A MECHANISM FOR MANAGING THE BUFFER POOL IN A RELATIONAL DATABASE SYSTEM USING THE HOT SET MODEL

Giovanni Maria Sacco and Mario Schkolnick

IBM Research Laboratory
San Jose, California 95193

ABSTRACT

The design of the buffer manager in a Relational Database Management System can significantly affect the overall performance of the system. Thrashing is a common phenomenon that occurs in these systems due to the combination of a regular pattern of accesses made by a process and the competing requests for buffer resources made by concurrently executing processes. In this paper, we present a buffer management algorithm based on a model of database requests. A discussion of problems encountered by traditional methods for buffer management as well as extensions to the algorithm are also presented.

1. TRADITIONAL METHODS FOR BUFFER MANAGEMENT

Database Systems typically use an LRU replacement technique [LANG77] to manage their internal buffer(s). The LRU technique has proved to be more efficient than other methods in reducing the amount of paging that occurs in these systems. Also, the technique is relatively simple to implement, something that is highly desirable in a high performance system. However, as the following examples show, there are many cases where serious problems occur.

1. If one of the processes is running a batch like job, i.e., one that performs a long sequential scan on the data while at the same time requesting pages very rapidly, the pages referenced by it will tend to go to the top of the stack, causing the pages used by other processes to be flushed out of the buffer. This causes the batch like process to take precedence over the other processes. If these are short, fast transactions, the situation becomes intolerable.
2. Another case where LRU behaves very badly is when a process cycles through a set of pages, which is larger than the set of pages that can fit in the buffer. In this case, every new reference to a page causes a fault. This effect is called **internal thrashing**.
3. A more serious problem that occurs in multiuser systems is that one process may use pages more rapidly than another one that has a looping behavior as it references pages. In this case, even if the set of pages that are rereferenced inside the loop may be smaller than the buffer size, the "stealing" of pages by the first process effectively reduces the available frames in the buffer for the second process. This reduction can be such that the second process generates a page fault on every request. In this case we say that there is **external thrashing**.

Mechanisms for dealing with thrashing have been developed in Operating Systems. The most well known of these uses the principle of the Working Set Model [DENN68]. A buffer manager using this model defines a window τ and observes the average number of different pages that are requested by a process in a period equal to τ. This number is called the working set size of the process, σ. A scheduler then ensures that while this process is running, at least σ pages are allocated to it. Alternatively, the buffer manager can give different priorities to page requests from the various processes, depending on their working set size. Processes with a large working set size will get more buffer frames allocated to them. In high performance Database Management Systems, this last mechanism has several drawbacks. In the case of a process, like the batch process described in case 1 above or the one described in case 3, the Working Set Model tries to give to this process many buffer frames, causing external thrashing of other processes. If a process loops over a number of pages larger than the window size τ, as in case 2, the working set mechanism will attempt to reserve τ frames for this process in the buffer, where having just one frame associated to it would have caused the same level of faults to occur. Thus the buffer frames are poorly utilized. Finally, in the case of a process with a looping behavior, as it goes from one loop of pages to the next, the working set size will temporarily increase causing this process to get more frames assigned to it. As before, this causes external thrashing. All of the above cases may also cause scheduling problems if processes are scheduled considering their working set size requirements. Moreover, the working set model is expensive to implement, in terms of instructions executed, a fact that has discouraged its use in high performance database systems.

2. THE HOT SET MODEL

Relational Database Management Systems have high level language interfaces allowing their users to state their processing requirements without specifying how the required data should be accessed. The system has internal mechanisms that decide on the best strategy to access the

data. We call these strategies **access plans**. In System R[1], the internal mechanism is called the Optimizer [SELI79]. Since the access plan is generated by the system, it turns out that the pattern of data pages that are referenced can be predicted at the time the access plan is generated. It is not hard to see that for all plans, the pattern of accesses involves looping through sets of pages.[2] This fact has led to the introduction of the Hot Set Model [SACC82] to study this behavior. A set of pages over which there is a looping behavior is called a **hot set**. If the access plan is executed within a buffer whose size exceeds the size of the hot sets, processing will be fast, as the pages that are rereferenced in a loop will stay in the buffer. On the other hand, if the buffer size is below the value required to contain a hot set, the number of page faults required grows rapidly.

This effect can be observed in Figures 1-3 where the total number of page references made while processing different queries is shown as a function of the buffer size. As can be seen from these figures, there may be more than one looping pattern, causing several discontinuities to occur in this curve. Each one of these discontinuities is called a **hot point**. Also, there is a minimum buffer size under which the query cannot run. This is a particular characteristic of System R. When processing a request, System R forces some pages to remain in the buffer pool, independently of what the LRU algorithm would call for [LORI77]. (A page that is forced to remain in the buffer pool is called **fixed**. Because pages are fixed, System R does not implement a true LRU mechanism for managing its buffer pool.[3]) This last discontinuity is also referred to as a hot point. The largest hot point smaller than the available buffer size is called the **hot set size**. for the query. In particular, the hot set size could correspond to the number of pages a process could keep fixed at the same time. Examples of hot set sizes for various queries are shown in Figures 1-3, using a buffer size of 20 pages. These figures show that running a query without a number of pages in the buffer equal to the hot set size is very expensive. Moreover, when the processing of the request is done under these conditions, the behavior of the Optimizer as presented in [SELI79] is questionable. In fact, if processing is done with a buffer size smaller than the hot set size determined for the access plan, another access plan, with a smaller hot set size might be the best plan to take. Thus, in this situation, the optimizer incorrectly generates the first access plan.

An example of this situation is shown in Figure 4. For a buffer size larger than 10, the cheapest path is to use EMP as an outer relation, at a cost of 5,000 cost units. However if that many frames are not available in the buffer pool at the time the query is processed, then the cost of using this plan goes up to 44,000 cost units. In this case , accessing DEPT as outer would have been preferable, at a cost of 31,000 units of cost. (If there are more than 20 buffer frames at the time the query is processed, the cost for this plan goes down to 13,000 cost units.

The above discussion suggests that the hot set size should be determined, and used, when deciding on the best access plan [SACC82]. Examples of hot set size formulas for some access plans that are examined by the System R optimizer are given at the end. Note that the formulas predict the average size of a hot set. This effect is also considered in the schema presented in the next section.

3. A BUFFER MANAGEMENT SCHEMA BASED ON THE HOT SET MODEL

The basic idea for the buffer management schema is to try to ensure that requests are run with an "effective" buffer of size equal to their hot set size. To do this, we maintain separate LRU chains, one for each process. Each chain has a maximum size equal to the hot set size for the request being serviced by that process. The buffer frames not associated with any process are collected into a separate LRU list, called the **free list**. Each LRU list has 2 numbers associated to it: **numref**, the number of frames that are requested by the process, and **numall**, the actual number of buffer frames in the LRU list. Numref is equal to the hot set size of the process. Numall can be different from numref if at the time the process requests an LRU stack of size σ, the size of the free list is less than σ. In this case, the entire contents of the free list are assigned to the LRU stack of the requesting process (see step 2b below.) When this happens, we say that the list is **deficient**.

When a process requests a page, the content of the entire buffer is examined to determine if the page is there.[4]

If the page is found, then if the page is also in the local LRU stack of the process, the local stack is updated. When a page is found, a process may fix the page to ensure it will not be flushed out of the buffer pool while the process is actively working on it. Each page has a reference counter for the purpose of keeping track of the number of times it has been fixed. When a process fixes a page, its reference counter is incremented by one.

If the page is not found, then the least referenced page in the local stack, whose reference count is zero, is flushed out of the buffer pool and replaced by the requested page[5]

[1] In this paper, we use System R [ASTR76,BLAS79] as a model of a relational database management system. The ideas presented here carry over other relational DBMS.

[2] Note that even for the case of a relation scan, there is a loop between a control page and a data page.

[3] Fixing pages is done for performance reasons. We expect that any high performance relational DBMS will have this feature. The buffer management schema presented below assumes that this is done by introducing a special case in step 3b2. If no pages are fixed, steps 4 and 5 of the schema do not exist and step 3b2 can be simplified.

[4] This can be done efficiently using a hashing technique. This is a well known method for a search a buffer pool.

[5] For simplicity, we do not examine the actions to be taken if a page in the buffer has been modified by a process. Usually, these pages are less preferable as candidates to be flushed out since their replacement involves a write operation to secondary storage.

When a process unfixes a page, the reference count of the page is decreased by one.

When a process terminates, the pages of its stack are allocated to processes whose LRU stack are deficient. This allocation is done by finding a deficient LRU stack and giving it as many free frames as required to complete the number of pages in its hot set size σ. If there are still free pages left, another deficient process is found and frames allocated to it as before. If there are no more deficient processes and there are still free frames left, they go to the free list.

A step by step description of the buffer management schema follows:

1. Initialize: Assign to the free list all the buffer frames.
2. New process arrives: Allocate empty LRU list. Set numref=σ and numall=0. Get as many as σ pages from the free list. Set numall to be the number of pages that were actually obtained (if the free list size is less than σ then numall will be be less than σ and the free list becomes empty.)
3. Process requests a page: Search for a page in the buffer pool.
   a. Page found. If page is in local LRU stack, update local stack; else do nothing.
   b. Page not found. A page fault occurs. Have to determine a page to be flushed out of the buffer pool. There are two cases to consider:
      1. If the local stack contains unfixed pages then flush out the least recently used (i.e., use the modified LRU algorithm.)
      2. If the local stack contains only fixed pages then get a frame from the free list, if it is not empty (and flush the corresponding page). If the free list is empty, get a frame from another LRU stack containing an unfixed page. Stacks that are deficient are to be preferred. Increase numall by 1. Note that numall may become larger than numref in this case.
4. Process fixes a page: Increase reference count by one.
5. Process unfixes a page: Decrease reference count by one.
6. Process releases a page: Do nothing.
7. Process terminates: Reallocate the LRU stack among deficient processes, trying to satisfy completely a process before satisfying another one. If no deficient process exists, return frames to the free list.

4. COMMENTS ON BUFFER MANAGEMENT ALGORITHM

The above schema presents the basic components of a buffer management schema based on the idea of the hot set size. We now discuss the behavior of the algorithm under different situations:

1. The various hot sets scanned during the evaluation of the request need not be all of the same size. The optimizer only computes an average number for the hot set size, based on available statistics such as the sizes of the relations involved, the cardinalities of different columns, the filter factors of predicates, etc. Thus during execution, a process may loop over a set of pages larger than this hot set size. Assuming that numref is larger than the maximum number of pages that the process simultaneously maintains fixed, then step 3b2 will not be executed on its behalf. Thus, its LRU stack will never be increased and internal thrashing will occur as the process steals pages from step 3b2 makes sure that the process gets another page. The issue of what happens when all the pages in the buffer are fixed and a request for another one is itself. Note however, that there will not be any external thrashing. To accommodate for this situation, the optimizer could request a hot set size greater than the average it computed. However, this leads to underutilization of the buffer pool. Another possibility is to dynamically increase the allocation of frames to this process if the free list is not empty. These frames would be released if another process requested frames from the free list and found it empty (i.e., consider the extra frames obtained by the first process to be an extension of the free list.) Although this change could be incorporated into the buffer manager, the solution presented in the previous section appears to be a good balance between simplicity and performance.
2. If the hot set size for a process is equal to the average number of fixed pages that it requires, then a situation can occur where more pages than the allocated number have to be simultaneously fixed. In this case, made is outside the scope of this paper. Solutions that have been adopted in this case include bringing down the system or dynamically extending the buffer.
3. To make sure that the number of deficient processes is kept to a minimum, one can schedule the requests so that the sum total of the hot set sizes of the processes that are being serviced is less than or equal to the size of the buffer pool. The question of scheduling is not considered here. However we note that this kind of scheduling may lead to deadlock problems that are hard to manage [OBER80]. To avoid the deadlock problem one can restrict the set of concurrent processes (i.e., reduce the multiprogramming level). Although this does not eliminate the possibility of processes running with stacks that are deficient it does eliminate the issue of deadlock related to buffer requests and considerably reduces the occurrence of internal thrashing. Moreover, knowledge of the hot set sizes of the request issued by processes can be used to decide on an appropriate multiprogramming level. Thus, the restriction on the multiprogramming level appears to be a suitable complement to the algorithm presented here.
4. Note that in the schema presented above, a page in the buffer appears in only one LRU chain. Thus, if a page is shared by two processes, the first one requesting it will put it on its own LRU chain. Although this may seem arbitrary, there is no problem in doing so. If the shared page remains active (e.g., a control page in System R) it will remain high in the LRU chain (i.e., it is unlikely that the page will be flushed out.) On the other hand, if the cross reference to the page is a sporadic event, the page will migrate from one LRU stack to the other. This is not important since these occurrences are infrequent.

minimum hot point (i.e., the minimum number of pages required to run). The schema presented here guarantees that, if processing of this query is allowed, it will not suffer external thrashing. (Note that for these requests there can be no internal thrashing.) In a high performance system, in order to insure that a certain level of fast queries is active at the same time, the buffer can be divided into two regions; one for fast requests, the other for slow ones. (The initialization step would create two free lists each having the size of the respective region. A reasonable choice is to have both regions of the same size.) The fast requests are thus guaranteed a certain minimum number of frames for them. In order to maintain good utilization of the buffer pool, the free list for the fast queries has to be allowed to take frames from the free list for the slow queries (this will happen when there are no slow queries present in the system.) As before, adequate service can be maintained by restricting the multiprogramming level.

6. The last comment has to do with contention problems in the buffer manager. Clearly, the stack manipulations that are performed on the free list in steps 2 and 3 of the algorithm must be serialized. Since the operations that are done are removal and insertions of elements in LRU chains, the path length through the critical region is no different than that through the code of a standard buffer manager with one LRU stack. Thus this algorithm should not introduce additional serialization problems. In fact, it may even decrease them as the manipulations of private LRU chains need not be serialized.

5. EXAMPLES OF HOT SET SIZE COMPUTATION

As an illustration of how the hot set size can be estimated by the optimizer of a relational database management system, we show some examples using System R. For each two way join between relations R1 and R2, assume R1 is the outer relation and R2 is the inner. Control pages are not included in the expressions below. The following terms are used below

P(R1)	Number of pages for relation R1.
P(R2)	Number of pages for relation R2.
dindex (R2)	Depth of the index on R2.
ls(R2)	Number of pages in the inner loop for R2. It is given by P(R2) divided by the number of different values for the attribute upon which the join is performed. This estimate is based on the uniform distribution assumption.
I(R2)	Number of pages in the index used to access R2.
lsleaf(R2)	Number of index leaf pages scanned on an inner loop of R2.

for a sequential scan on both R1 and R2,

$$\text{hot point} = 1 + P(R2)$$

for an index scan on R1, sequential scan on R2,

$$\text{hot point} = 2 + P(R2)$$

for a sequential scan on R1, index scan on R2 (smooth discontinuity), interpolate between

$$1 + \text{dindex}(R2) + \text{ls}(R2)$$

and

$$1 + I(R2) + P(R2)$$

Example 1.

Type 2 join:

for a sequential scan on both R1 and R2,

$$\text{hot point} = 1 + \text{ls}(R2)$$

for a sequential scan on R1, index scan on R2,

$$\text{hot point} = 1 + \text{lsleaf}(R2) + \text{ls}(R2)$$

where lsleaf(R2) is the number of leaf pages in the inner lo
For an index scan on R1, sequential scan on R2,

$$\text{hot point} = 1 + \text{dindex}(R1) + \text{ls}(R2)$$

Example 2.

In Example 1, the first formula is derived by reserving enough frames to contain the entire R2 relation, plus one frame for a data page for R1. If a frame for a data page of R1 is not reserved the access to R1 causes the first page in the R2 loop to be replaced, and consequently the entire set of pages in the R2 loop to be lost. For the second formula, an additional frame is reserved for the leaf pages of the index, which is always accessed before accessing R1 data pages. The third formula presents a smooth discontinuity. The minimum number of faults is achieved when all the access entities for R2 (index and data pages) completely fit in the buffer. The number of faults will increase in a roughly linear way, until only a number of frames sufficient to hold an average loop on the second relation, is available. This assumes substantial rereferences between succesive inner loops. If the number of data pages in the referenced relation is large, and the join filtering is high, data page rereferencing will be very low. In this case, P(R2) is substituted for ls(R2). The formulae in Example 2 are derived using analogous considerations. Values for the estimated number of hot set size obtained using these expressions are shown in Figures 1-3.

The above formulae are easily generalizable to n-way joins, and represent a conservative estimate of the hot points. The number of hot points to be computed varies from n-1 to 3(n-1) (in the case of smooth discontinuities), where n is the number of relations referenced in the query.

This is a worst-case figure. In general, only the maximum hot point needs to be computed.

6. REFERENCES

[ASTR76] Astrahan, M. M., et al. "System R: A Relational Approach to Database Management," ACM Trans. on Database Systems, 1, 2, June 1976 (97-137).

[BLAS79] Blasgen, M. W., et al. "System R: An Architectural Update," IBM Research Report: RJ2581, July 1979.

[DENN68] Denning, P. J. "The Working Set Model for Program Behavior," Comm. of the ACM, 11, 5, May 1968 (323-333).

[OBER80] Obermark, R. "Global Deadlock Detection Algorithm," IBM Research Report RJ2845, June 1980.

[SACC82] Sacco, G.M. and Schkolnick, M. "Buffer Management in Relational Database Management Systems," in preparation.

[SELI79] Selinger, P. G., et al. "Access Path Selection in a Relational Database Management System". Proc. of the 1979 SIGMOD Conference.

[LANG77] Lang, T., et al. "Database Buffer Paging in Virtual Storage Systems," ACM Trans. on Database Systems, 2, 4, Dec. 1977 (339-351)

[LORI77] Lorie, R. "Physical Integrity in a Large Segmented Database," ACM Trans. on Database Systems, 2, 1, Mar. 1977 (91-104)

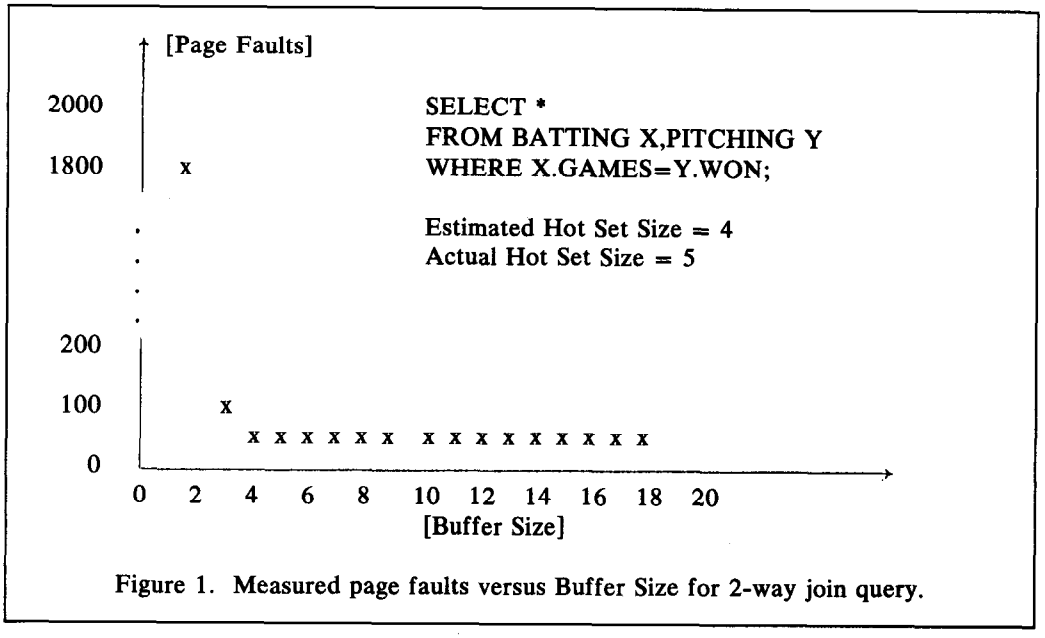

Figure 1. Measured page faults versus Buffer Size for 2-way join query.

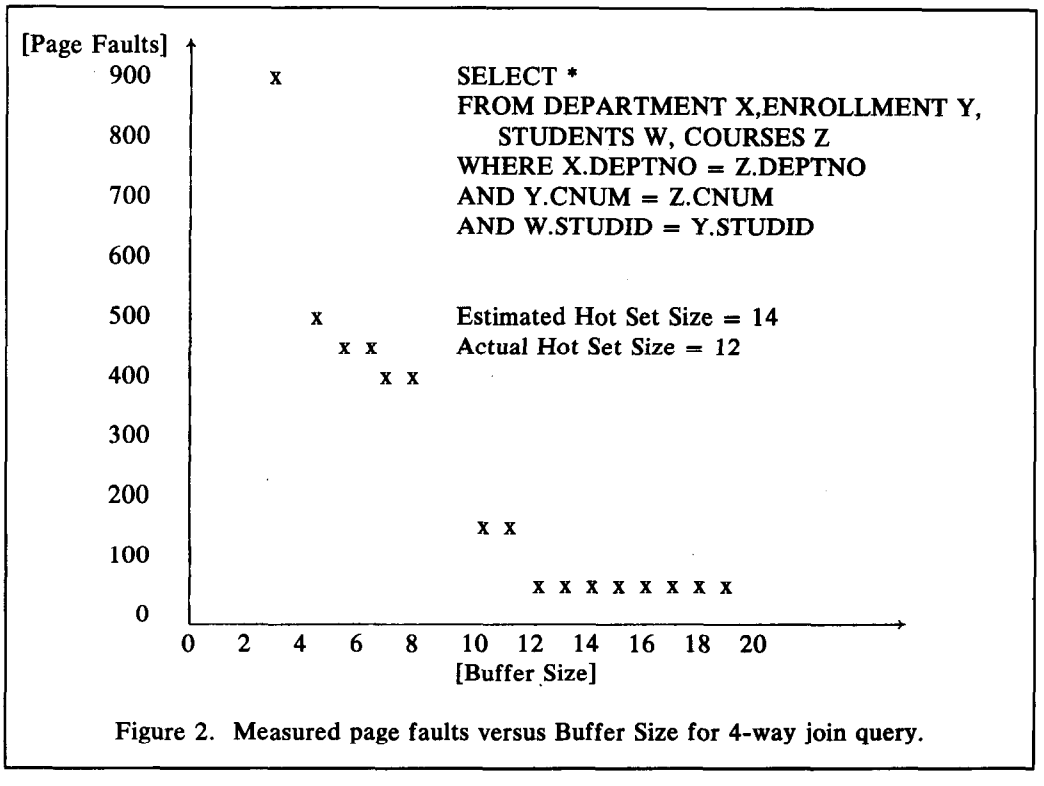

Figure 2. Measured page faults versus Buffer Size for 4-way join query.

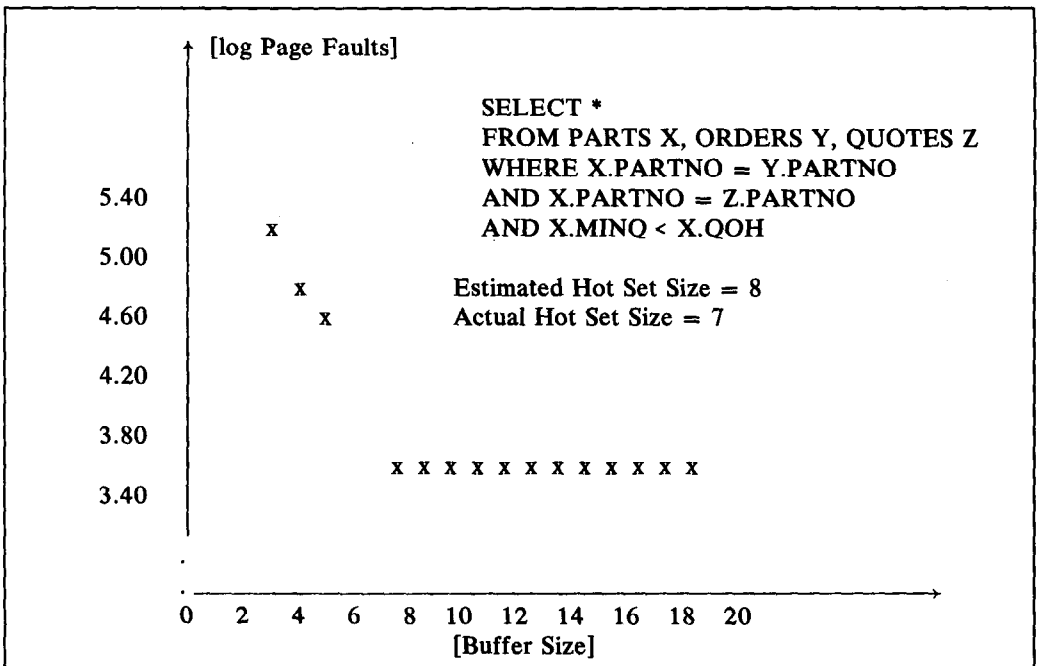

Figure 3. Log (base 10) of measured page faults versus Buffer Size for 3-way join query.

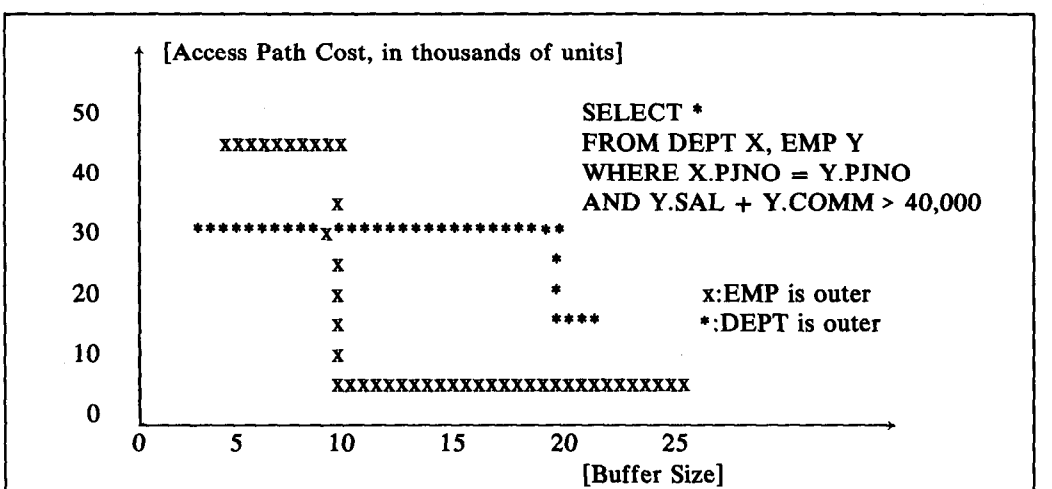

Figure 4. Computed access path cost for two ways of performing 2-way join: in both cases, an index scan on PJNO is used for both tables