N_1), N, Ref(N_2),...,N, Ref(N_s), N>. Managing the index pages with the commonly used LRU replacement algorithm is not efficient since it attempts to keep recently accessed pages which may correspond to pages of a traversed subtree that will not be used again.

Consider the example index shown in Figure 1 where node A is the root of the index. Suppose node B is the anchor node; that is, the subtree rooted at node B is the smallest subtree that contains the answer. The logical page reference string for the index traversal is <A_1, B_2, C_3, D_4, E_5, D_6, F_7, D_8, G_9, D_{10}, H_{11}, D_{12}, C_{13}, ...>. The subscript in each page number denotes the reference sequence number.

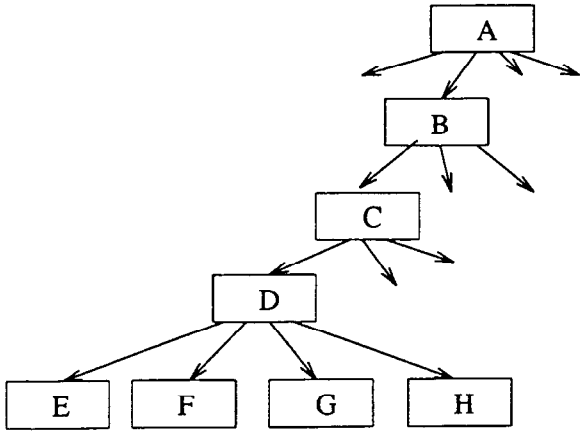

Figure 1: Traversal of a Hierarchical Index

Let us assume that the allocated buffer has 5 pages. Table 1 compares the buffer contents as the index tree is being traversed. Each row in the table shows the buffer contents after the page indicated by the reference number in column one is accessed. The second column shows the buffer contents managed under the LRU replacement strategy while the third column shows the state of the buffer under Belady's *Optimal Replacement Strategy*, which selects the page with the longest expected time until its next reference for replacement [Bel66]. A buffer entry I/J means that a page replacement has occurred with page I replacing page J.

Reference Number	LRU Strategy LRU			--→	MRU	Optimal Strategy				
5	A	B	C	D	E	A	B	C	D	E
6	A	B	C	E	D	A	B	C	D	E
7	B	C	E	D	F/A	A	B	C	D	F/E
8	B	C	E	F	D	A	B	C	D	F
9	C	E	F	D	G/B	A	B	C	D	G/F
10	C	E	F	G	D	A	B	C	D	G
11	E	F	G	D	H/C	A	B	C	D	H/G
12	E	F	G	H	D	A	B	C	D	H
13	F	G	H	D	C/E	A	B	C	D	H

Table 1: Buffer Contents Under LRU and Optimal Strategy

Clearly, the LRU strategy is sub-optimal since it results in more page faults due to poor replacement decisions. Useful pages like B and C, which are needed later, were being replaced by pages in reference numbers 9 and 11 respectively. In fact, by comparing the current buffer contents, we see that only one out of the five buffer pages under the LRU strategy is useful (page C) while under an ideal situation, three pages are useful (pages B and C, and page A which is very likely to be referenced again in subsequent queries).

A hint passing mechanism that enables the buffer manager to exploit any semantics inherent in the index structure or the predictability of its access pattern is desirable. For example, it was observed in [ChK89] that object-oriented applications perform more navigation than ad-hoc queries during runtime, and that most of the access patterns are predictable. The performance of a buffer manager is dependent on both the size of the allocated buffer pool as well as the replacement strategy used. In this paper, we focus on the second issue and propose a priority-based mechanism that enables index methods to pass hints to improve on the buffer replacement decision. This makes the buffer replacement strategy extensible since a replacement strategy can be customized for query reference patterns based on an index structure. A tighter control over buffer page replacement will lead to an improvement in the number of page hits and hence the system performance. The performance analysis for a hierarchical index example indicates that our strategy is more superior than conventional strategies

used in buffering index pages. The proposed approach can be generalized to handle different data pages.

The rest of the paper is organized as follows. In the next section, we review related work on buffer management and extensible database systems. In section 3, we present the design of a mechanism for extensible buffer replacement strategy. Section 4 illustrates an application of the proposed mechanism using a hierarchical index as an example. Both analytical and experimental results that compare the performance of our approach with the LRU strategy are also presented. Finally in section 5, we compare the proposed approach with existing mechanisms, and highlight some future directions.

2. Related Work

In this section, we review earlier work on buffer management strategies for relational DBMS and examine the various approaches adopted in existing buffer managers.

The objective of a buffer replacement strategy is to optimize the selection of a replacement page so as to maximize the number of page hits for an allocated number of buffer pages. The performance of a replacement algorithm depends on its utilization of the buffer pages. For a buffer size of say B pages, an ideal utilization is when the buffer always contains the next B unique referenced pages. This is impractical in general since it demands a priori knowledge of the paging characteristics of a query. Existing replacement algorithms use various criteria to predict the reference behaviour of queries. These buffer management algorithms (e.g. LRU, FIFO) are basically adaptations of memory management policies used in operating systems. A detailed discussion of the various algorithms can be found in [EfH84]. These algorithms are however, not suitable for relational database environment as they do not take advantage of the access pattern of the query plan [Sto81]. In [SaS82, SaS86], Sacco and Schkolnick proposed the hot set model, a model for buffer management using the LRU strategy that takes into account the page reference behaviour of queries to determine the optimal buffer space allocation for a query. Chou and DeWitt [ChD85] extended this model with the DBMIN algorithm that separates the modeling of the reference behaviour from any particular buffer management algorithm. Thus, in addition to determining the optimal buffer size for a query, the DBMIN algorithm also uses knowledge of the access pattern to select an optimal replacement strategy for the query. Empirical results show that these query-oriented strategies outperform the conventional virtual memory page replacement strategies.

More recent work has looked into buffer management in a DBMS with workload consisting of transactions of different priority levels, such as in transaction processing and real-time systems. In [CJL89], two priority buffer management algorithms were proposed, the *Priority-LRU*, based on LRU, and the *Priority-DBMIN*, based on DBMIN. The major difference between the new schemes and their non-priority equivalents is that the proposed strategies allow higher priority transactions to steal buffer pages from lower priority transactions. The priority-based approach was further developed in [JCL90], where a new algorithm, the *Priority-Hints* was proposed. This new algorithm was shown to perform better than Priority-LRU, and to perform as well as Priority-DBMIN, even for workload that consisted of transactions of equal priority.

Although various extensible database systems have been designed and implemented, only a few of them have looked into means of passing hints to guide the buffer replacement decision to reduce page faults and to fully utilize the buffer. In Exodus [CDG90], most of the system components are extended using a programming language, E, which is an extension of C++. One way suggested in [RiC87] to enable index methods to pass hints to the buffer manager is to allow the database implementor to associate buffering hints with operator methods that are defined for abstract data type classes. In Starburst [HCL90], the buffer pool manager uses the CLOCK algorithm to provide hints on the expected future use of a page. However, it is not clear to what extent these various hint passing mechanisms improve the system performance.

In WiSS [CDK85], a flexible data storage system designed for very high performance, the buffer pool manager uses a LRU replacement strategy combined with hints from the system on which pages are important. The hint specified can be low, mid or high. WiSS buffer manager selects the page with the lowest hint and the oldest timestamp for replacement.

From the literature survey, we observe two trends. Firstly, due to the inadequacy of traditional buffer replacement strategies for database systems, there is a development of increasingly more sophisticated query-oriented approaches for buffer management where knowledge about the reference behaviour of queries are exploited to achieve better performance. Hence, to obtain even better buffer utilization, it seems that more knowledge will have to be communicated to the buffer manager. Secondly, the buffer manager in recent systems adopts basically the same approach: a hint mechanism is used to enhance its replacement strategy which is typically the LRU strategy or a variation of it. However, this approach is rather restrictive since the replacement strategy is actually fixed and it is not clear to what degree the hints can influence the replacement decision. Thus, a

more effective hint mechanism is desirable so that more information can be passed to the buffer manager to improve its performance.

3. Design of Hint Passing Mechanism

The design of an effective hint passing mechanism should satisfy two objectives. Firstly, the mechanism should be flexible enough to allow different levels and types of hints to be passed depending on the index methods. For example, an index method with a more predictable reference behaviour should be able to provide more detailed hints that result in a tighter and better control of the buffer replacement. Secondly, the mechanism should be easy to use.

As can be observed from the various existing buffer replacement strategies, the criteria used for replacement differ basically in their interpretation of the priority associated with the buffer pages. For example, the FIFO algorithm assigns higher priority to younger buffer pages than older ones, the LFU algorithm favours more frequently referenced buffer pages, and the LRU algorithm allocates higher priority to more recently referenced pages. This leads to the motivation of using priority as a representation of hints to the buffer manager, allowing an index method to pass replacement hints to the buffer manager by assigning priority values to the buffer pages to reflect the desired replacement criteria. The buffer manger can then select from among the buffer pages, the one with the lowest priority for replacement. Different replacement strategies can be implemented by varying the assignment of priority values. The explicit use of priority to encode the replacement criteria not only enables existing replacement policies to be implemented but also facilitates experimentation and fine-tuning of new strategies to tailor to new index methods and applications. In fact, this priority-based approach is a generalization of all the replacement strategies since the selection of buffer pages for replacement is basically a scheduling problem. With such an approach, different semantics can be dynamically encoded using priority values to provide a more accurate and better control of the replacement decision.

In our priority-based hint mechanism, each buffer page allocated for an index method is associated with a priority value. Every index page fetched into the buffer will be assigned a priority value which will be modified as the buffer pages are accessed to reflect its "replacement potential" relative to the other buffer pages. When a page fault occurs, the buffer manager will then select the page with the lowest priority value for replacement. Since both the initialization and updating of priority values are determined by the replacement strategy, the buffer manager must share the same interpretation of the priority values as the replacement strategy in order to choose the correct replacement page. To facilitate this requirement, we design an abstraction for a replacement strategy that consists of four interface routines and an allocated memory space, referred to as the work space. The work space serves as a global data region that is accessible by both the interface routines and the index method. The contents of the work space are dependent on the information required by the replacement strategy to encode the priority values. The details of the interface routines are shown in Table 2.

Interface Routine	Description
initialize()	Allocates memory for the work space and initializes its contents. This routine is invoked before the use of the index method.
accessPage(P: pageAddress)	Update the priority values of the pages in the buffer when an index page, P, is accessed.
comparePriority(B_1, B_2: priorityValue)	Evaluate two input priority values and return the lower of the two. This is used by the buffer manager to select a replacement page.
changePriority(P: pageAddress; B: priorityValue)	Update the priority value of an index page, P, to B.

Table 2: Interface Routines for Replacement Strategy

The first three interface routines are *buffer interface routines* and can only be invoked by the buffer manager. To enable priority values to be updated between buffer manager calls, another routine, *changePriority()*, is provided for use in the index method. Each replacement strategy is therefore defined by this abstraction of four routines and work space. We outline two buffer manager routines in Figure 2 to illustrate how they interact with the buffer interface routines. The algorithms are simplified to show only the necessary details.

Access to the various instances of replacement strategy abstraction is through the use of a vector list indexed on the replacement strategy. Vectors of functions and values have been widely used in Starburst as a mechanism to permit easy extensibility [HCL90]. The set of interface routines can also be extended easily, if necessary. The use of the interface routines should become clearer in the next section. In addition to coding the interface routines, defining a new replacement strategy also requires registration of information into system catalogs so that the new strategy is made known to the system and at the same time, it also serves as documentation for future uses and extensions.

```
{ This routine allocates a new buffer pool }
openBuffer (bufferSize)
begin
  Invoke initialize() to initialize work space;
end.

{ This routine searches for pageNo in buffer pool }
getPage (pageNo)
begin
  if pageNo is not found in the buffer then
      use comparePriority() to select a
      replacement page from among the buffer pages;
  Let pageAdd be the address of pageNo;
  Invoke acessPage(pageAdd) to update the priority
      of the page;
end.
```

Figure 2: Outline of Interaction between Buffer Manager and Buffer Interface Routines

4. An Application of the Hint Mechanism

This section illustrates an application of the proposed mechanism using a variant of the B^+-tree index example introduced in Section 1. We show that it is possible to design an optimal replacement strategy using the new priority-based approach by exploiting the predictable access pattern of the index method. Both analytical and experimental results are also presented to compare the performance of the proposed approach with the LRU strategy. We chose to analyze the performance of our proposed strategy on the B^+-tree variant because the structure, when its leaf nodes are not linked sequentially, behaves like other unconventional hierarchical indexes that are based on B^+-trees (e.g. R-trees [Gut84]).

4.1. Priority-based Replacement Strategy

The index pages in the buffer can be partitioned into two sets based on their usefulness. A buffer page is considered useful if it is referenced again later and useless otherwise. The usefulness of an index page in this example is known because of the predictability of the reference pattern. Within each set, the pages can be further partitioned using the level number of the page with respect to the anchor node. The priority assignment is as follows. Useful pages are assigned higher priority than useless ones. Among useless buffer pages, priority is given to pages that are closer to the root level since they are more likely to be referenced again in subsequent queries. For useful pages, priority is given to the more recently used pages since they will be re-referenced sooner because of the depth-first traversal reference pattern. This translates to a lower priority for a useful page that is closer to the root level. Ties among useless index pages at the same level are resolved arbitrarily. However, with page sharing, the other priority values associated with the pages need to be considered too. Note that there can be at most one useful page in the buffer of a certain level number. This priority allocation scheme is dynamic since the priority of a page may decrease as the index tree is traversed. A useful page becomes useless after the subtree rooted at that useful page has been traversed.

The replacement strategy designed for the B^+-tree variant using the proposed dynamic priority scheme yields an optimal strategy. Useful pages, being assigned higher priority, will not be chosen for replacement until all the useless buffer pages have become replaced. Then under the designed replacement strategy, the useful page with the lowest level number will be selected for replacement, which corresponds to the page with the longest expected time until its next reference for a depth-first traversal. For example, consider the index tree in Figure 1. Suppose now the allocated buffer has only four pages and the anchor node is page A. A page fault will occur on the 5^{th} page reference. Since all the buffer pages are useful, under the proposed priority allocation, the page with the lowest level number, page A, will be selected for replacement. The choice would have been the same under Belady's Optimal Replacement Policy because among the buffer pages {A, B, C, D}, page A has the longest future reference distance.

The designed replacement strategy is a hybrid replacement strategy since the useful pages are managed differently from the other pages. In fact, the strategy can be treated as a combination of the LRU and MRU replacement policies. Useful pages are managed in a LRU manner as those pages with a lower level number are accessed less recently, are assigned a lower priority. On the other hand, the management of useless pages is in a MRU order since those pages with a lower level number are assigned a higher priority.

4.2. Implementation of Interface Routines

In this section, we demonstrate how the dynamic priority scheme discussed in the previous section can be easily implemented using the replacement strategy procedural interface.

Following the discussion, the priority of an index page is based on two factors — its status (useful or useless) and level number. We associate a priority byte to each buffer page in used to represent its priority value. The status and level number information is encoded as follows. The most significant bit (MSB) is used to encode

```
initialize()
begin
  Allocate memory for variables status and level;
  Initialize status to useless;
  Initialize level to 1;
  Set pointer in appropriate vector to the
      allocated memory;
end.

accessPage(P : pageAddress)
begin
  Locate the buffer page containing P;
  Encode its priority byte using the values
      in the work space;
end.

comparePriority (B1, B2 : priorityByte)
begin
  if MSB(B1) = off and MSB(B2) = off then
      return MAX (B1, B2)
  else
      return MIN (B1, B2)
end.

changePriority (P : pageAddress; B : priorityByte)
begin
  Locate the buffer page containing P;
  Update its priority byte to B;
end.

indexSearch (r1, r2 : dataValue)
begin
  Search the index for anchor node using query range
      [r1, r2];
  Invoke changePriority() to change the priority byte of
      anchor page such that its status becomes useful;
  Set status in the work space to useful;
  Traverse index tree rooted at anchor node using
      depth-first traversal;
      Whenever an index page P backtracks
        invoke changePriority() to change the priority
        byte of P such that its status becomes useless;
      As the index is traversed, update the level number
        in work space;
end.
```

Figure 3: Outline of Buffer Interface & indexSearch Algorithms

the page status. It is set for a useful page and off otherwise. The remaining least significant bits (LSB) are used to encode the level number of the page. The work space therefore consists of just two variables: status and level. The algorithms for the interface routines and the index search are outlined in Figure 3.

Notation	Definition
h	Height of index subtree rooted at the anchor node, excluding the leaf level.
N_i	Node at level i in subtree rooted at the anchor node, $0 \le i \le h$. Therefore, N_0 is the anchor node and N_h refers to a parent-of-leaf node.
f	Order of index, no internal node has more than 2f keys.
k	Average number of keys in an internal node or average fan-out of the index.
B	Number of allocated buffer pages, $B > 0$.
P_{load}	Total number of page faults caused by initial reading of index pages.
P_{LRU}	Total number of page faults caused by re-referencing of replaced pages using the LRU replacement strategy.
P_{NEW}	Total number of page faults caused by re-referencing of replaced pages using the proposed priority-based replacement strategy.

Table 3: Notations Used in Analysis

An advantage of the proposed priority-based approach is that the replacement strategy can be easily extended and fine tuned. For example, to handle updated index pages, one of the seven LSBs can be reassigned as a dirty bit and the *comparePriority()* routine modified to allocate higher priority to dirty pages to defer their more expensive replacement.

4.3. Analytical Results

This section provides an analytical comparison of the performance of the proposed and LRU replacement strategies. The performance metric used is the number of page faults. Table 3 explains the notations used in the

analysis.

We assume in the analysis that each index node is $ln2$ full on average [Yao78], which implies $k = 2f\,ln2$. The total number of page faults = $P_{load} + P_{LRU/NEW}$ where $P_{load} = \sum_{i=0}^{h} k^i$, assuming that every index page in the subtree rooted at the anchor node is accessed.

For the proposed strategy, the optimal buffer utilization occurs when B = h + 1. In this case, the buffer always contains a complete path of useful nodes, {N_0, N_1, ..., N_{h-1}, N_h}, from the anchor node level down to the parent-of-leaf level. Since all the pages to be reused are always available in the buffer, therefore $P_{NEW} = 0$. Clearly, no benefit will be gained if more than (h + 1) buffer pages are allocated. However, when $B < h + 1$, the initial retrieval of nodes N_B, N_{B+1}, ..., and N_h, will replace the useful nodes N_0, N_1, ..., and N_{h-B}. Subsequently, a page fault will occur whenever an index node, N_i, $1 \le i \le h+1-B$, backtracks to its parent node. Therefore, for the proposed strategy,

$$P_{NEW} = \begin{cases} \sum_{i=1}^{h+1-B} k^i & \text{if } B < h+1,\ h > 0 \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

For the LRU strategy, when B = 1, $P_{LRU} = \sum_{i=1}^{h} k^i$, if h > 0, since a page fault will occur whenever an index node backtracks to its parent node; otherwise $P_{LRU} = 0$ if h = 0. When $1 < B < k + 2$, all the useful nodes except N_{h-1}, will be replaced by the retrieval of the k nodes at level h. Hence, this means that a page fault will occur whenever an index node, N_i, $1 \le i \le h-1$, backtracks to its parent node. Therefore, $P_{LRU} = \sum_{i=1}^{h-1} k^i$ when $1 < B < k + 2$. However, when B = k + 2, an additional useful node, N_{h-2}, is also not paged out by the fetching of level h nodes. This is because N_{h-2} is always accessed before the retrieval of its subtree so that the traversal of its next subtree will replace only the pages occupied by the previously traversed subtree, leaving N_{h-2} still in the buffer. In fact, (k + 2) is the minimum number of pages required in the buffer in order that no page fault arises when a node N_{h-1} backtracks to its parent node, N_{h-2}. By extending this argument, we conclude that in general, the minimum number of buffer pages required such that no page fault will occur when an index node, N_{h-i+1}, backtracks to its parent node at level h − i, is $(2 + \sum_{j=1}^{i-1} k^j)$, $i \ge 2$. Note that when sufficient buffer space is available such that no page fault occurs when backtracking from an index node at level i to level (i − 1), it also implies that no page fault will occur when backtracking from a node at levels greater than i. It follows that when $2 + \sum_{j=1}^{i-1} k^j \le B < 2 + \sum_{j=1}^{i} k^j$, $i \ge 2$, $P_{LRU} = \sum_{j=1}^{h-i} k^j$, since a page fault occurs only when an index node N_j, $1 \le j \le h-i$, backtracks to its parent node. The analysis for the LRU strategy is as follows:

$$P_{LRU} = \begin{cases} \sum_{j=1}^{h} k^j & \text{if } B = 1,\ h > 0 \\ \sum_{j=1}^{h-1} k^j & \text{if } 1 < B < k+2,\ h > 1 \\ \sum_{j=1}^{h-i} k^j & \text{if } 2 + \sum_{j=1}^{i-1} k^j \le B < 2 + \sum_{j=1}^{i} k^j,\ i \ge 2,\ h \ge 3 \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

The analysis has identified the various hot points, as defined by the Hot Set Model [SaS86], for both the proposed and LRU strategies.

4.4. Experimental Results

In this section, we present the results of experiments conducted on the B^+-tree variant index. The results are verified with the analytical results. The same notations are used as in the analysis.

For the experiments, we are interested in comparing the performance under different index structures (i.e., width and depth), in particular, for different values of the order of the index, f and the height of the index rooted at the anchor node, h. B^+-tree variant indexes were constructed with 700,000 records randomly generated over the domain of [0..500,000] and with f = 5, 10, 20, 30 ... 100, allowing us to study the effect of f on the performance. A mixture of 100 range queries were then run on each of the index built. These range queries are randomly generated to cover different percentage of the data value range, from 25% to 75%, so as to obtain different values for h. For each query, we measured the number of page faults caused by both initial loading, P_{load}, and re-referencing, $P_{NEW/LRU}$, under the two replacement strategies for different buffer pool size. The average values for the 100 queries are then computed.

We compare the performance of the two strategies using the performance gain and the absolute number of page faults. The performance gain is defined as follows:

$$\text{Performance Gain} = \frac{P_{LRU} - P_{NEW}}{P_{LRU} + P_{load}} 100 \quad (3)$$

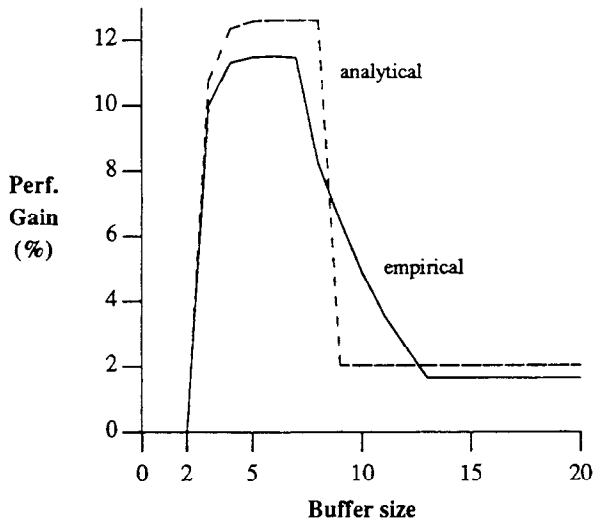

Figure 4(a): Performance Gain as a function of B, f = 5, h = 5

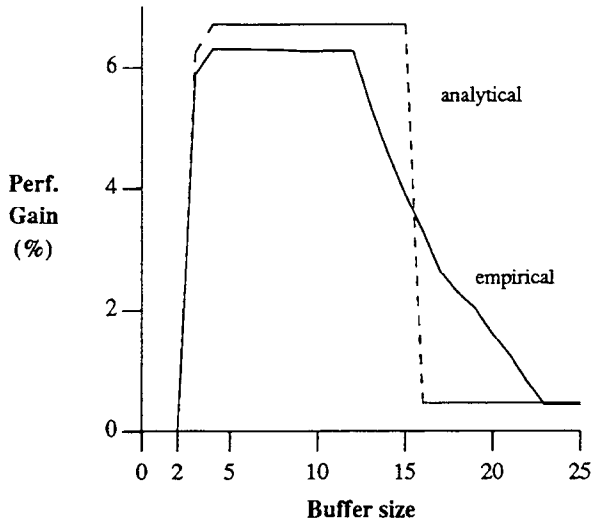

Figure 4(b): Performance Gain as a function of B, f = 10, h = 3

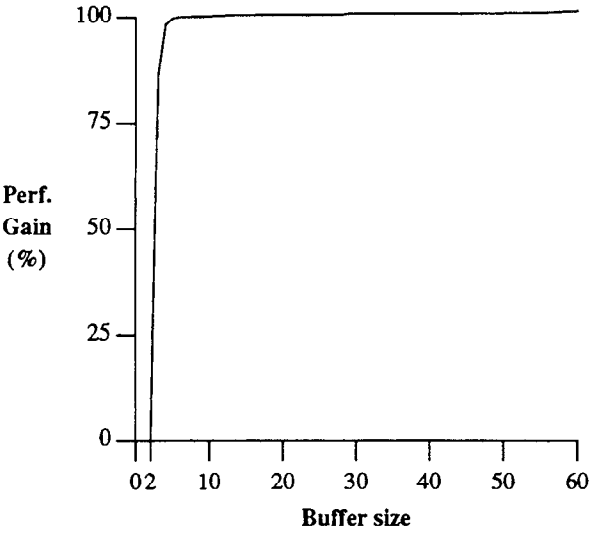

Figure 5(a): Performance Gain as a function of B, f = 5, h = 5

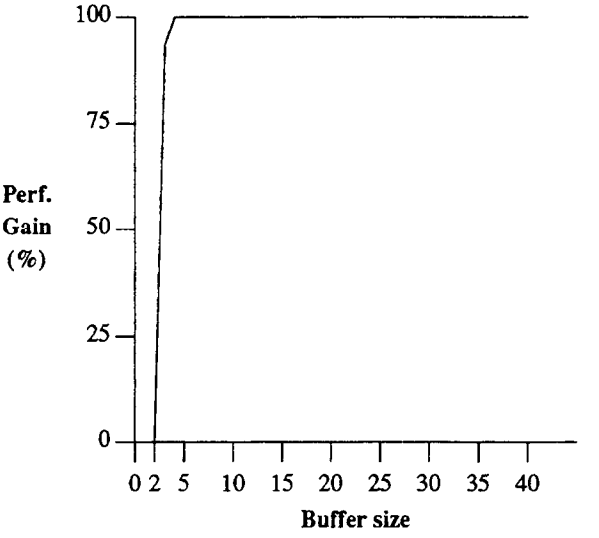

Figure 5(b): Performance Gain as a function of B, f = 10, h = 3

The performance gain for both analytical and experimental results are shown in Figures 4(a) and (b), respectively for indexes with f = 5 and f = 10. The graphs verify the analytical model; the differences are due to the assumptions made in the analysis such as every index node has exactly k child nodes. The experimental results agree with the analysis as the maximum gain occurs when the buffer utilization is optimal (B = h + 1). The initial sharp rise in the graphs is because P_{NEW} drops to zero for the proposed strategy at the hot point, B = h + 1. From these two experiments, we observe that the performance gain is highly affected by P_{load} (P_{load} for Figure 4(a) and 4(b) are respectively 10875 and 2760), which is determined by f and h. With a smaller f, the index has more nodes and hence a higher height, resulting in queries with higher h (height from the anchor node to the leaf minus one) and P_{load}. When P_{load} is higher, the page faults caused by re-referencing under the LRU, P_{LRU}, is more while the value of P_{NEW} stablizes very quickly to zero independent of P_{load}. As such, the relative performance gain is higher for larger P_{load}. We expect that as the index with large f increases its height, the performance gain will increase and become comparable to that of an index with smaller f.

Note that in reading an index, there is a fixed amount of initial loading which cannot be saved, which is

P_{load}. Depending on the index sizes, such initial loading can be far more than the number of page faults caused by re-referencing, $P_{NEW/LRU}$. In order to consider the performance gain independent of P_{load} which is in any case the same under either strategy, we modify the performance gain formula in (3) as follows:

$$\text{Performance Gain} = \begin{cases} 0 & \text{if } P_{LRU}=0 \\ \dfrac{P_{LRU}-P_{NEW}}{P_{LRU}} 100 & \text{otherwise} \end{cases} \quad (4)$$

Figure 5 depicts the corresponding graphs in Figure 4 using the modified performance gain formula. These graphs illustrate that the proposed replacement strategy is in fact optimal with performance gain of 100%, a significant improvement over the LRU strategy. The gain drops to zero when a sufficiently large buffer size is used.

In Figure 6, we illustrate the corresponding graphs in Figure 4 using the absolute number of page faults. It is clear from the graphs that the savings in the number of page faults is significant. Note that when B = h + 1, the total number of page faults under the proposed strategy drops steeply to P_{load} because $P_{NEW} = 0$. For the LRU strategy, more buffer pages are required to achieve the same level of efficiency as in the proposed strategy. The page fault difference is also greater for smaller value of f (larger value of P_{load}) because more pages are re-accessed under the LRU strategy. Experiments were also conducted for indexes with larger values of f. However, the number of nodes and the height for indexes with large f are still quite small for 700,000 records, yielding small P_{load}. We are now experimenting with more records so that the height of indexes is at least 3.

5. Discussions and Future Work

In this paper, we have proposed a priority-based hint mechanism which is not only more dynamic than current hint mechanisms but also makes the buffer replacement strategy extensible to some extent. To our knowledge, none of the existing extensible database systems provides such a level of extensibility. With our approach, an efficient buffer replacement strategy can be tailored for an index method to provide a tighter and better control over the replacement decision. This helps to achieve a better buffer utilization and hence improves the system performance.

We have demonstrated with a hierarchical index example that an optimal hybrid replacement strategy can be designed and implemented with our mechanism. A comparison of the proposed strategy and the LRU, using

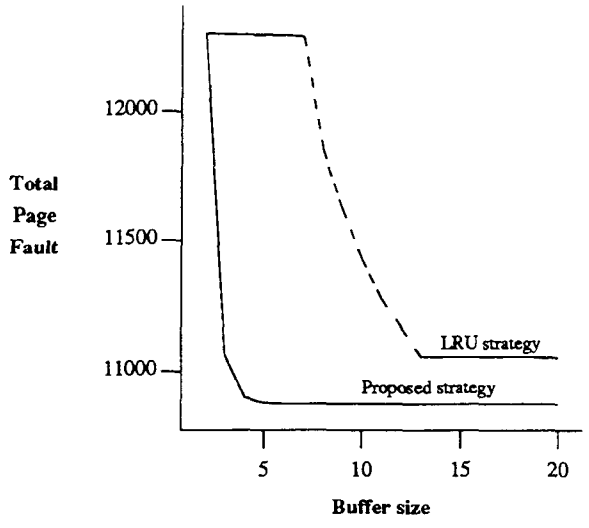

Figure 6(a): Total Page Fault as a function of B, P_{load} = 10875, f = 5, h = 5

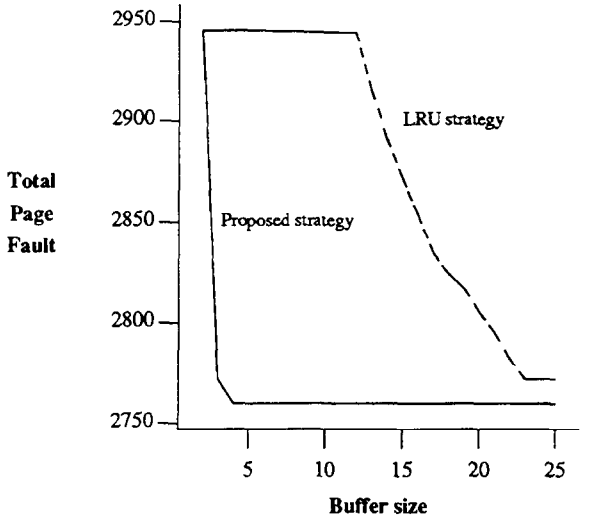

Figure 6(b): Total Page Fault as a function of B, P_{load} = 2760, f = 10, h = 3

both analytical and experimental results, shows that the proposed strategy outperforms the LRU strategy.

The proposed hint mechanism is more flexible as it is not tied to any particular replacement strategy since the replacement strategy is implemented by the hint mechanism. Unlike the buffer manager in WiSS, which uses a fixed replacement strategy with a restrictive three level hint mechanism, our approach enables encoding of different number of levels and types of hints, depending on the knowledge that can be exploited and incorporated into the design. Although the DBMIN algorithm supports a multi-strategy approach, it is based on a classification of

query reference patterns exhibited by common access methods and database operations. The set of strategies supported is therefore fixed and is hardwired into the database system. Addition of new replacement strategies to exploit reference patterns of new applications is not easy as the DBMIN approach is not extensible. Priority based replacement algorithms like Priority-LRU and Priority-DBMIN [CJL89], and Priority-Hints [JCL90] are designed for a multi-priority transactions based system. Our approach can be extended to incorporate the priority of transactions into the priority encoding used in the hint mechanism.

The flexibility and extensibility of the proposed mechanism also enables experimentation of various buffer management schemes. In [EfH84], Effelsberg and Haerder proposed various versions of two main classes of replacement algorithms: one is based on the GCLOCK algorithm and the other is based on the LRD (least reference density) algorithm. Robinson and Devarakonda [RoD90] have also proposed a frequency-based replacement algorithm that effectively combines the principles of locality of reference and reference frequency. All these algorithms involve the use of various parameters for fine tuning. The implementation and fine tuning of such algorithms can be supported by the proposed extensible mechanism.

Some applications that require a non standard replacement strategy include some heuristic-based join algorithms [FoP89, Omi89], a run-time clustering algorithm that exploits intelligent buffer replacement [ChK89], and a buffer manager that implements complex objects using pages with different sizes [Sik88]. We believe that the design of new access methods and algorithms will benefit from an extensible hint mechanism that enables domain related semantics to be exploited for greater efficiency.

Although the focus of this paper has been on management of index pages, the proposed mechanism can be applied to handle the access pattern of other types of pages under various database operations. As part of our future work, we intend to conduct a more extensive experimental study to compare the performance of various database operations using the proposed hint mechanism. There are also other directions that deserve further work. Instead of relying on the programmer to code the replacement strategy by providing a priority scheme, another approach is for the system to derive a priority assignment for the application based on its past reference behaviour and generate the code automatically. Other research areas include the incorporation of a transaction's priority into the hint mechanism for a system with different priority transactions and issues related to the use of the hint mechanism in a dual-buffer architecture employed in some object-oriented DBMSs (e.g. O_2 [Deu90], and DASDBS [SPS90]).

Acknowledgements

We would like to thank the anonymous referees for their useful comments.

References

[BBG88] Batory, D.S., Barnett, J.R., Garza, J.F., Smith, K.P., Tsukuda, K., Twichell, B.C., and Wise, T.E., "GENESIS: An Extensible Database Management System," *IEEE Transactions on Software Engineering*, Vol. 14, No. 11, November 1988, pp. 1711-1729.

[Bel66] Belady, L.A., "A study of replacement algorithms for virtual storage computers," *IBM System Journal*, Vol. 5, No. 2, 1966, pp. 78-101.

[CaH90] Carey, M.J., and Haas, L.M., "Extensible Database Management Systems," *ACM SIGMOD Record*, Vol. 19, No. 4, December 1990, pp. 54-60.

[CDG90] Carey, M.J., DeWitt, D.J., Graefe, G., Haight, D.M., Richardson, J.E., Schuh, D.T., Shekita, E.J., and Vandenberg, S.L., "The EXODUS Extensible DBMS Project: An Overview," in *Readings in Object-Oriented Database Systems*, S. Zdonik and D. Maier (eds.), Morgan-Kaufmann Publishers, 1990, pp. 474-499.

[CDK85] Chou, H-T., DeWitt, D.J., Katz, R.H., and Klug, A.C., "Design and Implementation of the Wisconsin Storage System," *Software Practice and Experience*, Vol. 15, No. 10, October 1985, pp. 943-962.

[ChD85] Chou, H-T., and DeWitt, D., "An Evaluation of Buffer Management Strategies for Relational Database Systems," *Proc. of 11th VLDB Conf.*, Stockholm, Sweden, August 1985, pp. 127-141.

[ChK89] Chang, E.E., and Katz, R.H., "Exploiting Inheritance and Structure Semantics for Effective Clustering and Buffering in an Object-Oriented DBMS," *Proc. of ACM SIGMOD Conf.*, Portland, Oregon, December 1989, pp. 348-357.

[CJL89] Carey, M.J., Jauhari, R. and Livny, M., "Priority in DBMS Resource Scheduling," *Proc. of 15th VLDB Conf.*, Amsterdam, August 1989, pp. 397-410.

[Deu90] Deux, O., et al., "The story of O_2," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, No. 1, March 1990, pp. 91-108.

[EfH84] Effelsberg, W., and Haerder, T., "Principles of Database Buffer Management," *ACM Transactions on Database Systems*, Vol. 9, No. 4, December 1984, pp. 560-595.

[FoP89] Fotouhi, F., and Pramanik, S., "Optimal Secondary Storage Access Sequence for Performing Relational Join," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 1, No. 3, September 1989, pp. 318-328.

[Gut84] Guttman, A., "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proc. of ACM SIGMOD Conf.*, May 1984, pp. 47-57.

[HCL90] Haas, L.M., Chang, W., Lohman, G.M., McPherson, J., Wilms, P.F., Lapis, G., Lindsay, B., Pirahesh, H., Carey, M.J., and Shekita, E., "Starburst Mid-Flight: As the Dust Clears," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, No. 1, March 1990, pp. 143-160.

[JCL90] Jauhari, R., Carey, M.J., and Livny, M., "Priority-Hints: An Algorithm for Priority-based Buffer Management," *Proc. of 16th VLDB Conf.*, Brisbane, Australia, August 1990, pp. 708-721.

[LOL92] Low, C.C., Ooi, B.C., and Lu, H., "H-Trees - A Dynamic Associative Search Index for OODB," to appear in *Proc. of the 1992 ACM SIGMOD Conf.*, San Diego, California, June 1992.

[Omi89] Omiecinski, E.R., "Heuristics for Join Processing Using Nonclustered Indexes," *IEEE Transactions on Software Engineering*, Vol. 15, No. 1, January 1989, pp. 18-25.

[RiC87] Richardson, J.E., and Carey, M.J., "Programming Constructs for Database System Implementation in Exodus," *Proc. of ACM SIGMOD Conf.*, San Francisco, December 1987, pp. 208-219.

[RoD90] Robinson, J.T., and Devarakonda, M.V., "Data Cache Management Using Frequency-Based Replacement," *Proc. of ACM SIGMETRICS Conf.*, Colorado, May 1990, pp. 134-142.

[SaS82] Sacco, G.M., and Schkolnick, M., "A Mechanism for Managing the Buffer Pool in A Relational Database System Using the Hot Set Model," *Proc. of 8th VLDB Conf.*, Mexico City, September 1982, pp. 257-262.

[SaS86] Sacco, G.M., and Schkolnick, M., "Buffer Management in Relational Database Systems," *ACM Transactions on Database Systems*, Vol. 11, No. 4, December 1986, pp. 473-498.

[Sik88] Sikeler, A., "VAR-PAGE-LRU: A Buffer Replacement Algorithm Supporting Different Page Sizes," *Proc. of Int. Conf. on Extending Database Technology*, Venice, Italy, March 1988, Lecture Notes in Computer Science 303, Springer-Verlag, pp. 336-351.

[SPS90] Schek, H.-J., Paul, H.-B., Scholl, M.H., and Weikum, G., "The DASDBS Project: Objectives, Experiences, and Future Prospects," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, No. 1, March 1990, pp. 25-43.

[Sto81] Stonebraker, M., "Operating System Support for Database Management," *Communications of the ACM*, Vol. 24, No. 7, July 1981, pp. 412-418.

[StR86] Stonebraker, M., and Rowe, L., "The Design of POSTGRES," *Proc. of ACM SIGMOD Conf.*, Washington, DC, June 1986, pp. 340-355.

[Yao78] Yao, A., "Random 3-2 Trees," *Acta Informatica*, Vol. 2, No. 9, 1978, pp. 159-170.