Flexible Buffer Allocation Based on Marginal Gains*

Raymond Ng, Christos Faloutsos and Timos Sellis†

Department of Computer Science
University of Maryland
College Park, Maryland 20742

Abstract

Previous works on buffer allocation are based either exclusively on the availability of buffers at runtime or on the access patterns of queries. In this paper we propose a unified approach for buffer allocation in which both of these considerations are taken into account. Our approach is based on the notion of *marginal gains* which specify the expected reduction on page faults in allocating extra buffers to a query. Simulation results show that our approach is promising, and allocation algorithms based on marginal gains perform considerably better than existing ones.

1 Introduction

Early studies on buffer allocation in database management systems focus on adapting memory management techniques used in operating systems [5, 9, 12]. In their frameworks, allocation is based only on the availability of buffers at runtime. On the other hand, recent works on buffer management [2, 11] focus exclusively on the access patterns of queries, and they show that their proposals give better performance results than the former approaches.

In this paper we propose a unified approach for buffer allocation in which both the access patterns of queries and the availability of buffers at runtime are taken into account. The objective is to provide the best possible use of buffers; a buffer is well-used if its allocation results in many page hits. The basis of our approach is the notion of *marginal gains* which specify the expected number of page hits that would be obtained in allocating extra buffers to a query. In the first half of this paper, we describe how to compute marginal gains based on the mathematical models we establish for each type of accesses patterns exhibited by relational operations. Then we propose a class of buffer allocation algorithms that make use of marginal gains to handle allocation. Simulation results indicate that our approach is promising and our algorithms perform considerably better than the existing ones.

In section 2 we review related work and motivate the research described in this paper. In section 3 we present mathematical models for computing the expected number of page faults for each type of database references. Then based on those models, we introduce in section 4 the notion of marginal gains. In section 5 we describe a class of buffer allocation algorithms based on marginal gains. Finally, we present in section 6 simulation results that compare the performance of our algorithms with other allocation methods. We conclude with a discussion on ongoing research.

2 Related Work

In relational database management systems, the buffer manager is responsible for all the operations on buffers, including buffer assignment to queries, replacement decisions and buffer reads and writes in the event of page faults. As outlined in Figure 1, it is also implicitly responsible for load control. When buffers are available, the manager needs to decide whether to activate a query in the waiting queue and how many buffers to allocate to that query. While these decisions may depend on many factors, previous proposals on buffer allocation focus on the availability of buffers at runtime and the access patterns of queries. These

*This research was partially sponsored by the National Science Foundation under Grants DCR-86-16833, IRI-8719458, IRI-8958546 and IRI-9057573, by DEC and Bellcore, by the Air Force Office for Scientific Research under Grant AFOSR-89-0303, and by University of Maryland Institute for Advanced Computer Studies (UMIACS).

†Christos Faloutsos and Timos Sellis are also affliated with UMIACS.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1991 ACM 0-89791-425-2/91/0005/0387...$1.50

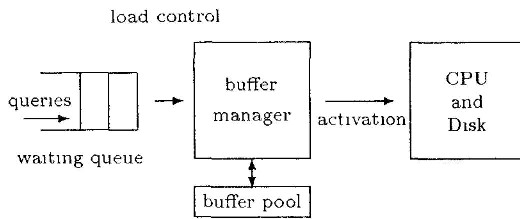

Figure 1: Buffer Manager and Related Components

proposals can be classified into two groups.

Allocation algorithms in the first group consider only the availability of buffers at runtime. They include early studies on database buffer management[5, 9, 12]. Their focus is on adapting memory management techniques used in operating systems to database systems. Those techniques include variations of First-In-First-Out (FIFO), Random, Least-Recently-Used (LRU), Clock, and Working-Set. However, as they fail to take advantage of the specific access patterns exhibited by relational database queries, they do not perform satisfactorily[2].

Allocation strategies in the second group consider exclusively the access patterns of queries. In the implementation of INGRES[13], Kaplan[7] proposes allocation on a relation basis. But Sacca and Schkolnick[11] were the first to analyze the reference patterns of relational queries and incorporate such knowledge into buffer allocation based on a Hot-Set model. This approach of buffer allocation is culminated in the work of Chou and DeWitt[2]. They introduce the notion of a locality set of a query, i.e. the number of buffers a query needs to use without many page faults. They propose the DBMIN algorithm that makes allocation equal to the size of the locality set. Simulation results show that DBMIN outperforms the Hot-Set algorithm and the algorithms referred to in the first group. Cornell and Yu[4] recently propose integrating buffer management with query optimization. They also determine the buffer requirements of queries based on their access patterns, without considering the actual availability of buffers at runtime. Table 1 summarizes the two groups of buffer allocation strategies discussed above.

While the strength of DBMIN and other algorithms referred to in the second group lies in their consideration of the access patterns of queries, their weakness arises from their oblivion of runtime conditions, such as the availability of buffers. This imposes heavy penalties on the performance of the whole system, such as low buffer utilization and throughput. For example, suppose the access pattern of a nested loop requires 18

algorithms	access patterns of queries	availability of buffers at runtime
FIFO, Random, LRU, Clock, Working-Set	–	√
Hot-Set, DBMIN	√	–
proposed algorithms	√	√

Table 1: Classification of Allocation Algorithms

buffers to be allocated. Now even if there are 17 buffers available, DBMIN still insists that this query waits for 18 buffers to be available. Thus, 17 buffers are left idle. In situations where resources are scarce and buffers are precious, idle buffers should be minimized.

On the other hand, consider the allocation for queries requiring random references, such as selections using non-clustered index. DBMIN allocates 1 buffer to a random reference[1]. Suppose there are 5 buffers available and the only query in the waiting queue is a random query. Again since DBMIN does not consider the availability of buffers, DBMIN assigns 1 buffer to the reference and lets 4 buffers idle.

These problems lead us to study a unified approach in buffer allocation which simultaneously takes into account the access patterns of queries and the availability of buffers at runtime. Considering the above examples again, our aim is to design algorithms that have the flexibility of allocating 17 buffers to the first query and 5 buffers to the random query. But then based on what criteria are those allocations justified? In particular, when query A is competing with query B for buffers, how can the decision of allocating buffers to A but not to B be justified?

The answer we proposed is to give allocations that make the best use of buffers. Intuitively, a buffer is well-used if its allocation results in many page hits. Thus, in next section we first present mathematical models analyzing the expected number of page faults for relational database references. Based on those formal models, we develop the notion of marginal gains which quantify how well a buffer is used.

3 Models for References

In this section we first review the taxonomy proposed by Chou and DeWitt[2] for classifying reference pat-

[1] Under circumstances very rare in practice, DBMIN may allocate b buffers to a random reference, where b is the Yao estimate on the average number of pages referenced in a series of random record accesses [14]. In reality, Yao estimates are usually too high for allocation. For example, for a blocking factor of 5, the Yao estimate of accessing 100 records of a 1000-record relation is 82 pages. Thus, DBMIN almost always allocates 1 buffer to a random reference.

terns exhibited by relational database queries. We analyze in detail the major types of references, and present mathematical models and formulas calculating the expected number of page faults using a given number of buffers. Based on these models, we formalize the notion of marginal gains in the next section.

3.1 Types of Reference Patterns

In [2] Chou and DeWitt show how page references of relational database queries can be decomposed into sequences of simple and regular access patterns. They identify four major types of references: random, sequential, looping and hierarchical. A random reference consists of a sequence of random page accesses. A selection using a non-clustered index is one example. The following definitions formalize this type of references.

Definition 1 A *reference Ref* of length k to a relation is a sequence $< P_1, P_2, \ldots, P_k >$ of pages of the relation to be read in the given order. □

Definition 2 A *random reference* $\mathcal{R}_{k,N}$ of length k to a relation of size N is a reference $< P_1, \ldots, P_k >$ such that for all $1 \leq i, j \leq k$, P_i is uniformly distributed over the set of all pages of the accessed relation, and P_i is independent of P_j for $i \neq j$. □

In a sequential reference, such as a selection using a clustered index, pages are referenced and processed one after another without repetition.

Definition 3 A *sequential reference* $\mathcal{S}_{k,N}$ of length k to a relation of size N is a reference $< P_1, \ldots, P_k >$ such that for all $1 \leq i, j \leq k \leq N$, $P_i \neq P_j$. □

When a sequential reference is performed repeatedly, such as in a nested loop join, the reference is called a looping reference.

Definition 4 A *looping reference* $\mathcal{L}_{k,t}$ of length k is a reference $< P_1, \ldots P_k >$ such that for some $t \leq k$,
i) $P_i \neq P_j$, for all $1 \leq i, j \leq t$, and
ii) $P_{i+t} = P_i$, for $1 \leq i \leq k - t$
The subsequence $< P_1, \ldots, P_t >$ is called the *loop*, and t is called the *length* of the loop. □

Finally, a hierarchical reference is a sequence of page accesses that form a traversal path from the root down to the leaves of an index. In [10] we show how the analyses for random, sequential and looping references can be applied to hierarchical references and other more complex types of references. Hence, due to space limitations, here we present mathematical models only for the three types of references. In particular we give formulas for computing the expected number of page faults using a given number of buffers s.

Definition 5 Let $Ef(Ref, s)$ denote the expected number of page faults caused by a reference Ref using s buffers, where Ref can be $\mathcal{L}_{k,t}, \mathcal{R}_{k,N}$ or $\mathcal{S}_{k,N}$. □

3.2 Random References

Throughout this section, we use $P(f, k, s, N)$ to denote the probability that there are f faults in k accesses to a relation of size N using s buffers, where $s \geq 1$ and $0 \leq f \leq k$. Thus for a random reference, the expected number of page faults is given by:

$$Ef(\mathcal{R}_{k,N}, s) = \sum_{f=1}^{k} f * P(f, k, s, N) \qquad (1)$$

To model a random reference, we set up a Markov chain in the following way. A state in the Markov chain is of the form $[f, k]$ indicating that there are f faults in k accesses for $f \leq k$. In setting up the transitions from states to states, there are two cases to deal with. In the first case, the number f of faults does not exceed the number s of allocated buffers. Thus, there must be f distinct pages kept in the buffers after f faults. Now consider a state $[f, k]$ in the chain. There are two possibilities to have f faults in k accesses. If the last access does not cause a page fault, that is with a probability f/N, then there must be f faults in $(k-1)$ accesses. In other words, there is an arc from state $[f, k-1]$ to state $[f, k]$ with a transition probability of f/N. The other arc to state $[f, k]$ is from state $[f-1, k-1]$ with a transition probability of $(N - f + 1)/N$. This corresponds to the case when there are $(f-1)$ faults in $(k-1)$ accesses, and the last page accessed is not one of the $(f-1)$ pages being kept in the buffers. Hence, the case for $f \leq s$ is summarized by the following recurrence equation:

$$P(f, k, s, N) = f/N * P(f, k-1, s, N) + (N - f + 1)/N * P(f-1, k-1, s, N), \; f \leq s \qquad (2)$$

In the second case, the number f of faults exceeds the number s of allocated buffers. Local replacement[2] must have taken place, and there are always s pages kept in the buffers. Similar to the above analysis for the case $f \leq s$, the situation for $f > s$ is summarized by the following recurrence equation:

$$P(f, k, s, N) = s/N * P(f, k-1, s, N) + (N - s)/N * P(f-1, k-1, s, N), \; f > s \qquad (3)$$

[2] Since the reference is random, the choice of local replacement policies is irrelevant.

In addition, the base case is $P(0,0,s,N) = 1$ for all $s \geq 1$. Thus, the expected number of page faults $Ef(\mathcal{R}_{k,N}, s)$ can be computed according to Equation 1. Unfortunately, except for the case where $s = 1$, we do not have a simple closed form formula for $Ef(\mathcal{R}_{k,N}, s)$. But the formula below gives very close approximations to the actual values:

$$Ef(\mathcal{R}_{k,N}, s) \approx \begin{cases} N * [1 - (1 - 1/N)^k], & k < k_0 \\ s + (k - k_0) * (1 - s/N), & k \geq k_0 \end{cases} \quad (4)$$

where $k_0 = ln(1 - s/N)/ln(1 - 1/N)$. Intuitively, k_0 is the expected number of page accesses that fill all the s buffers. Thus, the top row of the formula corresponds to the case where none of the buffers that have been filled needs to be replaced. This first case uses Cardenas' formula [1] which calculates the expected number of distinct pages accessed after k random pages have been selected out of N possible ones with replacement. More accurate results may be obtained with Yao's formula [14], which assumes no replacement. All these formulas make the uniformity assumption; its effects are discussed in [3]. The second row corresponds to the case when local replacement has occurred. Then, s faults have been generated to fill the s buffers (which take k_0 page accesses on the average); for the remaining $(k - k_0)$ requests, the chance of finding the page in the buffer pool is s/N.

3.3 Sequential References

Recall from Definition 3 that each page in a sequential reference $\mathcal{S}_{k,N}$ is accessed only once. Thus, the probability of a page being re-referenced is 0. Hence, a sequential reference can be viewed as a degenerate random reference, and the formula below is obvious:

$$Ef(\mathcal{S}_{k,N}, s) \quad = \quad k \qquad (5)$$

3.4 Looping References

Recall from condition (i) of Definition 4 that within a loop, a looping reference $\mathcal{L}_{k,t}$ is strictly sequential. Thus, based on Equation 5, t page faults are generated in the first iteration of the loop. Then there are two cases. Firstly, if the number s of allocated buffers is not less than the length t of the loop, all pages in the loop are retained in buffers, and no more page faults are generated in the remainder of the reference. The choice of a local replacement policy is irrelevant in this case.

In the second case, if the number s of allocated buffers is less than the length t of the loop, the local replacement policy plays a major role in determining the number of page faults generated by a looping reference. In [10] we present analytical results for looping references under the replacement policies: First-In-First-Out, Random, Fix-Set and Most-Recently-Used (MRU). We also prove that MRU is the optimal replacement policy for looping references. Thus, here we present the results for MRU only. The analysis of MRU can be best explained by an example.

Example 1 Consider a looping reference with the loop $< a, b, c, d, e >$. Suppose $s = 3$ buffers are available for the reference. The following table summarizes the situation under MRU.

1	2	3	4	5	6	7	8	9	10	11	12	13
a	b	c	d	e	a	b	c	d	e	a	b	c
					*	*			*	*		
a	b	c	d	e	a	b	c	d	e	a	b	c
	a	b	b	b	e	a	a	a	d	e	e	e
		a	a	a	b	e	e	e	a	d	d	d

14	15	16	17	18	19	20	21	22	23	24	25
d	e	a	b	c	d	e	a	b	c	d	e
*	*			*	*			*	*		
d	e	a	b	c	d	e	a	b	c	d	e
c	d	d	d	b	c	c	c	a	b	b	b
e	c	c	c	d	b	b	b	c	a	a	a

The first row of the table indicates the numbers of page accesses. The second row shows the order the pages are accessed for five iterations of the loop. If a page hit occurs, the access is marked with an asterisk. The last three rows of the table indicate the pages kept in the buffers after that page access, with the page most frequently used in the top row.

This example demonstrates a few important properties of MRU. First note that there are five "mini-cycles" of length four which may not align with the iterations of the loop. They are separated by vertical lines in the table above. These mini-cycles also follow a cyclic pattern, namely the twenty-sixth access of the table will be exactly the same as the sixth access, and so on. Furthermore, within each mini-cycle, there are two "resident" pages – those that are not swapped out in that mini-cycle. For instance, for the first mini-cycle, the resident pages are a and e. Note that these resident pages are the pages that begin the next mini-cycle, avoiding page faults for those accesses; this property is exactly the reason why MRU is optimal. □

In general, given a loop of length t, the mini-cycles are of length $(t-1)$. In other words, in $(t-1)$ iterations of the loop, there are t different mini-cycles. Furthermore, these mini-cycles recur every $(t-1)$ iterations of the loop. Then in each mini-cycle, there are $(s-1)$ resident pages. Thus, there are $(t-s)$ faults in each mini-cycle. Hence, on the average, there are $(t-s) * t \, / \, (t-1)$ faults in each iteration of the loop.

Thus, the equation below follows immediately:

$$Ef(\mathcal{L}_{k,t}, s) = t + (t - s) * t * (k/t - 1)/(t - 1), \quad s \leq t \qquad (6)$$

Thus far, we have described analytical models for random, sequential and looping references. We have also presented formulas for computing the expected number of page faults using a given number of buffers. We are now in a position to introduce the key concept of this paper – *marginal gains*[3].

4 Marginal Gains

Definition 6 For $s \geq 2$, the *marginal gain* of a reference Ref using s buffers is defined as:

$$mg(Ref, s) = Ef(Ref, s-1) - Ef(Ref, s),$$

where Ref can be $\mathcal{L}_{k,t}, \mathcal{R}_{k,N}$ and $\mathcal{S}_{k,N}$. □

For any reference Ref, the marginal gain value $mg(Ref, s)$ specifies the expected number of extra page hits that would be obtained by increasing the number of allocated buffers from $(s-1)$ to s. Thus, marginal gain values specify quantitatively how efficiently a reference uses its buffers. Moreover, this quantification is at a granularity level finer than the locality set sizes used in DBMIN. For instance, a buffer allocation algorithm based on marginal gains can now decide whether it is beneficial to allocate 17 buffers to a looping reference which needs 18. In contrast, DBMIN can only allocate on a per locality-set-size basis, and can only wait for 18 buffers to be available. Similarly, when there are idle buffers, an allocation algorithm based on marginal gains may allocate to a random query more buffers than specified by its locality set size, provided that the marginal gain values of the random query are positive. While in the next section we present buffer allocation algorithms based on marginal gains, here we first analyze how the marginal gain values vary with the number of buffers. This analysis is crucial in designing the allocation algorithms to be presented.

To a looping reference $\mathcal{L}_{k,t}$, it is straightforward to see from Equation 6 that for any allocation $s < t$, extra page hits would be obtained by allocating more and more buffers to the reference, until the loop can be fully accommodated in the buffers. The allocation $s = t$ is the optimal allocation that generates the fewest page faults. Furthermore, any allocation $s > t$ is certainly wasteful, as the extra buffers are not used. The graph for looping references in Figure 2 summarizes the situation. The typical marginal gain values of looping references are in the order of magnitude of $O(10)$ or $O(10^2)$.

As far as looping references are concerned, DBMIN certainly agrees with our analysis in requiring the locality set size to be the length of the loop. However, one weakness of DBMIN lies in its insistence on optimal allocation for looping references. Such inflexibility may impose heavy penalties on performance.

Similarly, based on Equations (2)-(4), it is easy to check that the marginal gain values of random references are positive, but are strictly decreasing as the number of allocated buffers s increases, as shown in Figure 2. Eventually, the marginal gain value becomes zero, when the allocation exceeds the number of accesses or the number of pages in the accessed relation.

To random references, DBMIN allocates either 1 or b buffers where b is the Yao estimate on the average number of pages referenced in a series of random record accesses[14]. Again DBMIN is not flexible enough to take advantage of available buffers at runtime. In contrast, a buffer allocation algorithm based on marginal gains may allocate the idle buffers to the random reference, as long as the marginal gain values of the reference indicate that there are benefits to allocate more buffers to the reference. In fact, even if the number of idle buffers exceeds the Yao estimate, it may still be beneficial to have an allocation beyond the Yao estimate. It is however worth pointing out that the marginal gain values of a random reference are normally lower than those of a looping reference. The highest marginal gain value of a random reference is typically in the order of magnitude of $O(1)$ or $O(10^{-1})$.

Finally, as shown in Equation 5, the marginal gain values of sequential references are always zero, indicating that there is no benefit to allocate more than one buffer to such references (cf. Figure 2).

In this section we have formalized the notion of marginal gains and analyzed how the marginal gain values vary with the number of buffers. Next we present buffer allocation algorithms based on the analysis presented here.

5 Allocation Algorithms Based on Marginal Gains

As we have shown in the previous section, the marginal gain values of a reference quantify the benefits of allocating extra buffers to the reference. Thus, in a system where queries compete for a fixed number of

[3]The concepts of marginal gain and marginal utility have been widely used in ecomonics theory since the 18th century[8]. Here we merely apply the concept to database buffer allocations.

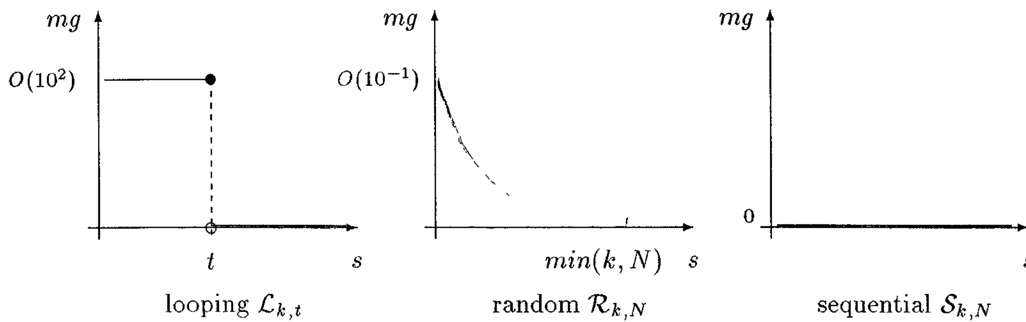

Figure 2: Typical Curves of Marginal Gain Values

buffers, the marginal gain values provide a basis for a buffer manager to decide which queries should get more buffers than others. Ideally, given N free buffers, the best allocation is the one that does not exceed N and that maximizes the total marginal gain values of queries in the waiting queue. However, such an optimization will be too expensive and complicated for buffer allocation purposes. Furthermore, to ensure fairness, we favor buffer allocation on a First-Come-First-Serve basis. In the following we present a class MG-x-y of allocation algorithms that are fair, easy to compute, and that achieve high marginal gain values.

5.1 MG-x-y

MG-x-y is a class of buffer management algorithms very similar to DBMIN. The only, but very important, difference between MG-x-y and DBMIN lies in the initial decision at load control time on the number of buffers to be allocated to each query (cf. Figure 1). In DBMIN, this number of buffers is always the locality set size. In MG-x-y, in addition to the locality set size, the number of available buffers at load control time is also taken into consideration. The algorithm shown below describes how MG-x-y makes such allocation decisions. Once the number of buffers to be allocated to a query is decided, MG-x-y operates in exactly the same way as DBMIN. See [2] for more details, such as the handling of page faults, and so on.

Whenever buffers are released by a newly completed query, MG-x-y invokes the algorithm below to try to activate the query at the head of the waiting queue by making a positive buffer allocation to the query. As long as there are buffers left, MG-x-y invokes this algorithm repeatedly, until the query then at the head of the waiting queue gets an allocation of zero buffers. Then all the queries in the waiting queue must wait for more buffers to be released (i.e. strictly First-Come-First-Serve). Recall from previous sections that we identify sequential, random and looping references as the most basic references. For the ease of presentation, the following algorithm only considers these three types of references.

Algorithm Allocate Let R be the reference at the head of the waiting queue, and $A > 0$ be the number of available buffers. Moreover, let x and y be the parameters of MG-x-y to be explained in detail shortly.
Case 1: R is a looping reference $\mathcal{L}_{k,t}$.
If the number A of available buffers exceeds the length t of the loop, allocate t buffers to the reference. Otherwise, if the number of available buffers is too low (i.e. $A < (x\% * t)$), allocate no buffers to this reference. Otherwise (i.e. $A \geq (x\% * t)$), give all A buffers to the reference R.
Case 2: R is a random reference $\mathcal{R}_{k,N}$.
As long as the marginal gain values of R are positive, allocate to R as many buffers as possible, but not exceeding the number A of available buffers and y (i.e. allocation $\leq$ minimum (A, y)).
Case 3: R is a sequential reference $\mathcal{S}_{k,N}$.
Allocate 1 buffer. □

5.2 Parameters of MG-x-y

MG-x-y has two parameters, x and y. The x parameter is used in determining allocations for looping references. As described in Case 1 above, MG-x-y first checks to see if the number of available buffers exceeds the length of the loop of the looping reference. Recall from the previous section and Figure 2 that the allocation which accommodates the whole loop minimizes page faults and corresponds to the highest total marginal gain values of the reference. Thus, if there are enough buffers, then like DBMIN, MG-x-y

query type	query operators	selectivity	access path of selection	join method	access path of join	reference type
I	select(A)	10 %	clustered index	–	–	$\mathcal{S}_{50,500}$
II	select(B)	10 %	non-clustered index	–	–	$\mathcal{R}_{30,15}$
III	select(A) ⋈ B	1 %	sequential scan	index join	non-clustered index on B	$\mathcal{R}_{100,15}$
IV	select(A) ⋈ B	4 %	clustered index	nested loop	sequential scan on B	$\mathcal{L}_{300,15}$

Table 2: Summary of Query Types

gives the optimal allocation. However, if there are not enough buffers, MG-x-y checks to determine whether a sub-optimal allocation is beneficial, via the use of parameter x.

In general, the response time of a query has two components: the waiting time and the processing time, where the former is the time from the arrival of the query to the time the query is activated, and the latter is the time from activation to completion. The processing time is minimized with the optimal allocation. But to obtain the optimal allocation, the waiting time may become too long. On the other hand, while a sub-optimal allocation may result in longer processing time, it may at the end give a response time shorter than the optimal allocation, if the reduction in waiting time more than offsets the increase in processing time. Hence, in trying to achieve this fine balance between waiting time and processing time, MG-x-y uses the heuristic that a sub-optimal allocation is only allowed if the total marginal gain values of that allocation is not too "far" away from the optimal. This requirement translates to the condition shown in Case 1 that a sub-optimal allocation must be at least x% of the optimal one.

While the inflexible DBMIN cannot take advantage of available buffers for allocation to random references, MG-x-y may allocate extra buffers to a random reference, as long as those extra buffers are justified by the marginal gain values of the reference. However, there is a pitfall simply considering only the marginal gain values of the random reference. As an example, suppose a random reference is followed by a looping reference in the waiting queue. In situations where buffers are scarce, giving one more buffer to the random reference implies that there is one fewer buffer to give to the looping reference. But since the marginal gain values of a looping reference are usually higher than those of a random reference, it is desirable to save the buffer from the random reference and to allocate the buffer to the looping reference instead. Since MG-x-y operates on a First-Come-First-Serve basis, MG-x-y uses the heuristic of imposing a maximum on the number of buffers allocated to a random reference. This is the purpose of the y parameter in MG-x-y.

relation A	10,000 tuples
relation B	300 tuples
tuple size	182 bytes
page size	4K

Table 3: Details of Relations

Notice that MG-100-1 is the same as DBMIN. Thus, the class MG-x-y generalizes DBMIN by allowing more flexibility in allocation. But note that for the research described in this paper, it is not our focus to find the best x and y parameters for MG-x-y for a given mix of queries. Instead, our main purpose is to show that it is beneficial to perform flexible buffer allocation based on marginal gains which incorporate in the allocation both the access patterns of queries and the availability of buffers at runtime. Methods of determining the best values of x and y are a topic of ongoing research.

6 Simulation Results

In this section we present simulation results on the performance of MG-x-y in a multiuser environment. As Chou and DeWitt have shown in[2] that DBMIN performs better than the Hot-Set algorithm, First-In-First-Out, Clock, Least-Recently-Used and Working-Set, we only compare MG-x-y with DBMIN.

6.1 Details of Simulation

In order to make direct comparison with DBMIN, we use the simulation program Chou and DeWitt used for DBMIN, and we experiment with the same types of queries. Table 2 summarizes the details of the queries that are chosen to represent varying degrees of demand on CPU, disk and memory [2]. Table 3 and Table 4 show the details of the relations and the query mixes used in the simulation respectively. In the simulation, the number of concurrent queries (denoted as the multiprogramming level in the figures shown in this sec-

	I	II	III	V
mix 1	10%	45%	45%	–
mix 2	10%	10%	10%	70%

Table 4: Summary of Query Mixes

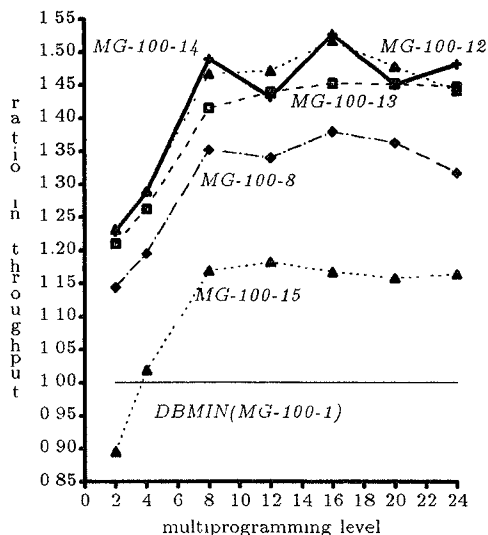

Figure 3: Mix 1, no Data Sharing

tion) varies from 2 to 16 or 24. Each of these concurrent queries is generated by a query source which cannot generate a new query until the last query from the same source is completed. Thus, the simulation program simulates a closed system. See [2] for more details.

6.2 Effectiveness of Allocations to Random References

The first mix of queries consists of 45% of queries of type II, 45% of queries of type III and 10% of queries of type I. The purpose of this mix is to evaluate the effectiveness of the policy of MG-x-y on allocating buffers to random references (e.g. query II and III). Since there are no looping references in this mix, the x parameter of MG-x-y is irrelevant and is simply set to 100. The y parameter is one of the following: 1 (DBMIN), 8, 12, 13, 14 and 15. Figure 3 shows the throughput at various multiprogramming levels using 35 buffers. To highlight the increase or decrease relative to DBMIN, the throughput values are normalized by the values of DBMIN, i.e. MG-x-y / DBMIN.

Compared with DBMIN, all other MG-100-y algorithms show a significant increase in throughput. As the y value increases from 1 to 15, the throughput increases gradually until y becomes 15. The increase in

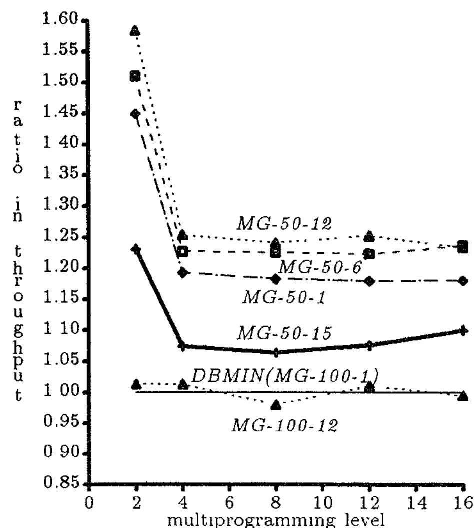

Figure 4: Mix 2, no Data Sharing

throughput can be attributed to the fact that the random queries are benefited by the allocation of more buffers. But when too many buffers (e.g. $y = 15$) are allocated to a random query, some of the buffers are not used as efficiently as they would have been, had they been allocated to other queries.

6.3 Effectiveness of Allocations to Looping References

The second mix of queries consists of 70% of queries of type IV and 10% each of queries of types I, II and III. The purpose of this mix is to evaluate the performance of MG-x-y in situations where there are many looping references to be executed. The x parameter of MG-x-y is set to one of the following: 100, 85, 70 and 50. The y parameter is one of 1, 6, 12 and 15. Figure 4 shows the throughput of MG-100-1, MG-100-12 and MG-50-y's at various multiprogramming levels using 35 buffers[4]. Again, the throughput values are normalized by the values of DBMIN.

All four MG-50-y algorithms show considerable improvement when compared with DBMIN. In particular, since the allocations for random and sequential references are the same for both MG-50-1 and MG-100-1, the improvement exhibited by MG-50-1 relative to MG-100-1 is due solely to the effectiveness of allocating buffers sub-optimally, whenever necessary, to looping references. As explained in the previous section, the improvement shown by MG-50-6 and MG-50-12 with respect to MG-50-1 is due to more effective allocations to random references. The reason why little improve-

[4] The results for MG-70-y's and MG-85-y's are similar to those for MG-50-y's, and they are omitted for brevity.

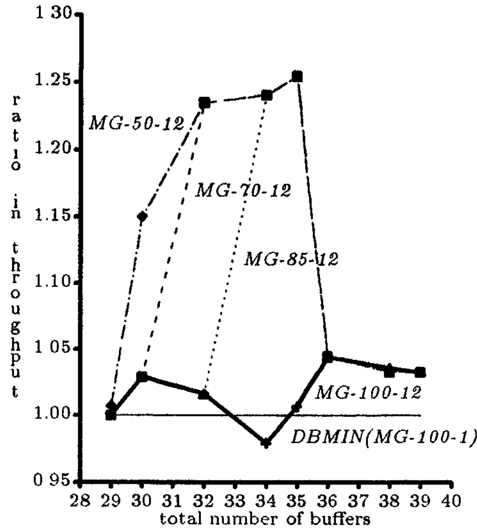

Figure 5: Throughput vs Buffers (mpl = 8)

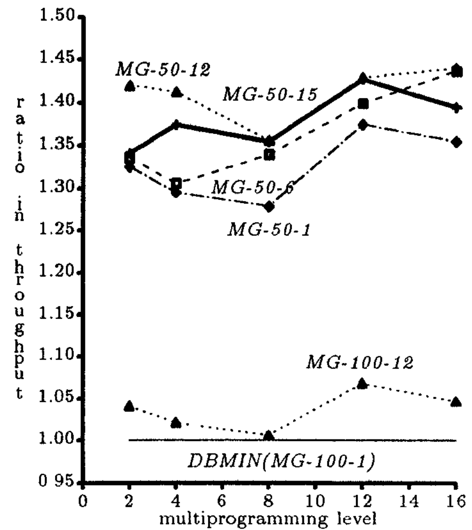

Figure 6: Mix 2, full Data Sharing

ment is shown between MG-50-12 and MG-50-1 is that there is only a small percentage of random queries in this mix. Finally, for reasons described in the previous subsection, MG-50-15, though performing better than DBMIN, does not perform as well as the other MG-50-y algorithms.

Since the Yao estimates for the random queries used are 12, the algorithm MG-100-12 to a certain extent represents the "optimal" algorithm, as it allocates buffers to minimize the number of page faults of the reference. However, such optimal allocations may induce a high waiting time for queries, resulting in low buffer utilization and throughput of the system. See [10] for graphs comparing the average waiting time of queries and buffer utilization under MG-x-y algorithms and DBMIN.

Thus far, we have seen how the performance of MG-x-y varies with different values of x and y. Figure 5 shows how the relative throughput varies with the number of total buffers used in running this mix of queries at a multiprogramming level of 8 [5]. It shows that when the number of total buffers becomes 30, MG-50-12 allows sub-optimal allocation to a looping reference, and the throughput of the system increases significantly. As the total number of buffers increases, MG-70-12 and MG-85-12 follow MG-50-12 and perform better than DBMIN and MG-100-12. The discrepancy is due to the insistence of DBMIN and MG-100-12 on the optimal allocation to a looping reference (18 in this case), which is blocking other queries from using the buffers. Now when this reference finally manages to get the optimal number of buffers (i.e. when the total number of buffers becomes 36), DBMIN performs not too much worse than the others. In this case, the difference in throughput is due to the effective allocations to random references by the MG-x-12 algorithms. If the graph extends to higher numbers of total buffers, we expect that a similar pattern of divergence in throughput appears before every multiple of 18, though the magnitude will probably decrease.

6.4 Effect of Data Sharing

In the simulations carried out so far, every query can only access data in its own buffers. To examine the effect of data sharing on the relative performance of MG-x-y and DBMIN, we also run simulations with varying degrees of data sharing. Figure 6 shows the relative throughput of MG-50-y's, MG-100-12 and DBMIN running the second mix of queries with 35 buffers, when each query has read access to the buffers of all the other queries, i.e. full data sharing.

Compared with Figure 4 for the case of no data sharing, Figure 6 indicates that data sharing favors the MG-x-y algorithms[6]. This is not surprising because sub-optimal allocations give higher buffer utilization which in turn causes higher throughput, as data can be shared.

In sum, our simulation results indicate that MG-x-y shows considerable improvement over DBMIN. With the first mix of queries, MG-x-y is able to take advantage of idle buffers and allocate extra buffers to random queries. With the second mix of queries, MG-x-y also gives higher throughput, higher buffer utilization

[5] The graphs for other multiprogramming levels show similar shapes[10].

[6] For other query mixes we have used, the same phenomenon occurs [10].

and lower waiting time for queries than DBMIN. These results demonstrate the effectiveness of sub-optimal allocations to looping references. Finally, if data sharing is allowed among queries, the improvement in performance using MG-x-y is even bigger.

7 Conclusions

In summary, the principal contributions reported in this paper are as follows. First, we have proposed a unified approach for buffer allocation in which both the access patterns of queries and the availability of buffers at runtime are taken into consideration. This is achieved through the notion of marginal gains which give an effective quantification on how buffers can be used efficiently. Second, we have designed a class MG-x-y of allocation algorithms based on marginal gains. Generalizing DBMIN, these algorithms maximize the use of buffers. Simulation results show that our approach is promising, and our algorithms MG-x-y give higher throughput, higher buffer utilization and lower waiting time for queries than DBMIN. These results confirm our beliefs that idle buffers should be minimized, and that sub-optimal allocations, when handled properly, can improve the overall performance of the system.

In ongoing research, we are investigating extensions to MG-x-y, where the runtime conditions considered in buffer allocation include the characteristics of the queries already activated into the system[6]. We are also studying methods of finding the optimal x and y values of MG-x-y. Furthermore, we are considering derivation of formulas for the marginal gain values of more complex queries (such as sort-merge joins, etc.), and formal analysis for data sharing.

Acknowledgements. We would like to thank H. Chou and D. DeWitt for allowing us to use their simulation program for DBMIN so that direct comparison can be made. We would also like to thank Louiqa Raschid for constructive comments on the manuscript.

References

[1] A.F. Cardenas (1975) *Analysis and Performance of Inverted Data Base Structures*, CACM, 18, 5, pps 253–263.

[2] H. Chou and D. DeWitt. (1985) *An Evaluation of Buffer Management Strategies for Relational Database Systems*, Proc. 11th VLDB, pps 127–141.

[3] S. Christodoulakis (1984) *Implication of Certain Assumptions in Data Base Performance Evaluation*, TODS, June 1984.

[4] D. Cornell and P. Yu. (1989) *Integration of Buffer Management and Query Optimization in Relational Database Environment*, Proc. 15th VLDB, pps 247–255.

[5] W. Effelsberg and T. Haerder. (1984) *Principles of Database Buffer Management*, TODS, 9, 4, pps 560–595.

[6] C. Faloutsos, R. Ng and T. Sellis. (1990) *Predictive Load Control for Flexible Buffer Allocation*, Technical Report CS-TR-2622, University of Maryland, College Park.

[7] J. Kaplan. (1980) *Buffer Management Policies in a Database Environment*, Master Thesis, University of California, Berkeley.

[8] E. Kauder. (1965) *History of Marginal Utility Theory*, Princeton University Press.

[9] T. Lang, C. Wood and E. Fernandez. (1977) *Database Buffer Paging in Virtual Storage Systems*, TODS, 2, 4.

[10] R. Ng, C. Faloutsos and T. Sellis. (1990) *Flexible Buffer Allocation based on Marginal Gains*, Technical Report CS-TR-2624, University of Maryland, College Park.

[11] G. Sacca and M. Schkolnick. (1982) *A Mechanism for Managing the Buffer Pool in a Relational Database System using the Hot Set Model*, Proc. 8th VLDB, pps 257–262.

[12] S. Sherman and R. Brice. (1976) *Performance of a Database Manager in a Virtual Memory System*, TODS, 1, 4.

[13] M. Stonebraker, E. Wong and P. Kerps. (1976) *The Design and Implementation of INGRES*, TODS, 1, 3, pps 189–222.

[14] S. Yao. (1977) *Approximating Block Accesses in Database Organizations*, Communications of ACM, 20, 4, pps 260–261.