Buffer Manager

Buffer Management

The database buffer is the mediator between the basic file system and the tuple oriented file system. The buffer manager's task is to make the pages addressable in main memory and to coordinate the writing of pages to disk with the log manager and the recovery manager. It should also minimize the number of actual disk accesses for doing that.

1 Functional Principles of the Database Buffer

Each relation is can be mapped onto many files (each file containing data from one relation only). Each file is viewed as a set of equal-sized pages. All the database access modules (responsible for providing associative access, implementing joins, etc.) operate on the basis of page abstractions. Each tuple is located by specifying the identifier of the page in which it is stored, and the offset within that page. A page identifier has the following structure:

```
typedef struct
    ( FILENO        fileno;   /*file to which the page
                                belongs
      unsigned int  pageno;   /* page number in the
                                file
    ) PAGEID, *PAGEIDP;
```

The page numbers grow monotonically, starting from 0, within each file. Each page of that file system is the contents of the block with the same number in the corresponding basic file.

The database access modules reference their objects by addresses that are tuples of the type (PAGEID, offset). For executing instructions on such objects, however, these objects must be located in some process's virtual memory. Moving pages between a disk and the buffer pool is the buffer manager's basic function.

The buffer manager administers a segment in virtual memory, which is partitioned into portions of the equal size called *frames*. We assume that each frame can hold exactly one page.

- Buffer per file
- Buffer per page size
- Buffer per file type

The database buffer can be declared as a simple data structure:

What is the difference between a conventional file buffer and a database buffer?

1. The caller is not returned a copy of the requested page into his address space; he gets back an address of the page in the buffer manager's domain (to avoid anomalies, like 'lost update").
2. As a consequence of a page request, other pages (probably not related to the transaction issuing the request) can be written. If the page is modified the buffer manager is informed, but it will essentially decide by its own criteria when the modified page is written out to disk.

The buffer manager provides:

- **Sharing**. Pages are made addressable in the buffer pool, which is an area of shared virtual memory accessible to all processes that run the database code.
- **Addressability**. Each access module is returned an address in the buffer pool, denoting the beginning of the frame containing the requested page.
- **Semaphore protection**. Many processes can request accesses to the same page at the same time; the buffer manager gives them the same frame address. The synchronization of these parallel accesses in order to preserve serializability is not the buffer manager problem. It has only to provide a semaphore per page that can be used for implementing e.g. locking protocol.
- **Durable storage**. The access modules inform the buffer manager if their page access has modified the page, however, the page is written out to disk by the buffer manager, probably, at a time when update transaction is already committed.

All service requests to the buffer manager refer (via pointer) to a buffer access control block that is declared as follows:

```
typedef struct
    ( PAGEID        pageid;     /*id of page in file
      PAGEPTR       pageaddr;   /*base address of
                                  page in buffer pool
                                  this entry is set by
                                  buffer manager
      int           index;      /*record within page
      semaphore*    pagesem;    /*pointer to the
                                  semaphore for the
                                  page
      Boolean       modified;   /*flag – modified
                                  page
      Boolean       invalid;    /*flag – destroyed
                                  page
    ) BUFFER_ACC_CB, *BUFFER_ACC_CB;
```

The control block tells the caller what he has to know for accessing the requested page in the buffer. Both modified and invalid flags are initialized to FALSE. It may happen that for some reason the caller fails to function properly. The transaction must be aborted, and the buffer manager must be informed that the page contains the garbage. this is indicated by setting the flag to TRUE.

The call that fills in the buffer access control block is defined by the following function:

```
Boolean     bufferfix(PAGEID pageid, LOCK_MODE
                mode, BUFFER_ACC_CBP*
                address);
/*returns TRUE if the page could be allocated, FALSE
/*otherwise, if it is allocated the address of the first byte
/*of the page header in the buffer pool is returned in
/*BUFFER_ACC_CB. The fix count is increased by 1.
/*The semaphore protecting the page is acquired in the
/*requested mode (shared or exclusive).
```

To provide the correct access to frames, database systems typically use the **FIX-USE-UNFIX** protocol.

FIX. The client requests access to a page using the bufferfix interface. The page is fixed in the buffer, that is, it is not eligible for replacement.

USE. The client uses the page with the guarantee that the pointer to the frame containing the page will remain valid.

UNFIX. The client explicitly waives further usage of the frame pointer, that is, it tells the buffer manager that it no longer wants to use that page. The buffer manager can therefore unfix the page, which means that the page is eligible for replacement.

The interface for unfixing a page is defined as follows:

```
Boolean     bufferunfix (BUFFER_ACC_CBP);
/*returns TRUE if the page could be unfixed, otherwise
/*FALSE. If the unfix is possible, the fix counter is
/*decreased y 1. If the transaction has multiple fixes on
/*the page, it must issue the corresponding number of
/*unfix operations.
```

Several concurrent transactions can share concurrent access to a buffer page at the same time. In that case, each transaction is given the same pointer to the buffer pool, and each transaction fixes the page. If one transaction unfixes the page, the page does not become eligible for replacement.

Al modules operating at the buffer interface must strictly follow the FIX-USE-UNFIX protocol, with the additional requirements of keeping the duration of a fix as short as possible (even if a module knows that it might need access to a page again later on).

The FIX-USE-UNFIX operations are among the most frequently used primitives in a database system, and thus should be very fast.

Two additional operations are needed during normal processing.

```
Boolean     emptyfix(PAGEID pageid, LOCK_MODE
                mode, BUFFER_ACC_CBP*
                address);
/*returns TRUE if the page could be allocated, FALSE
/*otherwise. The function requests empty page to be
/*allocated in buffer. The identifier of the empty page
/*has to be provided by the caller. The buffer manager
/*returns a pointer to the buffer access CB like
/*bufferfix.
```

Whenever a page that was modified in the buffer pool by a successful transaction is to be replaced, it must be forced to durable storage, before it can be removed from the buffer pool. Since the basic file system is not accessible to the higher-level modules, they cannot issue a write operation. This has to be done by the buffer manager, so, there must be an interface for telling him about it.

```
Boolean     flush (BUFFER_ACC_CBP);
/*returns TRUE if the page was written to the file,
/*otherwise FALSE. The page is written to its block in
/*the file. The modified flag is set to FALSE, and the
/*page remains in the buffer. This function can be called
/*by any client. The buffer manager will acquire a
/*shared semaphore on the page while writing it.
```

2 Logging and Recovery from the Buffer's Perspective

The buffer manager is completely autonomous in handling the incoming requests. The decision which (modified) page and when write out to disk is left to the buffer manager. The buffer manager may apply LRU buffering to optimize overall system performance. Is it true?

It is true when we are talking about "normal" (non-transactional) file buffer. However, in a transactional system, the buffer manager must make sure not to violate the ACID properties, this requires some synchronization with both the log manager and the transaction manager.

Each time the page is written out to disk after modification, the old state of that page is lost. On the other hand, if a transaction modifies a page and then commits, and the buffer manager has not yet written that page to disk, a subsequent crash will leave a file system with the old (invalid) page.

Therefore, if the buffer manager decides to write out a modified page belonging to an incomplete transaction, atomicity can be violated; if it does not write out the modified page of a committed transaction, durability can be violated.

The buffer manager's rule in preserving transaction atomicity and durability

State of transaction	Page A	Page B	State of the database
Aborted	In buffer (old)	In buffer (old)	Consistent (old)
Aborted	In buffer (old)	On disk (new)	Inconsistent
Aborted	On disk (new)	In buffer (old)	Inconsistent
Aborted	On disk (new)	On disk (new)	Inconsistent
Committed	In buffer (old)	In buffer (old)	Inconsistent
Committed	In buffer (old)	On disk (new)	Inconsistent
Committed	On disk (new)	In buffer (old)	Inconsistent
Committed	On disk (new)	On disk (new)	Consistent (new)

At the time of crash, a page is either "in buffer" or "on disk". If "in buffer", then after the crash only the old state is available. If "on disk", then after the crash only the new state is available.

If the transaction T modifies two pages, A and B, and if there are completely autonomous decisions by the buffer manager, then any of the eight constellations shown above can occur.

In order to guarantee the correct execution of recovery during restart, the buffer manager and the log manager have to exchange information at run time that indicates whether a given log entry should be applied to a page in the database. The simplest way to do this is to assign a *state identifier* or *version number* to a page and to record with each log entry the value of the state identifier of the page to which it refers. During recovery, the state of the page found on disk can then be compared with the state recorded with the log entry. To this purpose we use the log sequence numbers (LSN).

There are three rules governing the interaction between the buffer manager, the log manager, and the recovery manager that take care of the LSN dependencies: the fix rule, the write-ahead-log rule, and the force-at-commit rule. One should keep in mind, however, that these rules essentially constraint the buffer manager's autonomy of when to write which page to disk.

Some buffer managers restrict themselves to a smaller class of protocols either to simplify the implementation or to optimize system performance.

Steal versus No-Steal Buffer Management

A page with modifications by an uncommitted transaction is a dirty page until either commit or rollback processing for that transaction has been completed. The buffer manager can either distinguish dirty pages from clean pages when deciding which page to remove from the buffer pool, or it can ignore the update status of a page.

In the later case, the buffer manager uses a ***steal policy***, which means pages can be written out to disk even if the transaction having modified the pages is still active.

The alternative is the ***no-steal policy***, in which case all dirty pages are retained in the buffer pool until the final outcome of the transaction has been determined.

Advantages:

The steal policy implies that rollback of a transaction requires access to pages on disk in order to reestablish their old state.

With the no-steal policy, no page on disk ever has to be touched when rolling back a transaction. Consequently, no log information for UNDO procedure will be needed. Roll back of a transaction during normal processing is also facilitated by the no-steal policy since all pages modified by such a transaction are simply marked "invalid" by the buffer manager. The problem with this policy is the size of the buffer pool + necessity of page locking.

Force versus No-Force Buffer Management

Force versus no-force concerns writing of clean pages from the buffer pool. The simple question here is: who decide, and when, that a modified page is written out to disk? There are two basic approaches:

Force policy. At phase 1 of a transaction's commit, the buffer manager locates all pages modified by that transaction and writes the pages to disk.

No-force policy. This is the liberal counterpart. A page, whether modified or not, stays in the buffer as long as it is still needed. Only if it becomes the replacement victim it will be written to disk.

Advantage of the force policy – it avoids any REDO recovery during restart. If transaction is successfully committed, then, by definition, all its modified pages must be on disk.

Why not use it as a standard buffer management policy?

Because of "hotspot" pages.

The force policy simplifies restart, because no work needs to be done for transactions that committed before the crash – it avoids REDO. The price for that is significantly more I/O for frequently modified pages. Another drawback is that a transaction will not be completed before the last write has been executed successfully, and the response time may be increased significantly as a consequence. With no-force policy, the only synchronous write operation goes to the log, and the volume of data to be written is usually about two orders of magnitude less.