UMIACS-TR-93-49 June 1993
CS-TR-3084

Adaptive Database Buffer Allocation Using Query Feedback[*]

ChungMin Melvin Chen Nicholas Roussopoulos

Institute for Advanced Computer Studies and
Department of Computer Science
University of Maryland, College Park
E-mail: min@cs.umd.edu, nick@cs.umd.edu

Abstract

In this paper, we propose the concept of using query execution feedback for improving database buffer management. A query feedback model which adaptively quantifies the page fault characteristics of all query access patterns including sequential, looping and most importantly random, is defined. Based on this model, a load control and a marginal gain ratio buffer allocation scheme are developed. Simulation experiments show that the proposed method is consistently better than the previous methods and in most cases, it significantly outperforms all other methods for random access reference patterns.

1 Introduction

The topic of buffer management in database management systems has been long investigated in the past. The goal of such research is to develop a buffer manager suitable for the database system in order to enhance the system performance. Early works [Rei76, Kap80, EH84] accomplished this goal by adapting conventional allocation and replacement strategies for virtual memory system to database management systems. Recently, another class of algorithms [CD85, SS86, NFS91] based on the prediction of *page access patterns* exhibited by queries were proposed. Taking into account the access patterns helps the latter methods to outperform the earlier ones in terms of system throughput.

[*]This research was sponsored partially by NASA under contracts NAS5-31351 and USRA 550-81, by the National Science Foundation under grant IRI-8719458. This paper also appears in *Proceedings of the 19th International Conference on Very Large Data Bases, 1993.*

buffer management algorithms	allocation/replacement policy			admission policy
	sequential	looping	random	
	$[l_{min}, l_{max}]$, rpl	$[l_{min}, l_{max}]$, rpl	$[l_{min}, l_{max}]$, rpl	
DBMIN [CD85]	$[1,1]$, –	$[t,t]$, MRU	$[1,1]$, RAN,	$\sum l_{min} \leq A$
MG-x-y [NFS91]	$[1,1]$, –	$[x\% * t, t]$, MRU	$[1,y]$, RAN	$\sum l_{min} \leq A$
predictive load control [FNS91]	$[1,1]$, –	$[f(load), t]$, MRU	$[f(load), b_{yao}]$, RAN	$\sum l_{min} \leq A$

Table 1: Buffer Management Algorithms

Although algorithms based on prediction of access patterns capture some of the behavior of database queries, they still have some major deficiencies. First, the strategies proposed were eligible only for very specific patterns; they did not fit well for general pattern (which was classified as *random* accesses). And second, in complicated queries, it is not easy to predict a priori the invoked access patterns. To remedy these, a general mechanism capable of automatically characterizing the access behaviors is desired.

In this paper, we propose a framework of database buffer management featured with *query feedback*. The purpose is to characterize access patterns automatically and refine the buffer allocation from prior query executions, in order to achieve better buffer utilization, and, hence, improve the overall system throughput. A quantitative model of characterizing query access behavior is presented and used for adjusting buffer allocation. We also show the results of our simulation which compares the performance improvement of the proposed work over the existing ones.

The rest of this paper is organized as follows. In Section 2, we review previous work and motivate the approach of using query feedback for database buffer management. In Section 3, we introduce the quantitative model for characterizing query access behavior, and describe the mechanism of feedback and adaptation to fulfill the model. Section 4 describes a buffer management system using query feedback for both load control and buffer allocation. A set of simulation results are given in Section 5, which show the advantages of using feedback in database buffer management. Section 6 summarizes this work.

2 Related Work and Motivation

The problem of buffer management could be formulated in short as follows. In a database environment where concurrent queries arrive and compete for limited buffer resources, the buffer manager's task is to reduce the disk operations and enhance the system's throughput by utilizing a dedicated buffer pool for caching the relation pages. When buffers are available, the buffer manager needs to decide whether to activate a query in the waiting queue, how many buffers to allocate to this query and/or to each relation accessed in this query, and finally, how to replace buffers. Therefore, there are three tasks conducted by the manager: *load control*, *buffer allocation*, and *buffer replacement.*

The first class of database buffer management algorithms [Rei76, SB76, Tue76, Kap80, EH84] used variations of traditional replacement techniques such as LRU (Least-Recently-Used) and Working-Set directly applied to the database buffer pool. However, because of the less page reference *locality* found in database systems than that found in virtual memory systems [RR76, EH84], and due to the lack of embedded load control mechanism, these conventional strategies do not perform satisfactorily and might cause severe system throughput degradation when buffer congestion occurs [CD85].

The above techniques failed to take advantage of specific page reference behavior exhibited by database algorithms such as nested-loop joins, hash joins, and etc. This was corrected in another group of buffer management algorithms [SS82, SS86, CD85, NFS91, FNS91] based on the page reference characteristics exhibited by database queries. In this class of algorithms, load control and buffer allocation are incorporated.

In [SS82, SS86], the authors suggested that in order to run a query efficiently, a minimum number of buffers, called *hot set size*, must be provided during execution. The approach of the hot set based algorithm is improved and refined in following papers [CD85, NFS91, FNS91] based on the classification of reference patters. They are summarized in Table 1, where a *sequential* pattern accesses a sequence of distinct pages, a *looping* pattern accesses a set of pages iteratively, and anything else is called a *random* pattern. Each pattern is associated with an allowable range of allocated buffers $[l_{min}, l_{max}]$, and a suggested replacement strategy *rpl*. Essentially, algorithms in this class only differ in determining the range $[l_{min}, l_{max}]$. For all algorithms, MRU (Most-Recently-Used) replacement is adopted for looping pattern, RAN replacement—which randomly selects a page for replacement—is used for random pattern, and no explicit replacement strategy is needed for sequential pattern since only one buffer page is allocated.

For the algorithm DBMIN proposed in [CD85], each pattern is allocated with a *fix* number of buffers (called *locality set*), this is reflected by $l_{min} = l_{max}$ for all patterns. For looping pattern, the locality set size t is the number of distinct pages referenced in the loop. An incoming query is activated only if the current available buffers, A, is greater than $\sum l_{min}$ — the sum of the minimum buffer requirement of each access pattern induced by the query. While DBMIN was shown to outperform the conventional algorithms, its strict allocation policy might not result in best buffer utilization. For example, a looping pattern query with locality set size of 100 will not be admitted to execution even if there are 90 buffer pages available.

A more flexible allocation algorithm MG-x-y was proposed in [NFS91]. MG-x-y is similar to DBMIN except on the load control decision and hence the number of buffers allocated. As shown in Table 1, MG-x-y allows a looping pattern to be executed if at least $x\% * t$ buffers are available; it allocates up to y buffers to a random pattern, as long as the *expected marginal gain*[1] is still positive and there are still available buffers. It was shown in the same paper that MG-x-y has better throughput improvement than DBMIN due to its flexible allocation. However, keeping x and y as global constants for all queries may not be adequate, since different reference strings, though of same reference patterns, can have completely different faulting behaviors.

In a more recent paper by the same authors [FNS91], a class of *predictive* load control algorithms were proposed. Subject to the current buffer availability, an incoming query is activated only if doing so, it will result in better expected system performance. In other words, $l_{min} = f(load)$ is computed as the minimum buffers needed for a waiting query to be activated in order to enhance the performance of the current load. For random pattern, $l_{max} = b_{yao}$ is the expected number of distinct pages referenced based on Yao's formula [Yao77]. This approach was shown to be more adaptive to different query loads than MG-x-y. However, the computation of b_{yao} and the expected system performance is based on the assumption of *uniform page access*[2], which in general is not true[3].

Obviously, the main weakness of the algorithms mentioned above lies in their inability of characterizing different *random* reference strings. In these schemes, all reference strings, other than sequential and looping, are categorized as *random*, and are treated equally based on the assumption of uniform page accesses. As a consequence, when a query with random reference strings on its accessed relations is admitted, these algorithms will try to allocate as many buffers as possible to each relation since the *expected* marginal gain is usually positive for a wide range of buffer sizes. This neglects the benefit of allocating more buffers to the relations which reduce the most page faults and less to the others.

Another problem is that for complex queries, the prediction of reference patterns may not be accurate due to their non-trivial access methods. For example, in a multi-relation join where

[1]The expected marginal gain is the expected number of page faults reduced per extra buffer allocated.

[2]The assumption of *uniform page access* assumes that a sequence of page references to a relation are distributed uniformly among all pages of the accessed relation.

[3]b_{yao}, in general, is much higher than the real number of page referenced.

non-clustered indices or hash tables are used, according to the classification scheme, the reference strings on these relations will be simply classified as random, while in fact the real page navigation may turn out to be of certain locality instead of uniformity. Another class of complex queries are those found in deductive databases or object-oriented databases. They have totally different access paths from standard relational algebra paths and unpredictable page reference behaviors. In deductive database systems, recursive queries usually generate reference strings which are not sequential or looping. In object-oriented databases, dereferencing of pointers due to its hierarchical structure also generate more "random" page visits.

To cope with these problems, we propose a feedback mechanism to capture query page reference behavior by collecting information during query executions. In this feedback model, all reference strings are associated with a few characteristic values. Buffer management strategies (including load control and buffer allocation), then are adjusted according to these feedback values. The exact size of buffers allocated to each reference string is determined by the current buffer availability and the feedback values that characterize the string. A simple load control mechanism is also adopted in the algorithm proposed here. However, the algorithm is basically an allocation-oriented approach as opposed to a load-control-oriented one. Although adaptive replacement strategies based on (limited) reference history have recently been explored for database systems [OOW93, Che92], in this paper we assume LRU unless a looping pattern is detected in which case we use MRU. Simulation results have shown that the feedback is advantageous. Furthermore, this approach is attractive and practical for the following reasons:

- Recurring and/or mutually-related queries (such as compiled queries, user-defined views, query embedded applications) are common, and, therefore, the use of the feedback information can significantly improve their performance.
- As we mention above, applications in non-relational database models make reference pattern prediction inadequate since most of the reference strings will be classified as random, and therefore, feedback is a proper way to collect knowledge about page reference behaviors.
- Most database systems have a software-based buffer manager, which can be extended to include the feedback mechanism with minimal overhead.

3 The Feedback Model

In this section, we propose a feedback model which is capable of characterizing the *faulting* behavior of any reference string using query execution feedback. The model is *general* in the sense that the behavior of any reference string will be automatically quantified with a faulting characteristic record once it is executed.

3.1 The Faulting Characteristic Model

Definition 1 A *reference string* $\mathcal{R} = \{r_1, r_2, \ldots\}$ is a finite sequence of page references, where each reference r_i is denoted by the corresponding page number. We use $|\mathcal{R}|$ to denote the *normalized length* of $\mathcal{R}$ where consecutive references to the same page are counted as one reference; and let $C(\mathcal{R})$ be the *number of distinct pages* referenced in $\mathcal{R}$. $\square$

As an example, suppose $\mathcal{R} = \{3, 2, 2, 1, 8, 1\}$, then $|\mathcal{R}| = 5$ because page 2 is referenced twice in a row and should be counted as only one reference. In this case, $C(\mathcal{R}) = 4$.

Definition 2 Given a reference string $\mathcal{R}$, and an employed buffer management algorithm B, the *faulting function* $f_{\mathcal{R},B}(b)$ is the *number of page faults* resulted as a function of allocated buffer size b. We will simply denote it as $f(b)$ when $\mathcal{R}$ and B are understood from the context. $\square$

It follows from the above definitions that:

$$C(\mathcal{R}) = f(C(\mathcal{R})) \leq f_{\mathcal{R},\mathrm{B}}(b) \leq f(1) = |\mathcal{R}|, \ \ for \ b \geq 1.$$

This expresses the fact that when a reference string is traced, no matter what buffer management strategy is adopted, at least $C(\mathcal{R})$ disk reads must be performed to access all the distinct pages; and at most $|\mathcal{R}|$ page faults can occur when only one buffer page is allocated. In general, there is no precise mathematical formula to express the page faults as a function of buffer sizes, even if the values of $C(\mathcal{R})$ and $|\mathcal{R}|$ are known. Though for certain class of replacement algorithms [M$^+$70], the faulting function on any reference string could be obtained by tracing the string only once, the overhead of computing and maintaining the number of buffer faults at *every* buffer size is large. For this reason, we introduce a model to characterize the faulting function.

Definition 3 (The Faulting Characteristic Model (FCM)) The *faulting characteristics* of a faulting function $f_{\mathcal{R},\mathrm{B}}(b)$ at buffer size b_0 is a triple $\rho_{b_0} = (g, c, s)$, where

$$\begin{aligned} g &= (f_{\mathcal{R},\mathrm{B}}(1) - f_{\mathcal{R},\mathrm{B}}(b_0))/(b_0 - 1), \\ c &= 1 + (b_0 - 1)\frac{f_{\mathcal{R},\mathrm{B}}(1) - C(\mathcal{R})}{f_{\mathcal{R},\mathrm{B}}(1) - f_{\mathcal{R},\mathrm{B}}(b_0)}, \end{aligned}$$

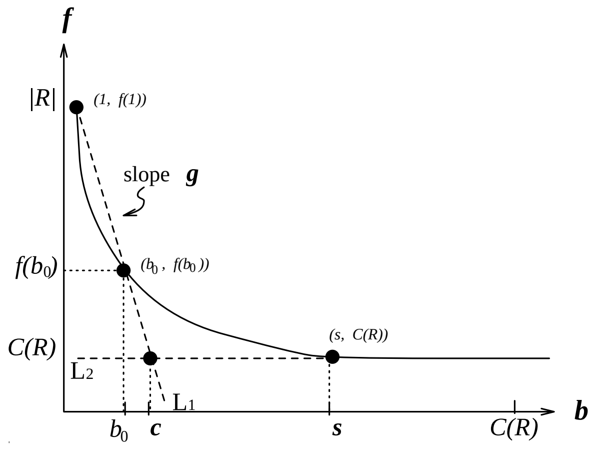


Figure 1: A Typical Faulting Function

$$s = \textit{the minimum } b \textit{ such that } f_{\mathcal{R},\mathtt{B}}(b') = f_{\mathcal{R},\mathtt{B}}(b)$$
$$\textit{for all } b' > b.$$

We call g the *average marginal gain*, c the *critical size*, and s the *saturated size.* □

Intuitively, ρ_{b_0} characterizes the general behavior of $f_{\mathcal{R},\mathtt{B}}(b)$ in the range $1 \leq b \leq b_0$. The idea is depicted in Figure 1, where a typical faulting function is plotted. Suppose b_0 buffers are allocated to the reference string, and as a result $f(b_0)$ faults occur during the execution. Then average marginal gain g is the slope of line L_1 which connects points $(1, f(1))$ and $(b_0, f(b_0))$, and it represents the average page fault reduction per extra buffer allocated in the range of 1 to b_0 buffers. The saturated size s is the smallest buffer size beyond which the slope becomes horizontal. Since s depends only on $\mathcal{R}$ and B, but not b_0, we use $s_{\mathcal{R},\mathtt{B}}$ for clarity when needed. It is easy to see that $f(s) = f(C(\mathcal{R})) = C(\mathcal{R})$, and, in general, s is smaller than $C(\mathcal{R})$[4]. Critical size c is the x-axis value of the intersection point of line L_1 and L_2, where L_2 is the horizontal line $f = C(\mathcal{R})$. In other words, critical size is the expected buffer size around which the rate of reduction slows considerably.

We use FCM to quantify both the page access and buffer fault behaviors of queries. For each relation accessed in a query, we associate it with a faulting characteristic record. Also, for each query, a faulting characteristic record is computed using the combined reference string. Based on these values, we can tune the buffer allocation in a more intelligent way. Before we go into the detail of how this is done, we first explain how the faulting characteristics are computed during

[4]In our experiments, if LRU is used and $\mathcal{R}$ is a random pattern, then $s_{\mathcal{R},\mathrm{LRU}}$ ranges from $C(\mathcal{R})/3$ to $C(\mathcal{R})$

execution, and discuss how these values are used and refined.

3.2 Feedback Mechanism

Suppose reference string $\mathcal{R}$ is currently running under buffer management algorithm B with dedicated buffer capacity b_0. According to the definition of FCM, there are four basic numbers to be computed: $f_{\mathcal{R},\mathtt{B}}(b_0)$, $|\mathcal{R}|$, $C(\mathcal{R})$, and $s_{\mathcal{R},\mathtt{B}}$.

$f_{\mathcal{R},\mathtt{B}}(b_0)$, number of page faults:

A counter for number of page faults is maintained. This counter will simply increase each time when a page fault occurs.

$|\mathcal{R}|$, normalized length of the reference string:

To detect consecutive references to the same page, two variables are maintained. The first variable records the previous logical page reference and will be refreshed at each page request; the other is a counter for $|\mathcal{R}|$ which will be increased by one each time when a page different from the previous reference (which is recorded in the first variable) is requested.

$C(\mathcal{R})$, number of distinct pages referenced in $\mathcal{R}$:

Either a static vector or a dynamic hashing data structure has to be maintained so that when a page fault occurs, the structure will be searched to see if the faulted page is a new occurrence. A counter for $C(\mathcal{R})$ will increase by one whenever a new occurrence encountered.

$s_{\mathcal{R},\mathtt{B}}$, the saturated buffer size:

Saturated size depends both on the reference string and the underlying buffering strategy. Its exact value in general is not easy to calculate, simply tracing the reference string repeatedly for all buffer sizes to find the saturated size is impractical. However, for LRU replacement, its saturated sizes can be found efficiently.

According to [M+70], LRU is a special member of a class of replacement algorithms called *stack algorithms.* Suppose we accommodate a LRU-stack of size $C(\mathcal{R})$ and trace $\mathcal{R}$ under LRU replacement, then according to that paper,

$$f_{\mathcal{R},\mathtt{LRU}}(b) = C(\mathcal{R}) + \sum_{i=b+1}^{C(\mathcal{R})} hit(i),$$

where $hit(i)$ is the frequency of page hits on the i'th position relative to the top of the LRU-stack. According to our definition, the saturated size is the minimum s such that $f_{\mathcal{R},\mathtt{LRU}}(s) = C(\mathcal{R})$,

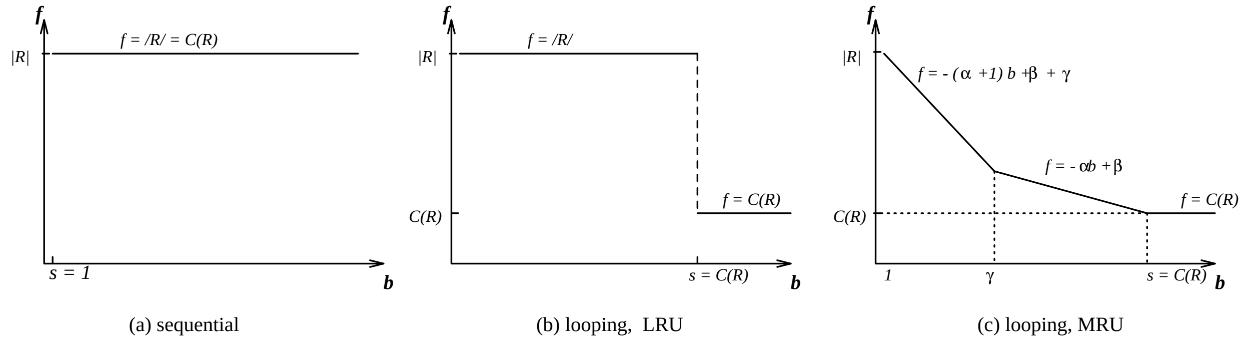


Figure 2: Faulting Functions of Sequential and Looping References

therefore, as a consequence of the above equation, $hit(i) = 0$ for all $(s+1) \leq i \leq C(\mathcal{R})$. We then have:

$$s_{\mathcal{R},\text{LRU}} = \max\{i | hit(i) \neq 0, 1 \leq i \leq C(\mathcal{R})\}.$$

If we associate the trace of $\mathcal{R}$ with a LRU-stack of size $C(\mathcal{R})$, $s_{\mathcal{R},\text{LRU}}$ can be obtained from the above expression. Actually, as long as the stack size allocated is greater than or equal to the saturated size (instead of $C(\mathcal{R})$), the saturated size can still be computed accordingly. In the case of insufficient stack size or the strategy is not LRU, we can either simply set $s_{\mathcal{R},\text{B}} = C(\mathcal{R})$ or estimate/compute its value based on some other techniques (refer to [Che92] for more detail).

3.3 Use and Adaptation of Faulting Characteristics

In this subsection, we discuss how the faulting characteristics can be used to differentiate the access patterns, and show how they can be adaptively adjusted, over recurring query executions, to reflect more informed faulting behaviors.

Figure 2.a shows a sequential pattern, where the page faults do not reduce as buffers increase. Therefore, no matter how many buffers are allocated to it, according to our feedback mechanism, the resulting characteristic record is always $(g, c, s) = (0, 1, 1)$. We can use this to detect a sequential pattern. As for looping references, its faulting function depends on the replacement strategy. Figure 2.b shows the case of LRU replacement under which each page reference results in a page fault, unless $C(\mathcal{R})$ buffers are provided. Therefore, if the feedback characteristic records of a reference string $\mathcal{R}$ turn out to be $(g, c, s) = (0, 1, C(\mathcal{R}))$ for several different buffer sizes less than $C(\mathcal{R})$, then the reference pattern is more likely looping and, thus, MRU should be tried during its next execution. Actually, the detection of a looping pattern can be quickly confirmed by checking if $hit(C(\mathcal{R})) = |\mathcal{R}| - C(\mathcal{R})$ during the first trace, i.e., every hit goes to the bottom of the LRU-stack. On the contrary, Figure 2.c shows the faulting function under MRU, where coefficients α, β, γ are uniquely determined from the values of $|\mathcal{R}|$ and $C(\mathcal{R})$[5]. Therefore, once a MRU is adopted for a looping reference string, its faulting function could be expressed as an equation with coefficients computed from the feedback values of $|\mathcal{R}|$ and $C(\mathcal{R})$.

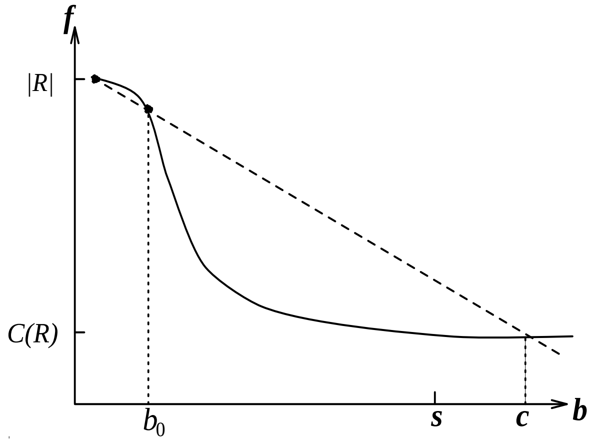


Figure 3: An Example of Non-Representative Feedback

For all reference strings, the obtained faulting characteristics depend on the initial allocated buffer sizes b_0. If the initial b_0 is too close to 1, the resulting characteristic may not represent the average behavior over a wider range of buffer sizes. This is illustrated in Figure 3 which results in $c > s$. Though this is a rare case, we can simply set the critical size to be equal to the saturated size, and adjust it in later executions. We propose here a simple adaptive procedure for adjusting the characteristic record to obtain a more informed record.

For the marginal gain and the critical size, their values are adjusted gradually to best reflect the faulting behavior. Formally, let ρ_b be the current characteristic record, where b is the buffer size from which ρ_b is computed. Now suppose b' is the buffer size allocated for the next execution, then $\rho_{b'}$ replaces ρ_b if either

a) $b' > b$ and $r_{b'} > \theta$, or

b) $b' < b$ and $r_{b'} < \theta$,

where $r_{b'} = \frac{f(b') - C(\mathcal{R})}{s - b'}$ is the slope between the two points $(b', f(b'))$ and $(s, C(\mathcal{R}))$. We call $r_{b'}$ the *residual gain* at b'. θ is a constant threshold set for all reference strings. Intuitively, $\rho_{b'}$ is a

[5]In [NFS91], the *expected* marginal gain for looping patterns is derived, whose value is a constant depending on the reference string. The formula we give here computes the *exact* values instead of expected value, the detail of the derivations can be found in [Che92].

more informed feedback than ρ_b if b' is closer to the size beyond which the residual gain becomes smaller than the pre-defined threshold θ. The first conditions in both cases guarantee that under a stingiest buffer allocation, a less informed record will not replace an earlier more informed one. This assures us that the adaptation, starting from either a under-allocation (case a) or an over-allocation (case b), eventually converges to a well informed characteristic record whose residual gain is close to θ.

4 Buffer Management based on FCM

In this section, we describe the load control and buffer allocation mechanisms based on FCM.

Load Control

Since the purpose of this paper is to demonstrate the strength of adaptive buffer allocation based on feedback information, a simple load control mechanism is used. The load control depends on the critical size c of query q. Note that we compute c using the combined reference string of q. A query q is activated if

$$A \geq 0.5 * c,$$

where A is the available buffers.

Buffer Allocation

In an environment where several concurrent reference strings are traced, the buffer allocation algorithm is used to allocate the buffers among the reference strings. For example, buffers can be allocated among concurrent queries, and within each query, the buffers assigned to this query can be again divided among all simultaneously referenced relations. DBMIN and MG-x-y are examples of such allocation algorithms. However, their allocation strategies for random reference strings are based on the assumption of uniform page accesses. As an alternative, we introduce an allocation algorithm based on the FCM. Since FCM quantifies the reference behavior from feedback, it provides a more accurate information than that based on uniformity.

Allocation based on the Marginal Gain Ratio (MGR) Given n concurrent reference strings, we allocate buffers proportional to their average marginal gains subject to the following constraints:

a) never allocate more than the saturated size (avoid waste), and

b) when the demand for buffers is high, never exceed the critical size of each string. □

We demonstrate the MGR allocation policy, which has been implemented in ADMS[6], with a complete example and show the adaptation process of the feedback mechanism FCM. A 3-way join query which accesses three base relations simultaneously is taken as an example:

```
select * from 10k1, 10k2, 10k3
where10k1.un1 = 10k2.un1 and 10k2.un2 < '500'
     and 10k1.5000 = 10k3.5000
```

Each relation contains 10,000 tuples spanning over 2,500 pages, while the query results in 1,000 tuples. The query is chosen so that none of the reference strings on the relations is of sequential or looping pattern.

Figure 4 tabulates the allocations and feedback values for a sequence of query executions by MGR. Each table denotes an execution, where A is the available buffers and column b corresponds to the buffers allocated to each relation. Column $f(b)$ is the resulting page faults. Column r denotes the residual gain computed based on b and $f(b)$ after the execution. Columns g, c and s are the *best so far informed* faulting characteristic record, they are adjusted properly after each execution. We also keep b^*, the buffer size of the best so far informed characteristic record, i.e. $\rho_{b^*} = (g, c, s)$, in the table. b^* is used in the future execution to determine if new feedback should replace the current one. Adaptation of characteristic record is marked with an asterisk. The threshold θ for the residual gain is set to 1.0 in this experiment.

Just before the first execution, since no feedback information is available, MGR simply allocates the available $A = 50$ buffers evenly among all relations. After the execution, $f(b), |\mathcal{R}|, C(\mathcal{R})$ and s are obtained, and based on these, r, g, c are computed. Since it is the first execution, all feedback will be kept. Note that $|\mathcal{R}|, C(\mathcal{R})$ and s are kept unchanged thereafter.

At Execution 2, 100 buffers are available, MGR allocates buffers proportional to the marginal gains obtained from previous execution, therefore, 10k1 is allocated $100*2.75/(2.75+7.06+0) = 28$ buffers, 10k2 is allocated $100 * 7.06/(2.75 + 7.06 + 0) = 71$ buffers, and 10k3 is allocated 1 buffer since its marginal gain is 0. As a result of this execution, the residual gains for 10k1 and 10k2 are computed to be 1.35 and 6.11, both of which are still greater than the threshold value 1.0, and since they correspond to an execution with more buffers than the previous, their characteristic records are replaced by the new feedback. For example, for 10k1, the characteristic record gets

[6]ADMS, the Advanced Database Management System, is a database management system developed at the Department of Computer Science, University of Maryland, College Park, [RES93].

Execution 1, $A = 50$									
	b	$f(b)$	r	b^*	g	c	s	$\vert\mathcal{R}\vert$	$C(\mathcal{R})$
10k1	17	951	1.39	17	2.75	196	372	995	458
10k2	17	879	6.98	17	7.06	123	125	992	125
10k3	16	897	0.06	16	0.00	1	719	897	853
total		2727							

Execution 2, $A = 100$							
	b	$f(b)$	r	b^*	g	c	s
10k1	28	924	1.35	28*	2.62*	206*	372
10k2	71	455	6.11	71*	7.67*	114*	125
10k3	1	897	0.06	16	0.00	1	719
total		2276					

Execution 3, $A = 300$							
	b	$f(b)$	r	b^*	g	c	s
10k1	174	663	1.03	174*	1.91*	282*	372
10k2	125	125	0.00	71	7.67	114	125
10k3	1	897	0.06	16	0.00	1	719
total		1685					

Execution 4, $A = 50$							
	b	$f(b)$	r	b^*	g	c	s
10k1	9	979	1.43	174	1.91	282	372
10k2	40	701	6.77	71	7.67	114	125
10k3	1	897	0.06	16	0.00	1	719
total		2577					

Execution 5, $A = 100$							
	b	$f(b)$	r	b^*	g	c	s
10k1	19	944	1.37	174	1.91	282	372
10k2	80	402	6.15	80*	7.47*	117*	125
10k3	1	897	0.06	16	0.00	1	719
total		2243					

Figure 4: Allocation and Adaptation of MGR

adjusted from $\rho_{17} = (2.75, 196, 372)$ of Execution 1 to a more informed one $\rho_{28} = (2.62, 206, 372)$ of Execution 2. However, for 10k3, since the faulting characteristics at buffer size 1 are meaningless, the characteristics obtained from Execution 1 $\rho_{16} = (0.00, 1, 719)$ will remain unchanged.

At Execution 3, 300 buffers are available, according to the marginal gain ratios obtained form Execution 2, 10k1 is first allocated $300 * 2.62/(2.62 + 7.67) = 76$ buffers and 10k2 with $300 * 7.67/(2.62 + 7.67) = 223$ buffers. However, according to MGR, no allocation can exceed the saturated size, thus only 125 buffers will be allocated to 10k2. The remaining $223 - 125 = 98$ buffers are then redistributed among 10k1 and 10k3, but since 10k3 has zero marginal gain and critical size of 1, 10k1 receives all the remaining buffers and results in totally $76 + 98 = 174$ buffers. After the execution, the residual gain of 10k1 is 1.03, which is greater than the threshold, so the characteristic record is again adjusted from $\rho_{28} = (2.62, 206, 372)$ to $\rho_{174} = (1.91, 282, 372)$. The characteristic record of 10k2 is not changed since its residual gain $r = 0$ is less than the threshold.

At Execution 4, 50 buffers are available, MGR allocates buffers based on the adjusted characteristic records resulting from the prior three executions. As a result, it produces 2577 page faults, which is less than the 2727 page faults produced by Execution 1 where the same buffer size is provided but no feedback is available. No characteristic records are adjusted since the buffer size allocated to each relation for this execution is less than the buffer size associated with the best informed characteristic record. For example, for 10k2, the allocated buffer size $b = 40$ at Execution 4 is less than $b^* = 71$ at Execution 3.

And finally at execution 5, where 100 buffers are provided, it produces 2243 page faults, a number that is slightly smaller than 2276, the one produced at Execution 2 with the same available buffer size. This can be attributed to the adaptation of the characteristic record through Execution 2 and 3. Also note that the characteristic record of 10k2 is adjusted again after this execution, because its residual gain $r = 6.15$ is greater than the threshold and its allocated buffer size $b = 80$ is also greater than $b^* = 71$ (as recorded in Execution 4).

We also experimented the MG-x-y algorithm for the same sequence of executions above, for which it resulted in equal buffer allocation among the relations since the reference strings have equal length and are random on equal size of relations. The page faults are compared with those of MGR in Table 2. It can be seen that the adaptability of MGR reduces the page faults.

We also plot the faulting curves for MGR and MG-x-y for different buffer sizes, assuming that the above adjusted characteristic record is used for MGR. The optimal replacement algorithm OPT [Bel66], which replaces the page that won't be used in the longest future, is also graphed for comparison. Figure 5.(a) compares the number of page faults at different buffer sizes. Relative

	Exec. 1	Exec. 2	Exec. 3	Exec. 4	Exec. 5
available buffer size	50	100	300	50	100
page faults of MGR	2727	2276	1685	2577	2243
page faults of MG-x-y	2727	2572	1951	2727	2572

Table 2: Page Faults Summary for `MGR` and `MG-x-y`

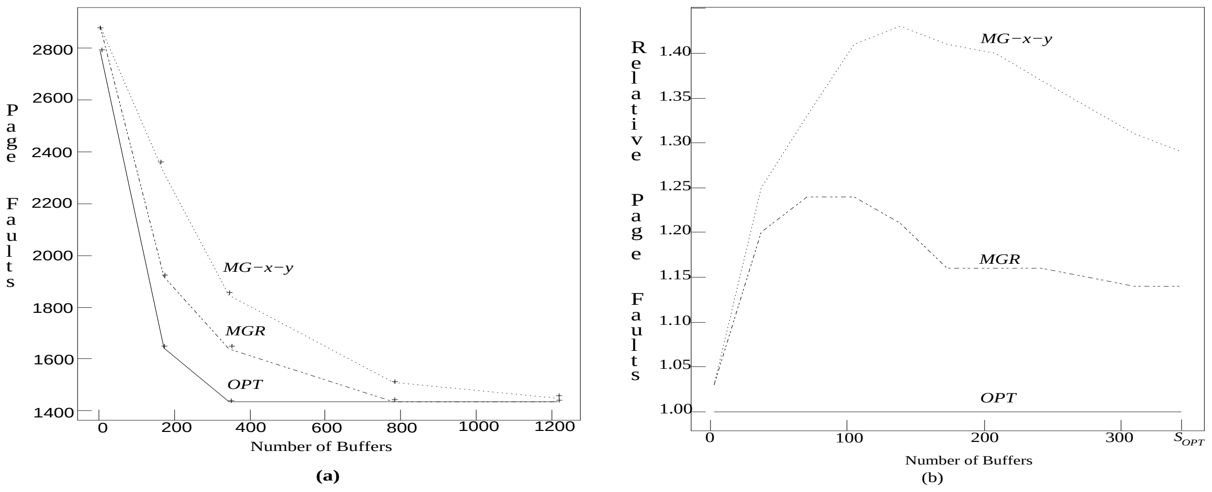


Figure 5: Page Faults Comparison among `MGR`, `MG-x-y` and `OPT`

page faults with respect to OPT are drawn in Figure 5.(b), with buffer size ranges from 1 to $s_{\texttt{OPT}}$, where $s_{\texttt{OPT}}$ is the saturated size under OPT replacement strategy. It is not hard to see that MGR performs much better than MG-x-y. Especially when buffer size is around 200, the page faults are reduced to half of the page faults over and above OPT.

5 Simulation Results

In this section we present a simulation for evaluating the performance of different database buffer management algorithms. The simulation is similar to the one used in [CD85, NFS91] which simulates a closed system with concurrent queries competing for buffers.

For the purpose of baseline comparison, LRU is selected as a representative since according to [EH84, CD85], it makes no significant performance difference from the other conventional strategies such as Working-Set and Clock. Two schemes, *local LRU* (LLRU) and *global LRU* (GLRU) are simulated. Local LRU maintains an LRU list for each relation of the concurrent queries. Global LRU manipulates the whole buffer pool under a single LRU list for all queries. There is no load control for global LRU, i.e., a query is admitted immediately as it arrives; for local LRU, a query is admitted only when there are still available buffers.

Since MG-x-y has been shown to outperform DBMIN using a flexible allocation [NFS91], we do not include DBMIN in the comparison. By trial and error, we have adjusted the values of x and y in MG-x-y so that it reached its best overall performance. Six pairs of (x, y) are experimented: (50, 100), (50, 200), (50, 400), (100, 100), (100, 200), and (100, 400). A query is admitted if the available buffers are more than the sum of each of its accessed relation's minimum buffer requirement. After a query is admitted, MG-x-y allocates as many buffers as possible to each relation, but not exceeding the specified upper bound. See Table 1 for detail.

Among the class of predictive load control algorithms [FNS91], the algorithm with the best overall performance, called EDU, is chosen as a representative in our simulation. Subject to the current available buffers, EDU activates a query only if it will result in better *effective disk utilization* than the one of the current state. After the query is admitted, it allocates buffers in the same way as MG-x-y does.

In MGR, we use a simple load controller based on FCM. As described earlier, a query is admitted only when the available buffers are more than half of the query's critical size. MGR uses the marginal gain ratios for buffer allocation for both queries and relations. It uses LRU replacement unless a looping pattern is detected, in which case MRU is used instead. During the simulation, MGR uses the adjusted faulting characteristics for each query. The overhead of characteristics feedback computations is also estimated and included in the simulation, though it is almost comparatively unperceptible to the query computations, according to our experiments on ADMS where MGR is implemented.

The reference strings are collected from executing a number of queries against the Wisconsin Benchmark database [BDT83] on ADMS. Each base relation contains 10,000 tuples spanning over 2,500 pages. The number of participating relations in each query varying from 1 to 3. Table 3 shows the access patterns for each of the queries we chose. To illustrate the impact of database buffer management, the queries have been chosen such that they access various numbers of distinct pages ranging from 100 to $1,500$. Q1 accesses 250 different pages, Q2 accesses 10 pages of the outer relation and has a looping access on a set of 98 pages of the inner relation. Q3 and Q4 have random accesses on their relations, with totally 1110 and 1416 distinct pages referenced, respectively.

Three different query mixes are used in the simulation and shown in Table 4. M1 simulates the situation where most of the reference strings are either sequential or looping, M2 simulates the situation where random references dominate, whereas in M3, all query types have the same

query type	no. of result tuples	no. of base relations	reference type
Q1	1000	1	*Sequential*
Q2	1000	2	*Sequential, Looping*
Q3	1000	2	*Random*
Q4	1000	3	*Random*

Table 3: Query Types

query mix	Q1	Q2	Q3	Q4
M1	40%	40%	10%	10%
M2	10%	10%	40%	40%
M3	25%	25%	25%	25%

Table 4: Query Mixes

frequency.

The number of concurrent queries (concurrency level) varied from 1 to 32. Initially, the queries are generated until the concurrency level is reached, and thereafter, no new query will be generated until a query finishes and leaves the system. Concurrent queries are scheduled by round robin for CPU. Unless mentioned otherwise, the size of the buffer pool is 1,000 pages. In all cases, the query mixes along with the configurations simulate an IO-bound closed system.

The simulations have been performed under different levels of data sharing. In *no data sharing*, all concurrent queries access different copies of the relations or completely different databases. In *partial data sharing*, every two of the concurrent queries access the same copy of database. And in *full data sharing*, all queries access the same database. The higher the sharing, the better buffer utilization due to the fact that pages in the buffers are used by several concurrent queries. However, due to the presence of concurrent queries on different copies of the database, even if a large number of buffers is available, it is not possible to load all the pages on demand into main memory simultaneously.

In the rest of this section, we interpret the results of the simulation. In all the presented figures, the *throughput* refers to the average number of queries finished per minute.

It should be pointed out that for the first time query runs, MGR simply uses MG-x-y strategy for allocation since no feedback is available yet. However, after the first query feedback, MGR uses the faulting characteristics to adjust the allocations for recurring queries. Because MGR uses more information about the behavior of the queries, it is expected to do better than all other techniques.

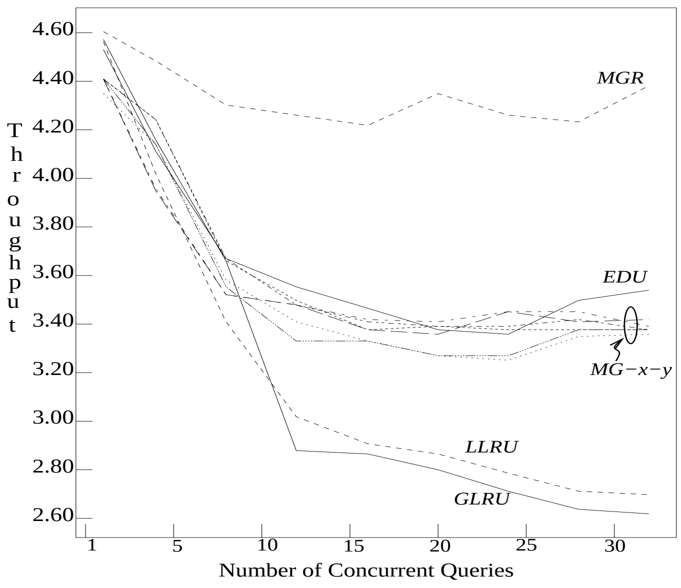


Figure 6: M3, no Data Sharing

Equal Frequency Query Mix

Figure 6 depicts the result of running query mix M3, where sequential and looping references occur as frequently as random references. In this case, MGR performs much better than all other strategies. The group of MG-x-y and EDU algorithms have similar performance which is significantly better than the LLRU and GLRU. As can be seen, MGR using FCM outperforms the probabilistic method used in MG-x-y and EDU, and the inferior LRUs.

Effect of Sequential and Looping References

Figure 7 compares the throughput of evaluating query mix M1 using different algorithms. Since sequential and looping references dominate, the use of pattern prediction and flexible allocation makes MG-x-y perform better than EDU. Load control does not has as much impact as buffer allocation in sequential and looping query mix. However, MGR performs even better than MG-x-y.

Effect of Random References

Query mix M2 simulates the effect of random references. Figure 8 shows the throughputs. In this case, the improvement of MG-x-y over LRUs is less substantial when compared to the previous figure. This is attributed to its inability to characterize random references. On the contrary, EDU now outperforms MG-x-y due to its ability of blocking a random reference when available buffers are not sufficient to increase or keep the system performance, thus avoid the

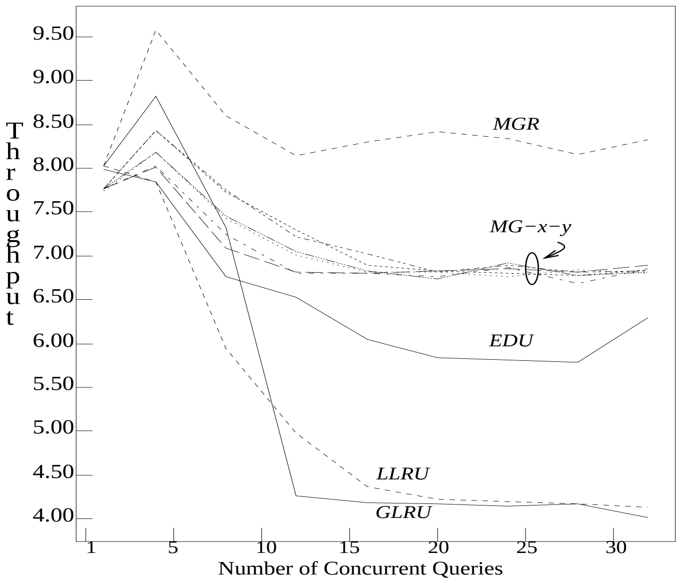


Figure 7: M1, no Data Sharing

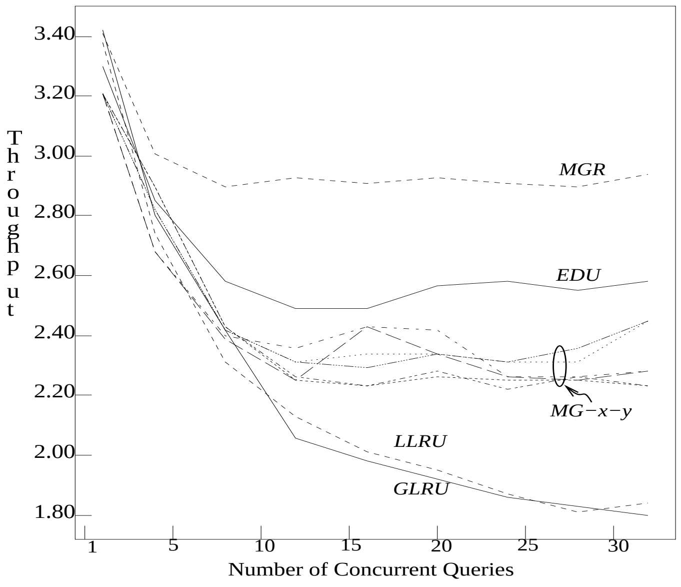


Figure 8: M2, no Data Sharing

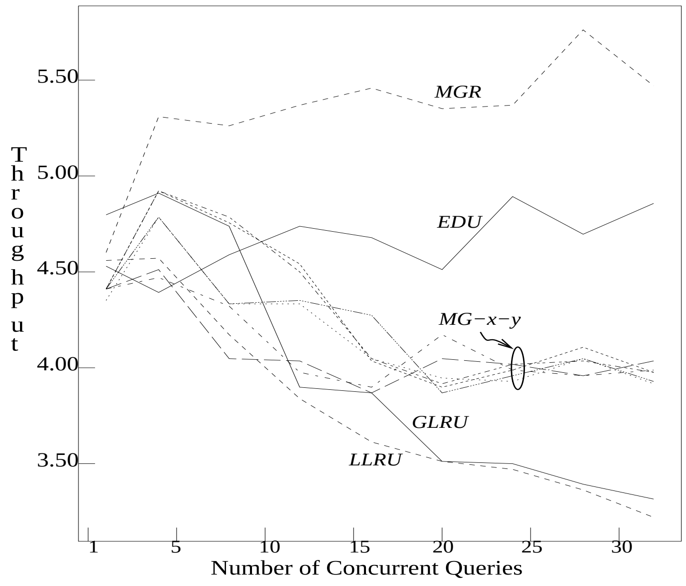


Figure 9: M3, Partial Data Sharing

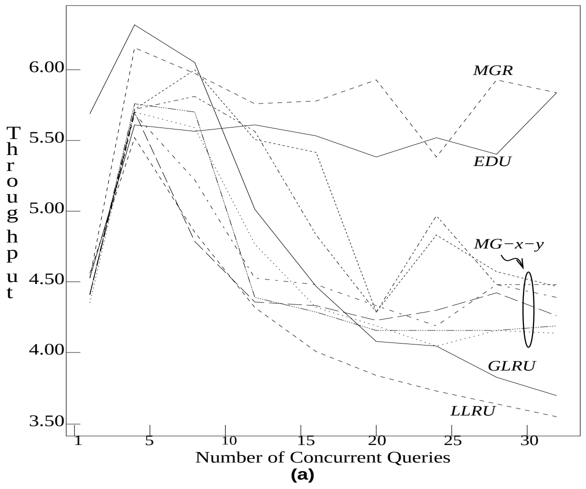
performance degradation. MGR, however, still provides substantial performance improvement, due to its ability to correctly characterize random access behavior.

Effect of Data Sharing

We used query mix M3 in this set of experiments. Figure 9 shows the result of partial data sharing. EDU now again performs fairly better than MG-x-y after concurrency level reaches 12. MGR remains a lot better than all the others.

Figure 10.a shows the effect of full data sharing. For concurrency levels between 1 to 8, GLRU outperform all other strategies including MGR. This is because in full data sharing, it is easy for global LRU to keep the *locality* sets of a few queries in buffers. However, when the number of concurrent queries increases, this advantage disappears because the overall locality set becomes too large to be accommodated by the buffer pool, and thus MGR and EDU win again. In this simulation, EDU is close to MGR, which indicates that when data sharing increases, the impact of buffer allocation decreases. Overall GLRU and MG-x-y perform roughly the same.

If the buffer pool gets smaller, from 1,000 to 600, the effect of full data sharing become less significant. This is shown in Figure 10.b, where GLRU now degrades drastically. Also, the performance improvement of MGR over EDU now increases again. This indicates that when buffer contention occurs, the buffer management algorithm which can characterize the reference behavior more accurately will result in better buffer utilization.



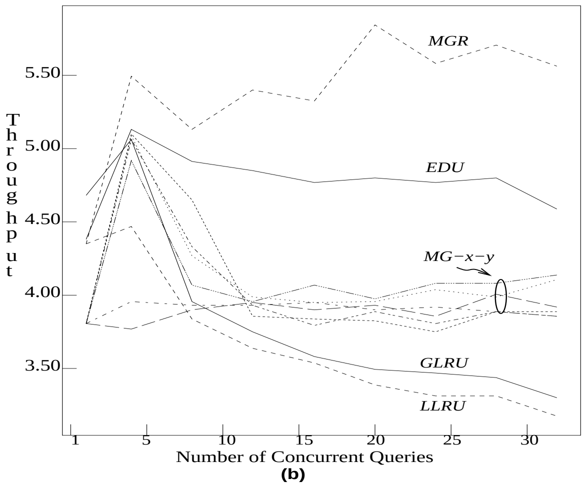


Figure 10: M3, Full Data Sharing, (a) 1000 buffers, (b) 600 buffers

To summarize, our simulation results show that MGR makes significant performance improvement over the pattern prediction style algorithm MG-x-y and the load-control-oriented algorithm EDU. In all cases of query mixes with no data sharing, MGR outperforms the second best strategy with 15% – 30% throughput improvement in average. We also observed the effect of data sharing, the results showed that, except for the cases of full data sharing with a very large buffer availability and small concurrency level, MGR is still favored. In sum, the significant performance of MGR over EDU and MG-x-y can be attributed to the advantage of using query feedback in adjusting buffer management which is more accurate than the pattern prediction and probabilistic methods.

6 Conclusion

In this paper, we propose the concept of using query execution feedback for improving database buffer management. A query feedback model which quantifies the page fault characteristics of all query access patterns including sequential, looping and most importantly random, is defined. Based on this model, a simple load controller and a buffer allocation scheme using marginal gain ratio are developed. The allocation scheme distributes the buffers among concurrent reference strings according to their quantified characteristics. An extensive set of simulations was conducted to compare the performance in throughputs of the proposed method with other existing ones. The simulations show that the proposed method is consistently better than the previous methods and in most cases, it significantly outperforms all other methods for random access reference patterns.

The advantage of MGR is the tuning of the buffer management techniques based on the real access behavior obtained by query feedback rather than probabilistic query path analysis where crude assumptions such as uniformity have to be made. Furthermore, since queries are treated as reference strings, our approach is applicable not only to relational algebra access paths but also to access paths of other more advanced database systems such as deductive and object-oriented databases.

References

[BDT83] D. Bitton, D.J. DeWitt, and C. Turbyfill. Benchmarking database systems, a systematic approach. In *Procs. of 9th VLDB*, 1983.

[Bel66] L. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[CD85] H. Chou and D. DeWitt. An evaluation of buffer management strategies for relational database systems. In *Proceeding of the 11th Intl. Conf. on VLDB*, pages 127–141, 1985.

[Che92] C. Chen. Adaptive query optimization. Thesis Proposal, Department of Computer Science, University of Maryland, College Park, Nov. 1992.

[EH84] W. Effelsberg and T. Haerder. Principles of database buffer management. *ACM TODS*, 9(4):560–595, 1984.

[FNS91] C. Faloutsos, R. T. Ng, and T. Sellis. Predictive load control for flexible buffer allocation. In *Proceeding of the 17th Intl. Conf. on VLDB*, pages 265–274, 1991.

[Kap80] J. Kaplan. Buffer management policies in a database environment. Master's thesis, University of California, Berkeley, 1980.

[M$^+$70] R. Mattson et al. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[NFS91] R. T. Ng, C. Faloutsos, and T. Sellis. Flexible buffer allocation based on marginal gains. In *Proceeding of 1991 ACM-SIGMOD Intl. Conf. on Management of Data*, pages 387–396, 1991.

[OOW93] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proceeding of 1993 ACM-SIGMOD Intl. Conf. on Management of Data*, pages 297–306, 1993.

[Rei76] A. Reiter. A study of buffer management policies for data management systems. Technical Report TR-1619, Mathematics Research Center, University of Wisconsin-Madison, 1976.

[RES93] N. Roussopoulos, N. Economou, and A. Stamenas. ADMS: A testbed for incremental access methods. *To appear in IEEE Trans. on Knowledge and Data Engineering*, 1993.

[RR76] J. Rodriguez-Rosell. Empirical data reference behavior in data base systems. *IEEE Computer*, 9(11), Nov. 1976.

[SB76] S.W. Sherman and R.S. Brice. Performance of a database manager in a virtual memory system. *ACM TODS*, 1(4), 1976.

[SS82] G. Sacca and M. Schkolnick. A mechanism for managing the buffer pool in a relational database system using the hot set model. In *Proceeding of the 8th Intl. Conf. on VLDB*, pages 257–262, 1982.

[SS86] G. Sacca and M. Schkolnick. Buffer management in relational database systems. *ACM TODS*, 11(4):474–498, 1986.

[Tue76] W. Tuel. An analysis of buffer paging in virtual storage systems. *IBM Journal of Research and Development*, 1976.

[Yao77] S.B. Yao. Approximating block accesses in database organizations. *Communications of ACM*, 20(4), 1977.