Joint Conference of 1996 International Computer Symposium
December 19~21, Kaohsiung, Taiwan, R.O.C.

Buffer Management for Relational Database Systems

Don Haderle, Jim Teng
555 Bailey Avenue
IBM Santa Teresa Laboratory
San Jose, CA 95141, USA

Jen-Yao Chung, Tai-Yi Huang[*]
P.O. Box 704
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598, USA

Abstract

Good buffer management is very important to the overall performance of a database management system. The buffer manager is the database component which manages data buffering. This paper describes the evolution of the buffer manager's design in managing the allocation of database buffers to maximize performance. We start with a discussion of the memory hierarchy. We next describe the improvement of the search scheme and the buffer replacement algorithm used by the buffer manager. Finally, we discuss the evolution of asynchronous prefetch and deferred write, two important features that significantly enhance I/O throughput.

1 Introduction

Commercial relational database systems are designed to support large databases with applications that might require a large number of I/O operations. To maximize database performance, it is desirable to minimize physical I/O activity and the resultant delay to applications whenever possible. The buffer manager is a component within the database which is designed for this purpose.

The buffer manager controls data buffering between virtual memory and DASD (direct access storage device). Data buffering provides a method to handle the mismatch between high speed CPU and slow DASD. The efficient movement of data between virtual memory and DASD, is implemented through a virtual storage space, called a *buffer pool*, consisting of many buffers of equal size matching the stored geometry of the data. A buffer pool contains data shared by all applications using database systems. The buffer manager manages the allocation of buffers in a buffer pool to database data as needed and supplies the virtual copies of database data as requested. Whenever a page is requested, the buffer manager first searches the buffer pool hoping to find it. If not found in the buffer pool, a physical I/O is issued to read the page from DASD into the buffer pool. The buffer manager reduces the number of DASD read I/Os by caching data which has a high probability of being accessed again, within the constraints of the size of buffer pool. In addition, it reduces the latency of DASD write I/O by deferring the externalization of committed changes and conducting the write I/O in a batch when possible. Thus, the overall performance of the database system is significantly dependent on the effective implementation of the buffer manager.

IBM Database 2^{TM} ($DB2^{TM}$) is IBM's relational database system for large mainframes. This paper describes the evolution of DB2's buffer manager design considerations to maximize DB2 performance. Section 2 describes how the buffer manager maintains the two-level storage hierarchy in a buffer pool, reflecting the main versus expanded memory characteristic. Section 3 describes the hashing scheme used by the buffer manager to determine whether a requested page is in a buffer pool. This section also describes how the buffer manager handles the dynamic expansion and contraction of a buffer pool and its hash table. Section 4 discusses the buffer replacement algorithm. DB2 uses the Least Recently Used (LRU) algorithm as its primary buffer replacement algorithm. This section also describes the MRU (Most Recently Used) scheme used by the buffer manager for reference-once patterns. Section 5 discusses the asynchronous prefetch feature and Section 6 discusses the deferred write feature. Finally, Section 7 concludes this paper.

2 Hiperpools

IBM mainframe system provides two levels of memory hierarchy: main memory and expanded memory. Main memory is designed to be "byte-addressable"; this means that processor instructions can operate on any sequence of bytes as one might expect. Expanded memory is "page addressable" only; this means that processors are limited to a small set of instructions which operate on the entire page and do not cross page boundaries. One can think of expanded memory as a place to cache data pages and provide a mechanism to move these pages to main memory when needed by processors, since the time to move the data is very short compared to an I/O. Based on the two level memory architecture, it provides the notion of a hiperspace, which is an address space that is mapped directly into expanded memory. DB2 uses two levels of storage hierarchy for each buffer pool to reflect the characteristics of the memories. The first level is the virtual buffer pool. The operating system determines whether the data is held in main memory, expanded memory, or pageed out to the paging devices (i.e. DASDs), and ensures that the page is in main memory when DB2 references it. Data should be paged out only in extreme memory starvation, which

[*]now with Microsoft Corporation, One Microsoft Way, Redmond, WA 98052-6399

```
1   if (p is found in the virtual buffer pool)
2      return the found buffer;
3   else {
4      steal a buffer s from the virtual buffer pool;
5      if (s should be stored) {
6         steal a buffer t from the hiperpool;
7         copy the content of s to t;
8         place t back to the hiperpool;
9      }
10     if (p is found in the hiperpool) {
11        copy the content of the found buffer
            u to s;
12        place u on the free buffer set in the
            hiperpool;
13     }
14     else
15        read p from DASD to s;
16     place s back to the virtual buffer pool;
17     return the buffer s;
18  }
```

Figure 1: The handling of the buffer pool during a page search

happens rarely. The second level is called the hiperpool. For each buffer pool, the hiperpool is optional. If defined, it will be allocated from hiperspaces to directly support the continued growth of the buffer pool into expanded storage. For ease of reference, the acronym "buffer pool" will be used to reference a pair consisting of a virtual buffer pool and a hiperpool if no distinction needs to be made between them.

A hiperpool, if defined, will be exclusively used by the buffer manager to back up moderately referenced non-dirty database data. With this large data cache, the number of DASD read I/O operations can be expected to be reduced. In addition, pages cached in hiperspace can be retrieved in microseconds as opposed to milliseconds from disk. Hiperpools can also give control to system programmers to limit the amount of main memory used by DB2 buffer pools. This control improves the overall system performance by minimizing main memory contention among all applications running on the same system.

Figure 1 shows the procedure performed by buffer manager at the request of a page p, we assume that the hiperpool is defined. The buffer manager first searches the virtual buffer pool. If p is found, a buffer pool hit returns. Otherwise, the buffer manager steals a buffer from the virtual buffer pool. If the stolen buffer s, may be referenced again, buffer manager caches the content of s to the hiperpool. Lines 6 to 8 in Figure 1 explain the details of the caching. The procedure that performs the tasks in line 4, 6, 8, 12, and 16, a buffer replacement algorithm, will be described later in Section 4.

Now the buffer s is ready for the requested page. The buffer manager continues to search for p in the hiperpool. If p is not found, the buffer manager initializes a read I/O to read the page from DASD to s. Otherwise, the found buffer u, is copied to s. To avoid double buffering for pages in the buffer pool, u is then invalidated immediately and placed on the free buffer set in the hiperpool. Finally, s is placed back to the virtual buffer pool and a buffer pool hit returns. The procedure needed in lines 1 and 10, a buffer search algorithm, will be described in Section 3.2.

3 Buffer Pools Management

There are multiple buffer pools in the database system. To best use available virtual storage, the buffer manager dynamically constructs a virtual buffer pool during the process of accessing the first table that requires a particular virtual buffer pool. Conversely, a virtual buffer pool is deleted when the buffer manager determines that there is no longer any usage of the virtual buffer pool. The buffer manager handles the creation and deletion of a hiperpool in a similar way.

3.1 Dynamic buffer pool size

To avoid the problems arising from fixed-size buffer pools, DB2 provides an online facility to set or alter the size of a buffer pool. In a database system that supports only fixed-size buffer pools, a buffer pool size has to be determined before the system is started and cannot be altered without restarting the database system. The fixed-size limitation may abnormally end a transaction due to unavailable buffer resources.

In early DB2 implementation, installations could specify the minimum size and the maximum size for each buffer pool. The two size attributes allowed buffer pools to be dynamically expanded and contracted. The buffer manager used the minimum size to create a buffer pool. At the request of a free buffer, the buffer pool was expanded if all buffers were held and the pool's maximum size had not yet been reached. In other words, the pool expansion was driven on demand and was temporary to the execution unit. Once the execution unit was committed/aborted, the buffer pool was contracted back to its original size. Because the expansion and contraction functions involved physically acquiring and releasing storage from the underlying operation system, excessive use of these functions caused thrashing, drastically degrading DB2 performance. Consequently, DB2 required system administrators to adjust buffer pool minimum size to maximize DB2 performance.

A time delay was proposed on contraction to reduce the thrashing. However, administrators preferred more direct control, resulting in DB2 dropping support of dynamic buffer pool resizing in favor of an operational command `ALTER BUFFERPOOL` provided in the current DB2 implementation to easily set or alter the size of each active or non-active buffer pool.

3.2 Dynamic hash table size

DB2 uses a hashing scheme on page number and database number to determine whether a requested page is in the buffer pool. The buffer manager maintains a hash table, as shown in Figure 2, for each virtual buffer pool and each hiperpool. A hash table is actually a series of anchor pointers. Each an-

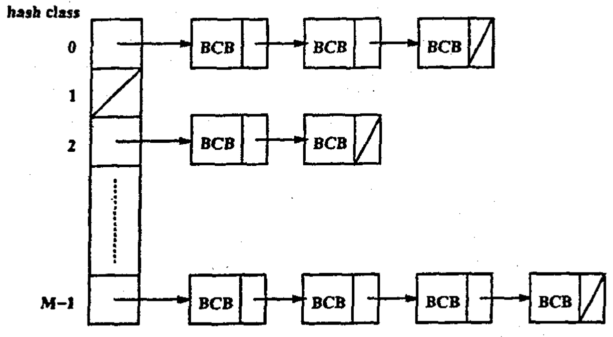

Figure 2: The hash table for buffer search

chor pointer points to a hash class; it is either a null pointer which indicates that there is no buffer in the hash class, or points to a linked list of buffer control blocks. A buffer control block (BCB) contains the address of the associated buffer and other control information. Buffer control blocks are allocated contiguously. By only looking into buffer control blocks, buffer manager minimizes the amount of memory paging in the searching process. In addition, the hashing scheme makes the time needed for a buffer search independent of the buffer pool size.

The ratio of a buffer pool size to its hash table size determines the average number of buffer control blocks in a hash class. The average number of comparisons during a buffer search depends on this ratio also. DB2 current implementation sets the number of hash classes to the larger of 64 or 20% of the buffer pool size. The reason for choosing 20% is based on the fact that the probability of having more than 7 buffer control blocks on a single hash synonym chain is less than 5%. Furthermore, to facilitate concurrent buffer searches for the symmetric multi-processors (SMP) processors, DB2 sets the number of latches that control hash search to one eighth of hash classes. For example, if a buffer pool has 1000 buffers, 200 hashing anchor points and 25 latches will be allocated. The ratios are empirically determined and will change over the years based on technology shift.

As said earlier, a buffer pool can be easily expanded or contracted via the ALTER BUFFERPOOL command. If the new pool size after expansion (contraction) is significantly larger (smaller) than the current size, the hash table size needs to be adjusted to speed up buffer search and/or utilize virtual storage efficiently. Thus, instead of keeping a static hash table size, DB2 dynamically builds a new hash table based on the new buffer pool size.

DB2 has made the process of rebuilding a hash table as transparent to applications as possible. The primary task in the rebuilding process is the migration of entries from the old hash table to the new one. Other systems do this by blocking access to the buffer pool or by completely stopping and restarting the database system. Instead, DB2 allows applications to access the buffer pool through the old hash table and latches during the rebuilding process. The actions taken by the buffer manager to rebuild a new hash table is listed below

1. Allocate a new hash/latch table based on the new pool size.
2. Set flag to indicate rebuilding hash table is in progress.
3. For each hash class of the old hash table, if the class is valid and non-empty, all pages are rehashed and moved to the new hash table. The class is then marked as invalid.
4. Once all hash anchors are being marked invalid, mark the new hash/latch table as the primary hash/latch table.
5. Set flag to indicate rebuilding hash table is done.
6. Perform garbage collection on the old hash table and latches when there are no remaining user references.

When rebuilding hash table is in progress, a page search will be carried out through the old hash class latch. If the old hash anchor is invalid, buffer manager continues to search the new hash table without acquiring latch on the new hash class. Furthermore, buffer manager always enqueues new pages to the new hash table during this period. In order to ensure that the old set of latches can still be used to serialize the manipulation of a hash class for both new and old hash tables during an expansion, DB2 requires that the new hash table size is an integral multiple of the old hash table size. Latches are referenced by their ordinal number and can be resolved to the old or new hash table by keeping appropriate state.

4 Buffer Replacement Algorithm

DB2 uses the Least Recently Used (LRU) algorithm as its buffer replacement algorithm. To increase the buffer hit ratio of frequently referenced or updated pages, all referenced buffers are placed in a chain manipulated by the LRU algorithm. If a requested page is found in the chain, the page is moved to the bottom of the chain, the most recently used page. Otherwise, a buffer is either released from the free buffer queue or the least recently used buffer, the one on the top of the chain, is dequeued if no free buffer is available. After being assigning to the page, the buffer is placed back on the bottom of the chain. This mechanism, however, has several deficiencies when both dirty pages (i.e. updated pages) and clean pages exist in the chain:

- During a physical write I/O, buffer manager has to search through the chain to find dirty pages (i.e. updated pages that have not yet been written back to disk).
- The implementation to deal with the case when a free buffer is requested when the least recently used page is dirty is sophisticated and inefficient.

4.1 Two-chain approach

To simplify the implementation but also keep a high buffer hit ratio for frequently referenced or updated pages, DB2 has implemented an approach that uses two chains, LRU-chain and Changed Page Chain (CPC). In this implementation, a clean page is stored in the LRU-chain and a dirty page is stored in both the LRU-chain and the CPC. Generally, the CPC is a subset of the LRU-chain. Both chains are manipulated by the LRU algorithm.

When a free buffer is requested and no free buffer is available, buffer manager always steals a buffer from the LRU-chain. The steal operation can be described as below:

1. If the buffer on the top of the LRU-chain is dirty, dequeue it from the LRU-chain. Continue with the next buffer until a clean buffer is found.
2. Dequeue the clean buffer on the top of the LRU-chain. Return this buffer.

The reason to dequeue a dirty buffer from the top of the LRU-chain is because it has not been referenced for a long period of time. On the other hand, no change is made on the CPC to preserve the order for writing out dirty pages.

Let the requested page be p. The following describes the handling of the LRU-chain and the CPC during a read operation and a write operation. We assume here that there is one LRU-chain and one CPC. However, these operations can be easily extended to a system which maintains multiple LRU-chains and CPCs.

Read operation

case (1) *p is not in the buffer pool:* The buffer manager either gets a buffer from the free buffer queue or, if no free buffer is available, uses the steal operation described above to steal a buffer. It next assigns the buffer to p and places it at the bottom of the LRU-chain.

case (2) *p is in the LRU-chain and is clean:* p is moved to the bottom of the LRU-chain.

case (3) *p is in the LRU-chain and is dirty:* p is moved to the bottom of the LRU-chain. Since p is dirty, it must also appear in CPC. However, the position of p in the CPC remains the same in this situation because if p is not updated for a long time, it is a good candidate for writing out in the next write I/O.

case (4) *p is in the CPC but not in the LRU-chain:* The buffer manager adds p to the bottom of the LRU-chain. The reason that p is not found in the LRU-chain is because it was dequeued in a steal operation. Since p is referenced again, it becomes least stealable and should be put at the bottom of the LRU-chain. Similarly, the CPC is not changed in this situation.

Update operation

case (1) *p is not in the buffer pool:* The buffer manager either gets a buffer from the free buffer set or, no free buffer is available, steals a buffer from the LRU-chain. It next assigns the buffer to p, marks it dirty, and places it at the bottom of the LRU-chain. On the other hand, p is added to the bottom of the CPC.

case (2) *p is in the LRU-chain and is clean:* The buffer manager first adds p to the bottom of the CPC. It next moves p to the bottom of the LRU-chain after marking it dirty.

case (3) *p is in LRU-chain and is dirty:* p is moved to the bottom of the LRU-chain. As we said earlier, p must also appear in the CPC. The buffer manager moves p to the bottom of the CPC.

case (4) *p is in CPC but not in LRU-chain:* p is moved to the bottom of the CPC. In addition, the buffer manager marks p dirty and adds it to the bottom of the LRU-chain.

When a write I/O operation is triggered (discussed in Section 6), a dirty page q on the top of the CPC will be dequeued and its content will be permanently written on DASD. Depending on whether q is in the LRU-chain, the manipulation of the LRU-chain can be discussed in the following two cases:

case (1) *q is in the LRU-chain:* Because write I/O is triggered by the buffer manager based on the usage of the buffer pool, it should not be treated as another reference to q. Therefore, q is marked clean and remains at the same position in the LRU-chain.

case (2) *q is not in the LRU-chain:* As we said earlier, the reason that q is not in the LRU-chain is because it was not referenced/updated for a long period of time and was dequeued in a steal operation. Consequently, when q becomes clean and stealable, it should be available for stealing sooner than any page in the LRU-chain. Therefore, q is added to the top of the LRU-chain.

4.2 MRU scheme

A data access pattern where data referenced are unlikely to be referenced again is called a reference-once pattern. Though the LRU algorithm is good for sequentially referencing behavior, it has an adverse effect to the overall buffer hit ratio when encountering a reference-once pattern. For such a pattern, the LRU algorithm will use the entire buffer pool in a round-robin fashion. This situation will reduce the buffer hit ratio for other concurrent transactions. In addition, if the buffer pool is larger than the virtual storage, i.e. only part of the pool can be stored in main memory and expanded memory at the same time, the round-robin access to the buffer pool will cause excessive system paging activities. As a result, the performance of the database system will be decreased.

To overcome this problem, DB2 provides the Most Recently Used (MRU) scheme for this kind of patterns. Each buffer accessed in a reference-once pattern is specially marked. Because the marked buffer will not be referenced again, the MRU scheme places it on the top of the LRU-chain once it becomes available for stealing. In other words, the buffer will become the most stealable one when it is ready for stealing.

5 Asynchronous Prefetch

Asynchronous prefetch and deferred write are two major performance features that enhance DB2 I/O throughput between DASD and virtual buffer pools. For pages that will be possibly referenced later, the asynchronous prefetch feature enables the buffer manager to read ahead and fill buffers with them while

DB2 is processing other buffers. Three prefetch functions are provided: sequential prefetch, list prefetch, and dynamic prefetch.

5.1 Sequential prefetch

Sequentiality of access is an inherent characteristic of many database systems. For example, the study done by Rodriguez-Rosell [6] on the data referencing behavior using IMS (Information Management System/360) [4] showed that strong sequentiality was found in the database system. The sequential prefetch function in buffer manager is provided for data access which reveals strong sequentiality, such as table scans, cluster index scans, and some DB2 utility programs. On the other hand, when non-sequential referencing behavior is encountered or there is a shortage of buffers, buffer manager will turn off this function and utilize synchronous reading instead. For example, in the current implementation the buffer manager will disable sequential prefetch when the percentage of available buffers falls below 25%.

Sequential prefetch was implemented first and used by DB2 utilities to improve utility performance. Sequential prefetch enables I/O parallelism for utilities. While a utility job is in the middle of processing records within a set of pages, the prefetch engine is working in parallel to read the next set of pages. By overlapping CPU and I/O, it significantly reduces the total elapsed time for I/O intensive type utilities.

The sequential prefetch was extended later and used to process queries that need to access data in sequential order. The intelligent query optimizer in DB2 can take a decision on whether to trigger sequential prefetch when choosing an access path to process a query. Sequential prefetch is enabled on data for table scans. For a clustered index scan, both data and index are enabled. For a well-organized index, sequential prefetch is also enabled on a matched or unmatched index scan.

In the latest implementation, sequential prefetch is implemented as follows. The buffer manager issues a sequential prefetch when a "trigger page" is referenced. Whether a page should trigger sequential prefetch is decided in an application PLAN or PACKAGE at BIND time. If a read engine is available and there are enough buffers in the pool when receiving a sequential prefetch, the buffer manager asynchronously acquires necessary buffers and loads them with the required data. By read engines, we mean the subtasks which perform DASD read I/O concurrently. Two sets of buffers are allocated from the 4K pool and 32K pool during a sequential prefetch. The number of buffers allocated, called the PreFetch Quantity (PFQ), depends on the characteristics of the program issuing sequential prefetch and the number of available buffers in a pool. For example, if the number of available buffers is 240 in the 4K pool and 40 in the 32K pool when any SQL program issues a sequential prefetch, 16 buffers and 2 buffers will be allocated from the 4K pool and 32K pool, respectively. Table 1 lists the value of prefetch quantity (PFQ) in each situation.

4K pool		
No. of buffers (N)	SQL	Utilities
$8 \le N < 224$	8	16
$224 \le N \le 1000$	16	32
$1000 < N$	32	64

32K pool		
No. of buffers (N)	SQL	Utilities
$1 \le N < 12$	1	2
$12 \le N \le 100$	2	4
$100 < N$	4	8

Table 1: The value of PreFetch Quantity (PFQ) in each situation

5.2 List prefetch

The buffer manager provides list prefetch to reduce I/O traffic for applications of random access with unclustered indexes. To achieve this goal, DB2 first stores the index entries of referenced data, instead of reading them immediately. These indexes are later sorted according to their RIDs (Record IDentifiers), which puts them in disk store sequence. If Multiple Index Access Path is selected, the AND(OR) operation on two sets of records is implemented by the intersection (union) function on the two sets of sorted RIDs. The two sets of RIDs are consolidated into a composite list that is later passed to the buffer manager. Finally, buffer manager uses the record identifiers list to asynchronously fetch data from DASD in one batch.

Because all index entries referenced are sorted according to record identifiers, at most one prefetch per data page is needed. In addition, if more than one record identifier points to the same page, the buffer manager only needs to read this page once. In contrast, those blocks of pages that are not referenced are skipped. As a result, if the referenced data pages are grouped closer together, fewer prefetches are needed. Compared with the purely sequential prefetch which may read the same page many times due to page purging, list prefetch improves DB2 performance on applications with unclustered indexes by reducing the number of DASD read I/O operations.

5.3 Dynamic prefetch

Dynamic prefetch is also knows as sequential detection at execution time. This feature activates sequential prefetch if buffer manager detects sequential or near sequential access pattern for data access which is not declared as sequential at BIND time. The buffer manager later deactivates sequential prefetch and returns to synchronous reading when the sequential access pattern stops. Generally, this feature is beneficial because it activates and deactivates sequential prefetch as it sees fit.

Dynamic prefetch can be applied for both index leaf pages and data pages. The detection of sequential access patterns usually happens on the inner table of a nested loop join, where data is accessed sequentially. It is also useful for sequential access not found at BIND

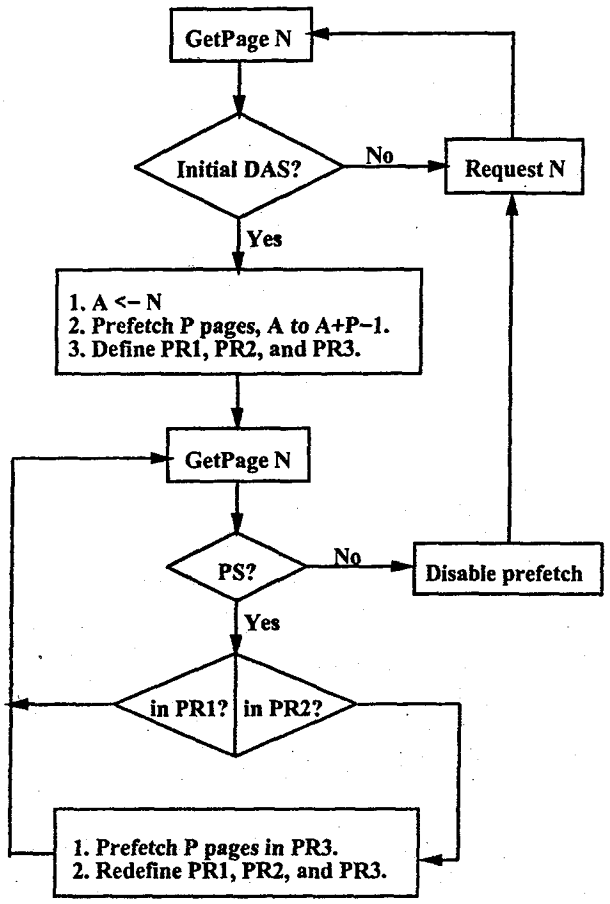
GetPage N
Initial DAS?
No
Request N
Yes
1. A <- N
2. Prefetch P pages, A to A+P-1.
3. Define PR1, PR2, and PR3.
GetPage N
PS?
No
Disable prefetch
Yes
in PR1?
in PR2?
1. Prefetch P pages in PR3.
2. Redefine PR1, PR2, and PR3.

Figure 3: The procedure for dynamic prefetch

time due to inaccurate calculations.

We first describe how sequential prefetch is triggered the first time. A page is considered Page-Sequential if it is within $P/2$ pages ahead of the previous page, where P is the prefetch quantity (PFQ) decided in the installation. If a page is Page-Sequential, DB2 determines further whether the access sequence is Data Access Sequential (DAS). Data access sequential is defined as a sequence where M or more of the last N pages are page-sequential. The initial data access sequential triggers the dynamic prefetch: P pages starting at the page being requested are loaded into buffers. The value of M/N is a threshold defined in DB2 to control when dynamic prefetch should be triggered. In current implementation, M and N have been set to 5 and 8, respectively. The numbers 5 and 8 are empirically derived.

Let A be the page that declares the initial data access sequential and triggers the first prefetch. Three ranges, $PR1$, $PR2$, and $PR3$, are defined as below:

- $\mathrm{A} \leq \mathrm{PR1} < \mathrm{A} + \mathrm{P}/2$
- $\mathrm{A} + \mathrm{P}/2 \leq \mathrm{PR2} < \mathrm{A} + \mathrm{P}$
- $\mathrm{A} + \mathrm{P} \leq \mathrm{PR3} < B$, where $B = \mathrm{A} + 2 * \mathrm{P}$.

For subsequent requests, the buffer manager first determines whether if a requested page is Page-Sequential. If not, the buffer manager disables the prefetch and keeps looking for another initial data access sequential. Otherwise, the buffer manager processes the request according to the following rules:

case (1) *the page is in PR1:* No prefetch is triggered. Buffer manager continues to process the next page request.

case (2) *the page is in PR2:* Prefetch for pages in $PR3$ is triggered. After the prefetch, three ranges and B are redefined: new $PR1$ is set to old $PR2$, new $PR2$ is set to old $PR3$, and new $PR3$ is defined as the page range starting at B for a length of P pages. After the redefinition of $PR3$, B is set to $B + \mathrm{P}$. Buffer manager continues to process the next page request with these new values.

In summary, the whole procedure of the dynamic prefetch is shown as the flowchart in Figure 3.

For example, let P be 32. Given a sequence of pages accessed

$$20, 30, 42, 50, 150, 62, 70, 76, 88, 100, 130, 152, 160$$

Table 2 shows how dynamic prefetch works. Column 1 gives the page requested. Column 2 records the action taken by buffer manager. Column 3 shows the status of the detection of the sequential access pattern.

Page accessed	BM action	Status
20	read 20	
30	read 30	Page-Sequential (PS)
42	read 42	PS
50	read 50	PS
150	read 150	
62	read 62	
70	read 70	PS
76	prefetch 76 to 107	PS and DAS
88		PS in PR1
100	prefetch 108 to 139	PS in PR2
130		disable prefetch
152	read 152	
160	prefetch 160 to 191	PS and another DAS

Table 2: An example of dynamic prefetch

6 Deferred Write

The enforcement of writing dirty pages at commit point in synchronous write protocols prolongs the time to do commit operation due to write I/O delays. Furthermore, it also stops other processes which need to update these pages. To overcome this problem, DB2 provides deferred write which batches write requests until they can be executed more efficiently with respect to the number of DASD I/O operations and disk arm movement. This is using Write Ahead Logging techniques. The benefits of using deferred write are to increase the probability of batching I/O, minimize the number of write I/Os for frequently updated pages, reduce the number of I/Os per transaction or query, and maximize I/O concurrency by scheduling multiple write engines.

6.1 Deferred Write Queue

Buffer manager normally enqueues dirty pages on the system Deferred Write Queue (DWQ). At commit point only update logs are written to DASD. Except during actual update, deferred writes can be initiated prior to, during, or after phase 2 of commit [1][2]. Because of the possibility of prior-commit writes, undo operation is provided to back out uncommitted transactions due to transaction abort. Similarly, because of the support of after-commit writes, redo operation is also provided to permanently write committed changes in case of situations like system crash.

To facilitate the separation of different datasets and related I/O requests, the deferred write queue is structured as a queue of dataset related queues. Each dataset related queue maintains a queue of dirty pages that belong to the same database dataset. In contrast to deferred write queue, a dataset related queue is also called a Vertical Deferred Write Queue (VDWQ). During a deferred write, a selected vertical deferred write queue first dequeues up to n dirty pages. If the VDWQ is not empty after dequeuing, it places itself back to the DWQ. n is an interval value of DB2 and set to 128. The dequeued pages are asynchronously written to DASD by a deferred write engine (described later in this section). After being successfully written, each page is marked clean and placed in the free buffer set. However, the content of the page is retained so if this page is accessed again before it is released from the free buffer set, no read I/O is required.

The deferred write queue (DWQ) was maintained in Last-In-First-Out (LIFO) order. When the DWQ threshold (described later in this section) is reached, i.e. the number of available buffers falls below a certain level, DB2 needs to select candidate VDWQs for writing out dirty pages. It always selects VDWQs that are newly placed on the DWQ. For each selected VDWQ, only n pages are dequeued and written back to DASD. If the selected VDWQ has more than n pages, it stays on the same relative position on the DWQ queue and could be selected again when the DWQ threshold is triggered again.

For a fairness consideration and also attempt to balance write I/Os to different DASD devices, the deferred write queue (DWQ) has been modified to be managed in First-In-First-Out (FIFO) order, as shown in Figure 4. In addition, once a vertical deferred write queue (VDWQ) is selected, it will be placed back to the tail of the DWQ queue if it has more than n dirty pages. This allows each VDWQ has equal probability of being selected when the DWQ threshold is reached.

The VDWQ used to be maintained in FIFO order. Thus, during a deferred write the first n dirty pages were dequeued. It is possible that the first n

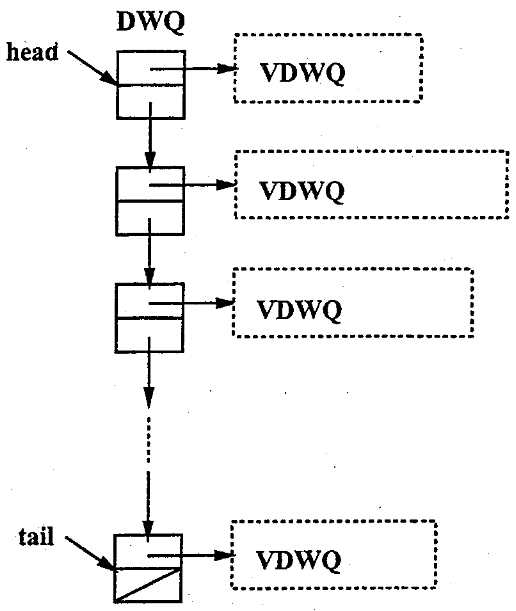

Figure 4: The Deferred Write Queue (DWQ)

dirty pages contain pages that may require frequent updates. For example, a DB2 space map page is updated more frequently than each individual data page because it covers a large range of data pages. When a write I/O is in progress for a page, the page is not allowed to be updated until the write I/O is completed. Therefore, it could have significant performance impact to transaction performance if the deferred write process constantly writes those hot pages to DASD. Frequently writing hot pages also defeats the benefit of reducing write I/O activities for database datasets. The deferred write algorithm has been enhanced to apply the LRU scheme to each VDWQ, instead of managing each VDWQ in FIFO order. When a deferred write is triggered, DB2 now chooses writing the n least recently updated pages from each VDWQ.

6.2 Asynchronous write engine

The actual operation writing dirty pages to DASD is performed as an asynchronous task by the buffer manager. The components designated for this task are called deferred write engines. The deferred write engines operate independently and in parallel with transaction processing to maximize I/O concurrency. However, similar to sequential prefetch, when the number of available buffers falls below a predetermined level, buffer manager will temporarily suspend deferred write and all writes will now occur synchronously to the updating transaction.

During a deferred write operation, a deferred write engine is dedicated for dirty pages belonging to the same vertical deferred write queue (VDWQ). Once the n pages are dequeued from the VDWQ, DB2 serially initiates write I/Os in multiples of 32-page increments under the same write engine task. The reason for setting an upper limit (i.e. 32 pages) on each chained write I/O request is to avoid holding up the device for a long period of time which may starve other tasks. Long chained I/Os could have significant performance impact to transactions that need to access data from the same DASD device.

The n dequeued pages were not sorted by physical disk address before the write I/O was initiated. Since DB2 database pages are being physically mapped to DASD in page number order, the device seek time could be minimized if write I/Os are scheduled in the page number sequence. Hence, the deferred write has been enhanced to sort the write pending pages before carrying out the write I/Os. This change improves not only write efficiency but read I/O efficiency because device queueing time is reduced.

To improve concurrency for page update, it is desirable to delay buffer locking for write I/O until it is placed on the write I/O chain. All pages dequeued from the VDWQ were locked for write I/O before they were passed to the write engine. As mentioned earlier, a write engine will serially schedule write I/Os for the n pages passed to it in multiples of 32-page increment. To improve concurrency for page update, DB2 has been enhanced to have the buffer locking done by the write engine. The write engine will only lock the page buffer for write I/O before it is placed on the write I/O chain. Therefore, within a write engine, at most 32 pages are locked with the write I/O in progress status. Once a chained write I/O request is completed, those pages are immediately unlocked.

DB2 supports multiple write engines which carry out concurrent write I/Os for different VDWQs. Though a write engine is dedicated for pages that belong to the same VDWQ, the write engine is not limited for this VDWQ only. After successfully finishing the write operation, a write engine is placed in a queue of free engines and ready for the next assignment, which may be requested by a different VDWQ. To maximize I/O throughput, DB2 allows the allocation of multiple write engines for one VDWQ during a deferred write.

6.3 Trigger conditions

Several events and thresholds defined in DB2 trigger deferred write. During a DB2 checkpoint or at dataset close point, all dirty pages are permanently written on DASD. Each VDWQ is assigned a deferred write engine to perform the write I/O. The thresholds are designed to monitor the usage of the buffer pool. By carefully choosing their values, the thresholds can prevent dirty pages from flooding the buffer pool, prevent dirty pages in a dataset from flooding the buffer pool, and free buffers as soon as a shortage of available buffers is detected.

7 Conclusion

This paper has described the design considerations of the DB2 buffer manager and its role in the DB2 system. When DB2 first came out twelve years ago, the processor memory capacity was limited and the processor speed was much slower than today. Several major buffer pool design decisions were made based on the limited memory capacity. For example, the dynamic buffer pool resizing feature implemented first was designed with a small buffer pool in mind. The temporary buffer pool expansion logic was in place to prevent DB2 process from terminating when the demand for buffer resource increases as a result of DB2 workload increases. With the large memories, this temporarily buffer pool expansion and contraction logic was no longer supported.

Several design changes were also made to ensure that DB2 can grow and scale with the growth of the processor and memory speed. In the beginning, DB2 could only process 30 transactions a second based on an IBM internal transaction workload. Today, it can perform over 960 transactions per second on the largest IBM mainframe. During the past twelve years, there has been thirty times throughput improvement. For transactions, the improvement attributable to DB2 has been about 2 to 3 times in that period. The rest is due to improvements in the hardware and in the operating system.

With the recent introduction of the S/390 parallel sysplex technology [5], DB2 has been extended to support shared data architecture across the sysplex. Major enhancements have been added to the buffer manager to support buffer coherency in a parallel sysplex environment. To support shared data, it is necessary to ensure that data cached in each DB2's buffer pool is known to other DB2s. The process of coordinating and ensuring valid data being accessed in each DB2's buffer pool is called buffer coherency. The details on extending the buffer manager to support a shared data architecture can be found in [3].

References

[1] R. A. Crus. Data recovery in IBM database 2. *IBM Systems Journal*, 23(2):178–188, 1984.

[2] D. J. Haderle and R. D. Jackson. IBM database 2 overview. *IBM Systems Journal*, 23(2):112–125, 1984.

[3] IBM Corporation. *DB2 for MVS/ESA V4.1 Data Sharing Planning and Administration*, SC26-3269 edition.

[4] IBM Corporation. *IMS/360 General Information Manual*, GH20-0765 edition.

[5] J. M. Nick, J. Y. Chung, and N. S. Bowen. Overview of ibm system/390 parallel sysplex- a commercial parallel processing system. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 488–495, Honolulu, Hawaii, April 1996.

[6] J. Rodriguez-Rosell. Empirical data reference behavior in data base systems. *IEEE Computer*, 9(11):9–13, November 1976.