Lecture #06: Buffer Pools

15-445/645 Database Systems (Spring 2024)
https://15445.courses.cs.cmu.edu/spring2024/
Carnegie Mellon University
Jignesh Patel

1 Introduction

The DBMS is responsible for managing its memory and moving data back and forth from the disk. Since, for the most part, data cannot be directly operated on the disk, any database must be able to efficiently move data represented as files from its disk into memory so that it can be used. A diagram of this interaction is shown in Figure 1. Ideally from the execution engine's perspective, it should "appear" as if all the data is in memory. It should not have to worry about how data is fetched into memory or how it is managed. It only requires pointers to memory locations to perform its operations.

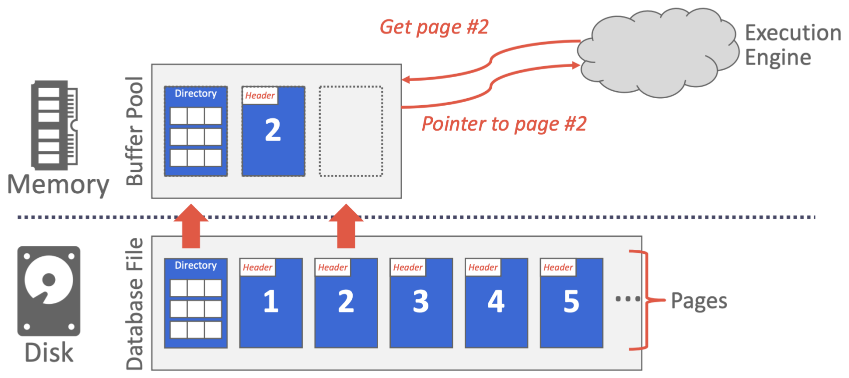

Figure 1: Disk-oriented DBMS.

Another way to think of this problem is in terms of spatial and temporal control.

Spatial Control refers to where pages are physically written on disk. The goal of spatial control is to keep pages that are used together often as physically close together as possible on disk to possibly help with prefetching and other optimizations.

Temporal Control is deciding when to read pages into memory and when to write them to disk. Temporal control aims to minimize the number of stalls from having to read data from disk.

2 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 by simple language primitives like mutexs and/or conditional variables.

3 Buffer Pool

The *buffer pool* is an in-memory cache of pages read from 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 pages. Each array entry is called a **frame**. When the DBMS requests a page, it is first searched in the buffer pool, if not found then it is copied from disk into one of the frames. Dirty pages are buffered and not written back immediately (will talk more about this below).

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.

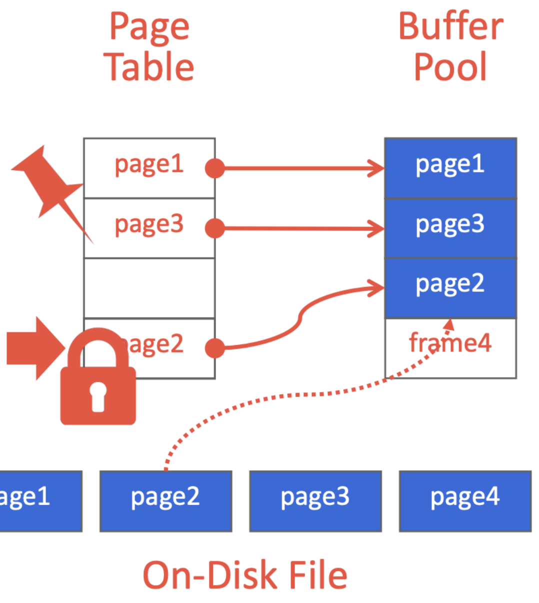

Figure 2: Buffer pool organization and meta-data

Buffer Pool Meta-data

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.

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.

Memory Allocation Policies

Memory in the database is allocated for the buffer pool according to two policies.

Global policies deal with decisions that the DBMS should make to benefit the entire workload that is being executed. It considers all active transactions to find an optimal decision for allocating memory.

An alternative is **local policies**, which makes decisions that will make a single query or transaction run faster, even if it isn't good for the entire workload. Local policies allocate frames to a specific transaction without considering the behavior of concurrent transactions. However, it will still support the sharing of frames across transactions.

Most systems use a combination of both global and local policies.

4 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 buffer pool.

Object IDs involve extending the record IDs to have an object identifier. Then through the object identifier, a mapping from objects to specific buffer pools can be maintained. 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. A more generic and uniform approach.

Pre-fetching

The DBMS can also be optimized by pre-fetching pages based on the query plan. Then, 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 like 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:** This need not necessarily be the next physical page on disk but 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).

5 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. To do so it uses a replacement policy to decide which page to evict. The implementation goals of replacement policies are improved correctness, accuracy, speed, and meta-data 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 picks to evict the page with the oldest timestamp. This timestamp can be stored in a separate data structure, such as a queue, to allow for sorting and improve efficiency by reducing sort 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.

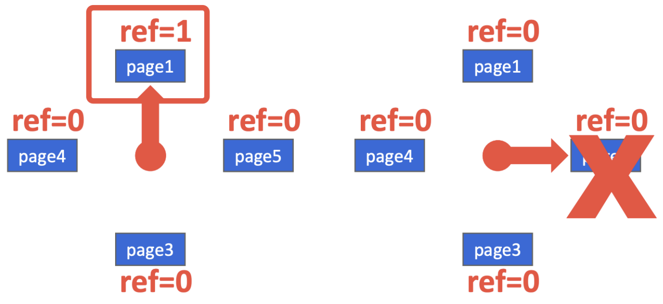

Figure 3: Visualization of CLOCK replacement policy. Page 1 is referenced and set to 1. When the clock hand sweeps, it sets the reference bit for page 1 to 0 and evicts page 5.

Issues

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

Firstly, LRU and CLOCK are 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.

Secondly, recently used does not account for the frequency of accesses. (E.g. the page directory may be frequently accessed but not recently but it should not be evicted.)

Aternatives

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 transaction/query 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

There are two methods to handling pages with dirty bits. The fastest option is to drop any page in the buffer pool that is not dirty. A slower method is to write back dirty pages to disk to ensure that its changes are persisted.

These two methods illustrate the trade-off between fast evictions versus dirty writing pages that will not be read again in the future.

One way to avoid the problem of having to write out pages unnecessarily 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.

6 Disk I/O and OS Cache

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:

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

Most disk operations go through the OS API. Unless explicitly told otherwise, the OS maintains its own filesystem cache.

However, most DBMS use direct I/O (O_DIRECT) to bypass the OS's cache to avoid redundant copies of pages and having to manage different eviction policies. **Postgres** 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. OS is not your friend :)

7 Other Memory Pools

The DBMS needs memory for things other than just tuples and indexes. These other memory pools may not always backed by disk depending on implementation.

- Sorting + Join Buffers
- Query Caches
- Maintenance Buffers
- Log Buffers
- Dictionary Caches