Lecture #06: Buffer Pools

15-445/645 Database Systems (Fall 2024)
`https://15445.courses.cs.cmu.edu/fall2024/`
Carnegie Mellon University
Andy Pavlo

1 Introduction

This semester, our focus will be on *disk-oriented database management systems*. A disk-oriented architecture means that the DBMS's primary storage location is in persistent storage, like a hard drive (HDD) or flash storage (SSDs). This is different from an in-memory DBMS, where data is stored in volatile memory.

In the Von Neumann architecture, data must be in memory before we can operate on it (a few exceptions apply for running operations directly on persistent storage). Any DBMS must be able to efficiently move data back and forth between disk and memory if it wants to operate on large amounts of data. The DBMS can achieve this with a Buffer Pool Manager. A diagram of this interaction is shown in Figure 1.

At a high level, the *buffer pool manager* is responsible for moving physical pages of data back and forth from buffers in main memory to persistent storage. It also behaves as a cache, keeping frequently used pages in memory for faster access, and evicting unused or cold pages back out to storage.

From the DBMS's perspective, it should "appear" as if the entire database resides in memory, when in reality the database might occupy more space than the available memory in the system. It should not have to worry about how data is fetched into memory or how it is managed. The DBMS only requires valid pointers to memory locations to perform its operations.

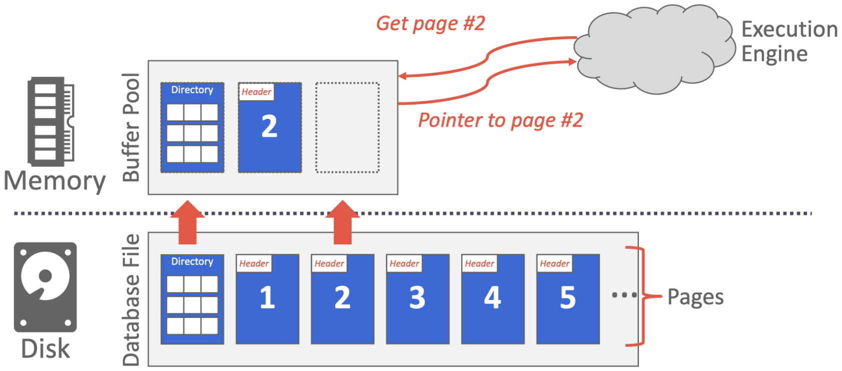

Figure 1: Disk-oriented DBMS.

There are two main things that we will try to optimize for:

1. **Spatial Control**, which refers to where pages are physically located on disk. The goal of spatial control is to keep pages that are used together often as physically close together as possible on disk. This can potentially help with prefetching and other optimizations.
2. **Temporal Control**, which refers to when pages have been brought into memory and when they should be written back out to disk. Temporal control aims to minimize the number of stalls from having to read data from disk.

2 Buffer Pool

The buffer pool is an in-memory cache of pages between memory and disk. It is essentially a large memory region allocated inside of the database to temporarily store pages. It is organized as an array of fixed-size **frames**. When the DBMS requests a page, the buffer pool manager first checks if the page is already stored in a frame of memory, and if it is not found, the page is read / copied into a free frame from disk. We consider the buffer pool manager as a write-back cache, where dirty pages are buffered and not written to disk immediately on mutation. This is in contrast to a write-through cache, where any changes are immediately propagated to disk.

The DBMS needs this buffer pool memory for many different things (just as most computer programs need memory for different types of data structures):

- Tuple Storage and Indexes
- Sorting and Join Buffers
- Query and Dictionary Caches
- Maintenance and Log Buffers

Depending on the implementation, the things listed above do not always have to be backed by disk / the buffer pool manager, and can simply rely on memory allocators like `malloc`.

A **page directory** is also maintained on disk, which is the mapping from page IDs to page locations in database files. All changes to the page directory must be recorded on disk to allow the DBMS to find on restart. It is often (not always) kept in memory to minimize latency to page accesses since it has to be read before accessing any physical page.

See Figure 2 for a diagram of the buffer pool's memory organization.

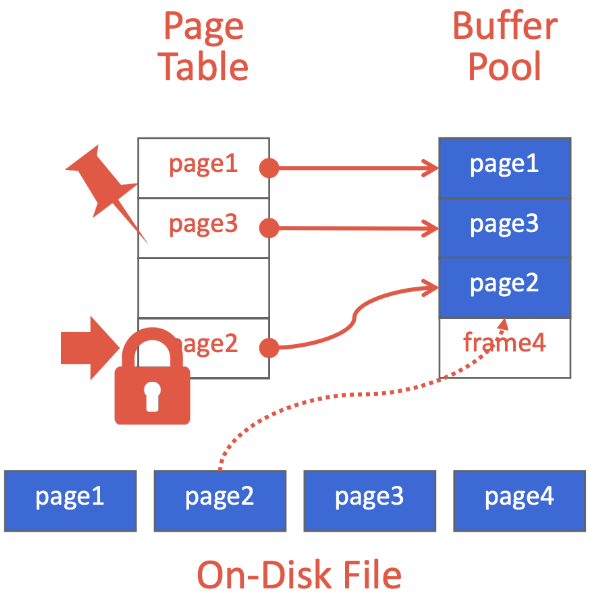

Figure 2: Buffer pool organization and meta-data

Buffer Pool Metadata

The buffer pool must maintain certain meta-data in order to be used efficiently and correctly.

The **page table** is an in-memory hash table that keeps track of pages that are currently in memory. It maps page IDs to frame locations in the buffer pool. Since the order of pages in the buffer pool does not necessarily reflect the order on the disk, this extra indirection layer allows for the identification of page locations in the pool. The page table also maintains additional meta-data per page, a dirty flag, and a pin / reference counter.

The **dirty flag** is set by a thread whenever it modifies a page. This indicates to the storage manager that the page must be written back to disk before eviction.

The **Pin / reference counter** tracks the number of threads that are currently accessing that page (either reading or modifying it). A thread has to increment the counter before they access the page. If a page's pin count is greater than zero, then the storage manager is not allowed to evict that page from memory. Pinning does not prevent other transactions from accessing the page concurrently. If the buffer pool runs out of non-pinned pages to evict and the buffer pool is full, an out-of-memory error will be thrown.

Page Table vs. Page Directory

To clarify the difference between the page table and the page directory in database management systems:

- The **page directory** is the mapping from page ids to page locations on the physical database files. All changes must be recorded on disk to allow the DBMS to find them on restart.
- The **page table** is the mapping from page ids to a copy of the page in buffer pool frames. This is an in-memory data structure that does not need to be stored on disk.

3 Operating Systems vs. Database Management Systems

Locks vs. Latches

We need to make a distinction between locks and latches when discussing how the DBMS protects its internal elements.

Locks: A lock is a higher-level, logical primitive that protects the contents of a database (e.g., tuples, tables, databases) from other transactions. Database systems can expose to the user which locks are being held as queries are run. Locks need to be able to roll back changes.

Latches: A latch is a low-level protection primitive that the DBMS uses for the critical sections in its internal data structures (e.g., hash tables, regions of memory). Latches are held for only the duration of the operation being made. Latches do not need to be able to roll back changes. This is often implemented with simple language primitives like mutexes and/or conditional variables.

Why not use the OS?

You may wonder why we need to use a buffer pool manager instead of simply relying on the Operating System's page cache. There are several problems with using something like memory-mapped files (`mmap`):

1. Transaction Safety: The OS can flush dirty pages at any time.
2. I/O Stalls: The DBMS doesn't know which pages are in memory. The OS might stall a thread on a page fault.
3. Error Handling: It is difficult to validate pages. Any page access can cause a `SIGBUS` signal that the DBMS then must handle.
4. Performance Issues: Internal OS data structure contention. TLB shootdowns.

The are *some* solutions to *some* of these problems (`madvise`, `mlock`, `msync`), but using these syscalls to get the OS to behave correctly is just as onerous as managing memory yourself.

The DBMS almost always wants to control things itself and can do a better job than the OS:

1. Flushing dirty pages to disk in the correct order.
2. Specialized prefetching.
3. Better buffer replacement policies.
4. Thread / process scheduling.

The Operating System is ***not*** your friend.

4 Buffer Replacement Policies

When the DBMS needs to free up a frame to make room for a new page, it must decide which page to evict from the buffer pool, and uses a *replacement policy* to make this decision. The implementation goals of replacement policies are correctness, accuracy, speed, and metadata overhead.

Note: Remember that a pinned page cannot be evicted.

Least Recently Used (LRU)

The Least Recently Used replacement policy maintains a timestamp of when each page was last accessed. The DBMS evicts the page with the oldest timestamp. This timestamp can be stored in a separate data structure, such as a queue that keeps pages in sorted order to reduce the search time on eviction. However, keeping the data structure sorted and storing a large timestamp has a prohibitive overhead.

CLOCK

The CLOCK policy is an approximation of LRU without needing a separate timestamp per page. In the CLOCK policy, each page is given a reference bit. When a page is accessed, it is set to 1. **Note:** Some implementations may allow an actual ref counter greater than 1.

To visualize this, organize the pages in a circular buffer with a "clock hand". When an eviction is requested, sweep the hand and check if a page's bit is set to 1. If yes, set it to zero, if no, then evict, bring in the new page in its place, and move the hand forward. Additionally, the clock remembers the position between evictions.

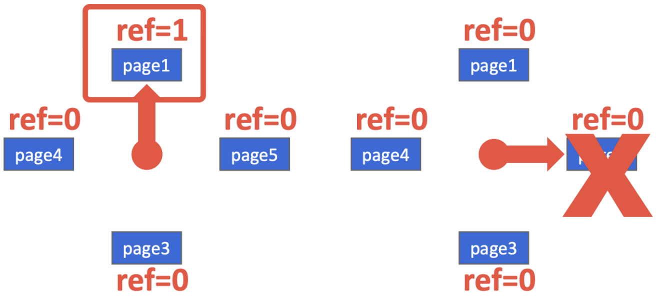

Figure 3: Visualization of CLOCK replacement policy. Page 1 is referenced and set to 1. When the clock hand sweeps, it sets the reference bit for page 1 to 0 and evicts page 5.

Issues

There are a number of problems with LRU and CLOCK replacement policies.

LRU and CLOCK are both susceptible to **sequential flooding**, where the buffer pool's contents are polluted due to a sequential scan. Since sequential scans read many pages quickly, the buffer pool fills up and pages from other queries are evicted as they would have earlier timestamps. In this scenario, the most recent timestamp is not the optimal one to evict.

LRU also does not account for the frequency of accesses. For example, if we have a page that is frequently accessed over time, we do not want to evict it if it wasn't accessed recently.

Alternatives

There are three solutions to address the shortcomings of LRU and CLOCK policies.

One solution is **LRU-K** which tracks the history of the last K references as timestamps and computes the interval between subsequent accesses. This history is used to predict the next time a page is going to be accessed. However, this again has a higher storage overhead. Additionally, need to maintain an in-memory cache of recently evicted pages to prevent thrashing and have some history for recently evicted pages.

An approximation for LRU-2 done by SQL is to have two linked lists and only evict from the old list. Whenever a page is accessed and it is already in the old list put it at the start of the young queue, else put it at the start of the old list.

Another optimization is **localization** per query. The DBMS chooses which pages to evict on a per query / transaction basis. This minimizes the pollution of the buffer pool from each query.

Lastly, **priority hints** allow transactions to tell the buffer pool whether page is important or not based on the context of each page during query execution.

Dirty Pages

Pages / frames keep track of a dirty flag / bit that denotes a page that has been modified by the DBMS. There are two cases in handling eviction of dirty pages. The fast path is when the page is not dirty, and the buffer pool manager can simply drop it. The slow path is when the page is dirty, and the buffer pool manager must write the changes back to disk to ensure data synchronization between memory and disk.

One way to avoid the problem of having to incur the cost of writing out pages is **background writing**. Through background writing, the DBMS can periodically walk through the page table and write dirty pages to disk. When a dirty page is safely written, the DBMS can either evict the page or just unset the dirty flag.

5 Disk I/O and OS Cache

The OS / hardware will try to maximize disk bandwidth by reordering and batching I/O requests. But they do not know which I/O requests are more important than others.

The DBMS maintains internal queue(s) to track page read/write requests from the entire system. The priority of the tasks are determined based on several factors:

- Sequential vs. Random I/O
- Critical Path Task vs. Background Task
- Table vs. Index vs. Log vs. Ephemeral Data
- Transaction Information
- User-based SLAs

The OS doesn't know these things and is going to get into the way. Most disk operations go through the OS API. Unless explicitly told otherwise, the OS maintains its own filesystem cache (aka page / buffer cache).

Most DBMS use Direct I/O (via the `O_DIRECT` flag) to bypass the OS's cache to avoid redundant copies of pages and having to manage different eviction policies. **PostgreSQL** is an example of a database system that uses the OS's page cache.

Side Note: `fsync` by default has silent errors and on errors marks the page as clean.

Again, the Operating System is ***not*** your friend.

6 Buffer Pool Optimizations

There are several ways to optimize a buffer pool to tailor it to the application's workload.

Multiple Buffer Pools

The DBMS can maintain multiple buffer pools for different purposes (i.e. per-database buffer pool, per-page type buffer pool). Then, each buffer pool can adopt local policies tailored to the data stored inside of it. This method can help reduce latch contention and improve locality. Object IDs and hashing are two approaches to mapping desired pages to a specific buffer pool. **Object IDs** involve extending the record IDs to have an object identifier. A mapping from objects to specific buffer pools can be maintained via these object IDs. This allows a finer-grained control over buffer pool allocations but has a storage overhead. Another approach is **hashing**, where the DBMS hashes the page ID to select which buffer pool to access. This is a more general and uniform approach.

Pre-fetching

The DBMS can also be optimized by pre-fetching pages based on the query plan. While the first set of pages is being processed, the second can be pre-fetched into the buffer pool. This method is commonly used by DBMSs when accessing many pages sequentially during a sequential scan. It is also possible for a buffer pool manager to prefetch leaf pages in a tree index data structure benefiting index scans. Note that this does not necessarily need to be the next physical page on disk, but instead the next logical page in the leaf scan.

Scan Sharing (Synchronized Scans)

Query cursors can reuse data retrieved from storage or operator computations. This allows multiple queries to attach to a single cursor that scans a table. If a query starts a scan and there is another active scan, then the DBMS will attach the second query's cursor to the existing cursor. The DBMS keeps track of where the second query joined with the first so that it can finish the scan when it reaches the end of the data structure. This satisfies correctness since the order of scans is not guaranteed by a DBMS and is often useful when a table is frequently scanned.

Note: Continuous scan sharing is an academic prototype that constantly runs a sequential scan for certain tables and queries can hop on and off. However, this is very wasteful and costly when paying per I/O.

Buffer Pool Bypass

The sequential scan operator will not store fetched pages in the buffer pool to avoid overhead. Instead, memory is local to the running query. This works well if an operator needs to read a large sequence of pages that are contiguous on disk and will not be used again. Buffer Pool Bypass can also be used for temporary data (sorting, joins).

7 Conclusion

The DBMS can almost always manage memory better than the OS. It can leverage the semantics about the query plan to make better decisions:

- Evictions
- Allocations
- Pre-fetching

To reiterate, the Operating System is ***not*** your friend.