

Carnegie
Mellon
University

Intro to Database
Systems (15-445/645)

Lecture #06

Memory & Disk I/O Management

FALL 2023 » Prof. Andy Pavlo • Prof. Jignesh Patel



ADMINISTRIVIA

Homework #2 is now due Wednesday October 4th @ 11:59pm

Project #1 is due Sunday October 2nd @ 11:59pm
→ Q&A Session: **Monday September 18th @ 6:30pm**
→ Special Office Hours: **Saturday October 1st @ 3pm-5pm**

LAST CLASS

Problem #1: How the DBMS represents the database in files on disk.

Problem #2: How the DBMS manages its memory and move data back-and-forth from disk.

DATABASE STORAGE

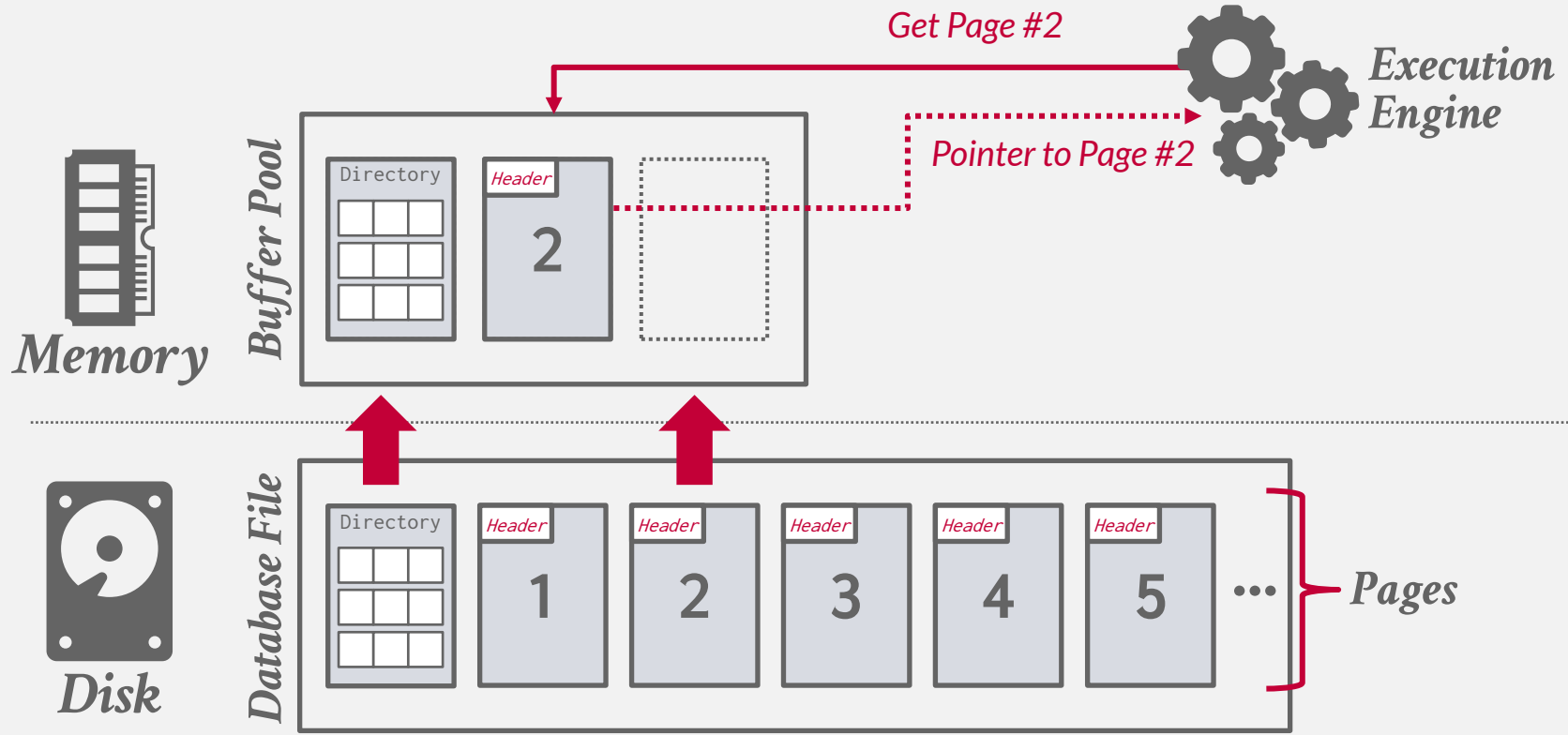
Spatial Control:

- Where to write pages on disk.
- The goal is to keep pages that are used together often as physically close together as possible on disk.

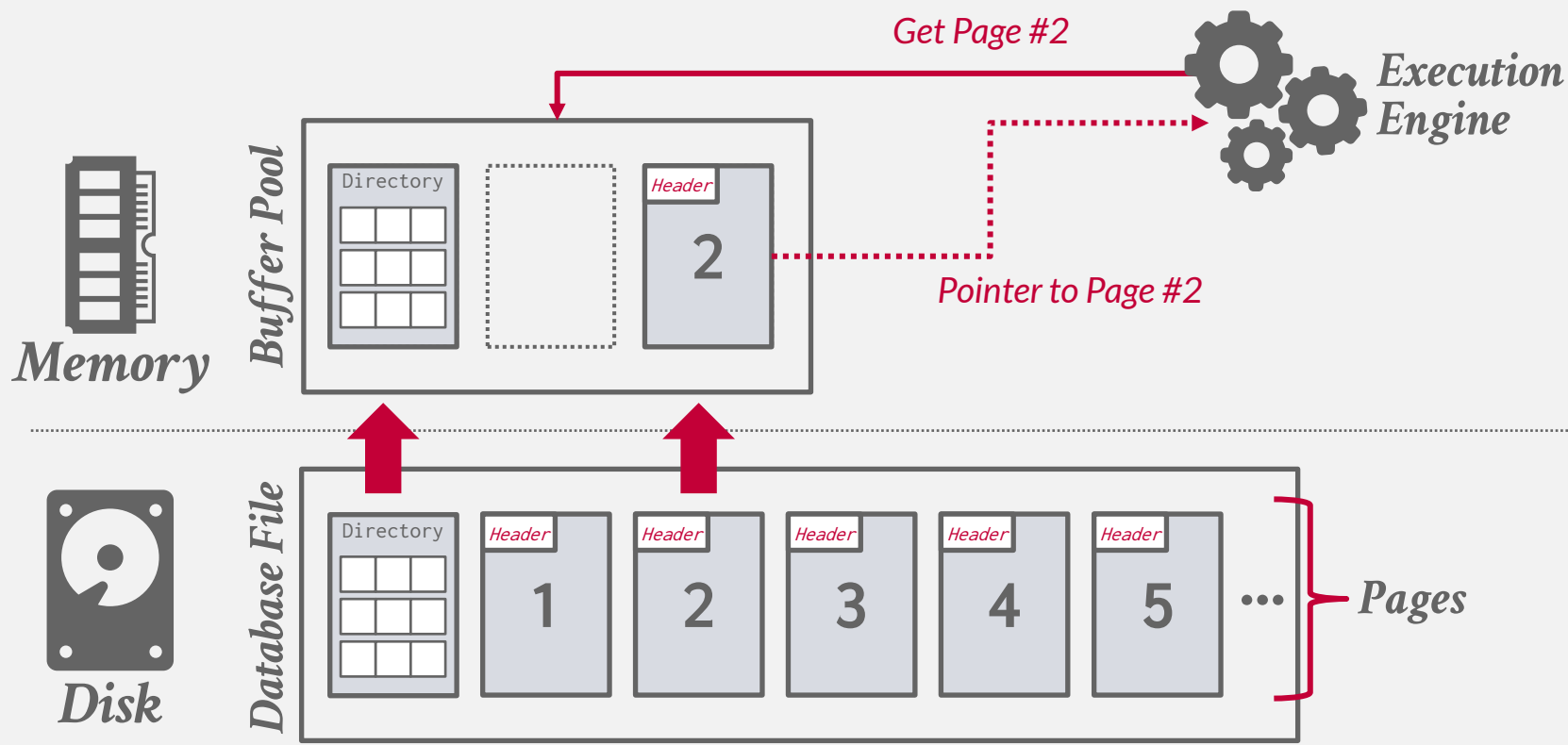
Temporal Control:

- When to read pages into memory, and when to write them to disk.
- The goal is to minimize the number of stalls from having to read data from disk.

DISK-ORIENTED DBMS



DISK-ORIENTED DBMS



TODAY'S AGENDA

Buffer Pool Manager

Disk I/O Scheduling

Replacement Policies

Other Memory Pools

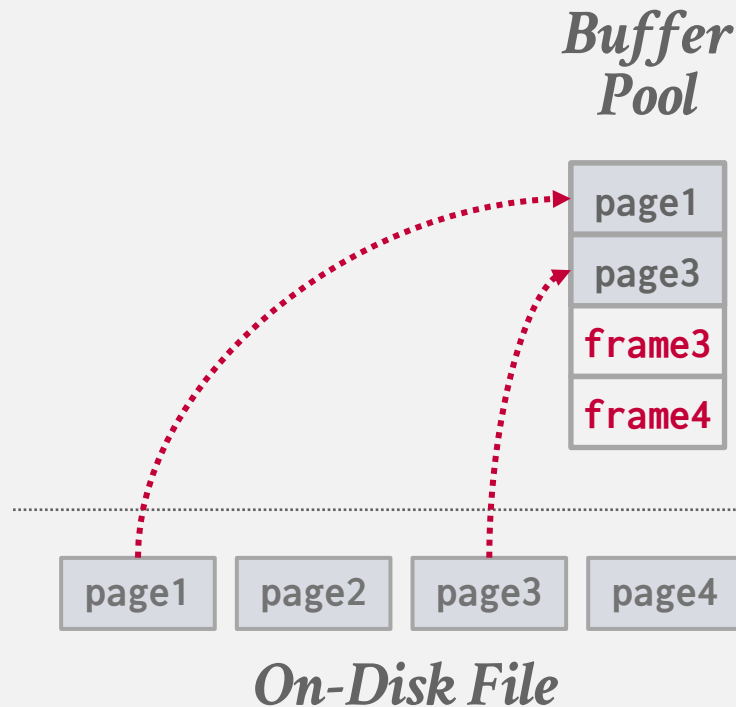
BUFFER POOL ORGANIZATION

Memory region organized as an array of fixed-size pages.

An array entry is called a **frame**.

When the DBMS requests a page, an exact copy is placed into one of these frames.

Dirty pages are buffered and not written to disk immediately
→ Write-Back Cache



BUFFER POOL META-DATA

The **page table** keeps track of pages that are currently in memory.

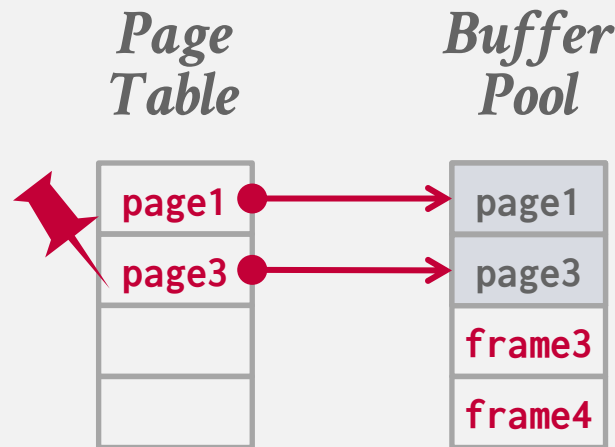
→ Usually a fixed-size hash table protected with latches to ensure thread-safe access.

Additional meta-data per page:

→ **Dirty Flag**

→ **Pin/Reference Counter**

→ **Access Tracking Information**



On-Disk File

BUFFER POOL META-DATA

The **page table** keeps track of pages that are currently in memory.

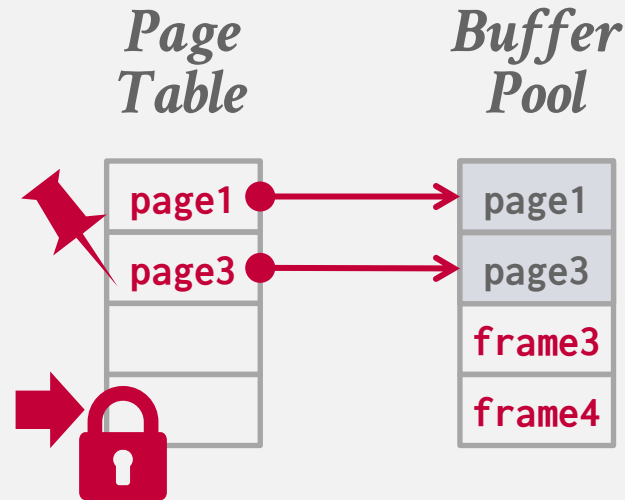
→ Usually a fixed-size hash table protected with latches to ensure thread-safe access.

Additional meta-data per page:

→ **Dirty Flag**

→ **Pin/Reference Counter**

→ **Access Tracking Information**



On-Disk File

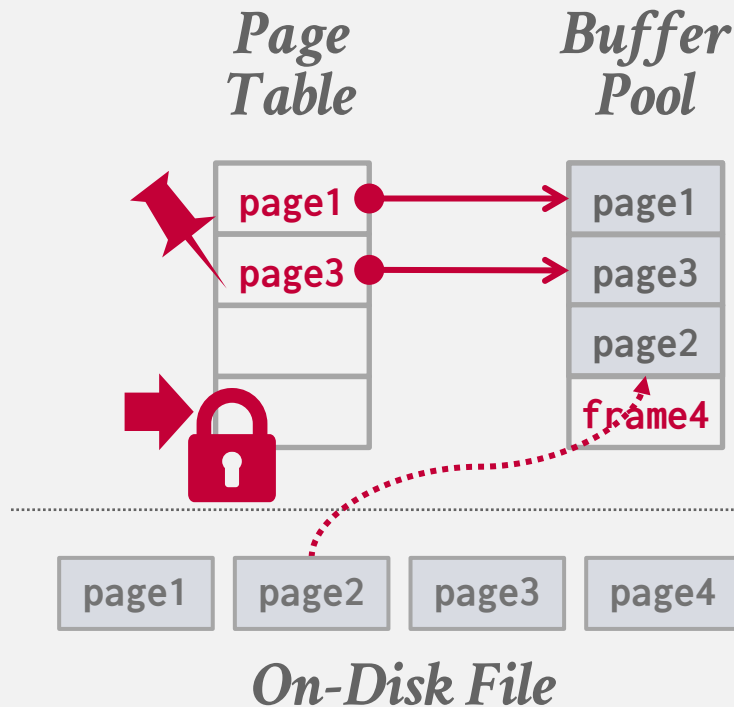
BUFFER POOL META-DATA

The **page table** keeps track of pages that are currently in memory.

→ Usually a fixed-size hash table protected with latches to ensure thread-safe access.

Additional meta-data per page:

- **Dirty Flag**
- **Pin/Reference Counter**
- **Access Tracking Information**



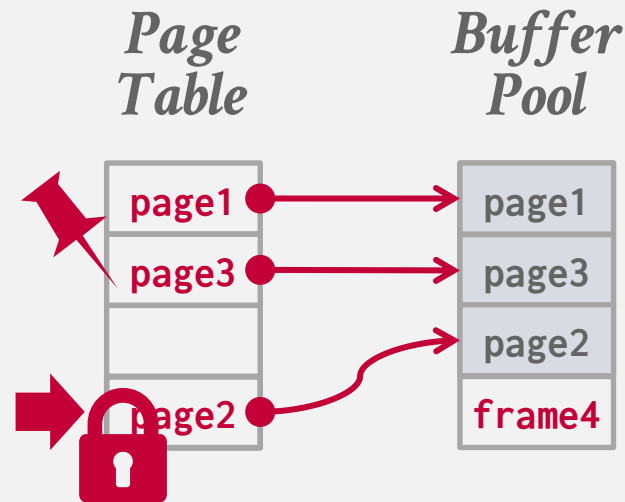
BUFFER POOL META-DATA

The **page table** keeps track of pages that are currently in memory.

→ Usually a fixed-size hash table protected with latches to ensure thread-safe access.

Additional meta-data per page:

- **Dirty Flag**
- **Pin/Reference Counter**
- **Access Tracking Information**



On-Disk File

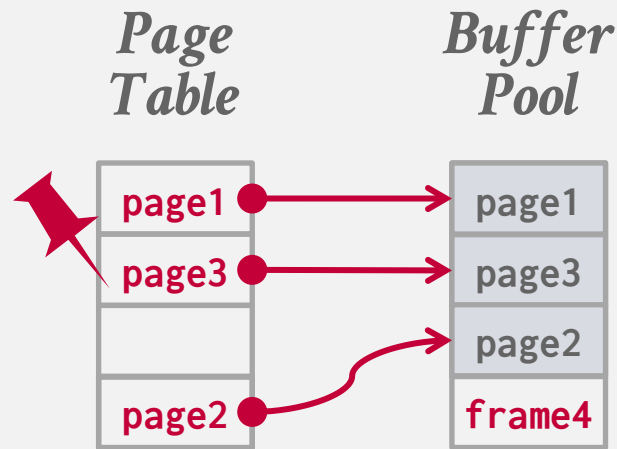
BUFFER POOL META-DATA

The **page table** keeps track of pages that are currently in memory.

→ Usually a fixed-size hash table protected with latches to ensure thread-safe access.

Additional meta-data per page:

- **Dirty Flag**
- **Pin/Reference Counter**
- **Access Tracking Information**



On-Disk File

LOCKS VS. LATCHES

Locks:

- Protects the database's logical contents from other transactions.
- Held for transaction duration.
- Need to be able to rollback changes.

Latches:

- Protects the critical sections of the DBMS's internal data structure from other threads.
- Held for operation duration.
- Do not need to be able to rollback changes.

← Mutex

PAGE TABLE VS. PAGE DIRECTORY

The **page directory** is the mapping from page ids to page locations in the database files.

→ All changes must be recorded on disk to allow the DBMS to find 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.

ALLOCATION POLICIES

Global Policies:

→ Make decisions for all active queries.

Local Policies:

→ Allocate frames to a specific queries without considering the behavior of concurrent queries.

→ Still need to support sharing pages.

BUFFER POOL OPTIMIZATIONS

Multiple Buffer Pools

Pre-Fetching

Scan Sharing

Buffer Pool Bypass

MULTIPLE BUFFER POOLS

The DBMS does not always have a single buffer pool for the entire system.

- Multiple buffer pool instances
- Per-database buffer pool
- Per-page type buffer pool

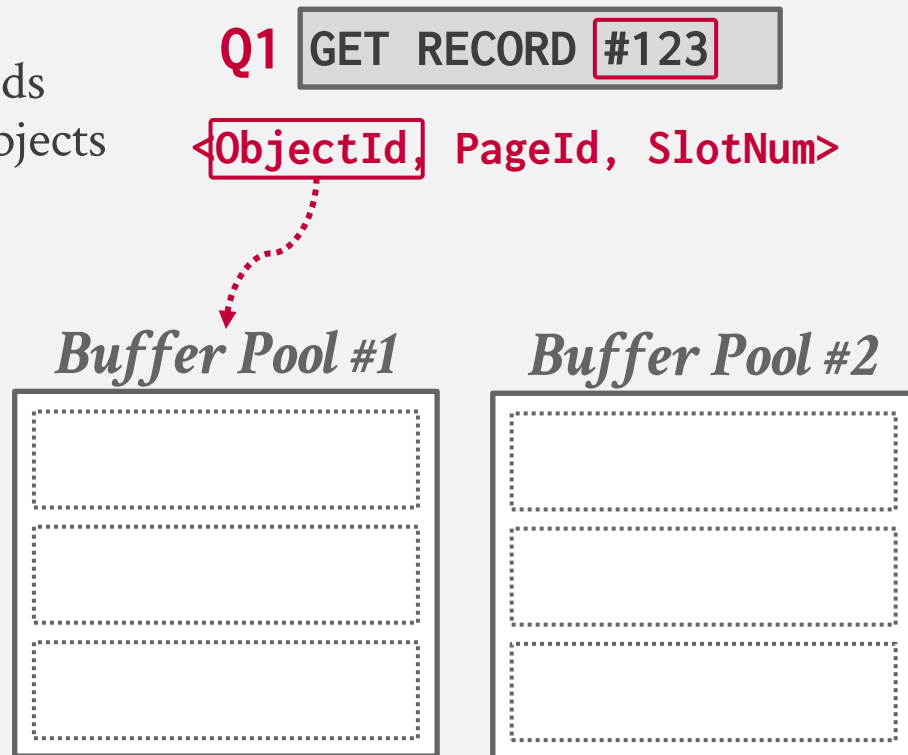
Partitioning memory across multiple pools helps reduce latch contention and improve locality.



MULTIPLE BUFFER POOLS

Approach #1: Object Id

→ Embed an object identifier in record ids and then maintain a mapping from objects to specific buffer pools.



MULTIPLE BUFFER POOLS

Approach #1: Object Id

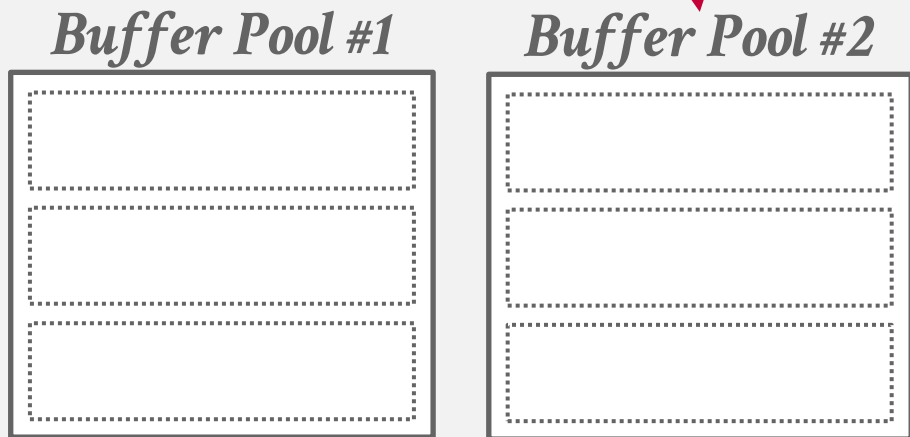
→ Embed an object identifier in record ids and then maintain a mapping from objects to specific buffer pools.

Approach #2: Hashing

→ Hash the page id to select which buffer pool to access.

Q1 GET RECORD #123

$\text{HASH}(123) \% n$

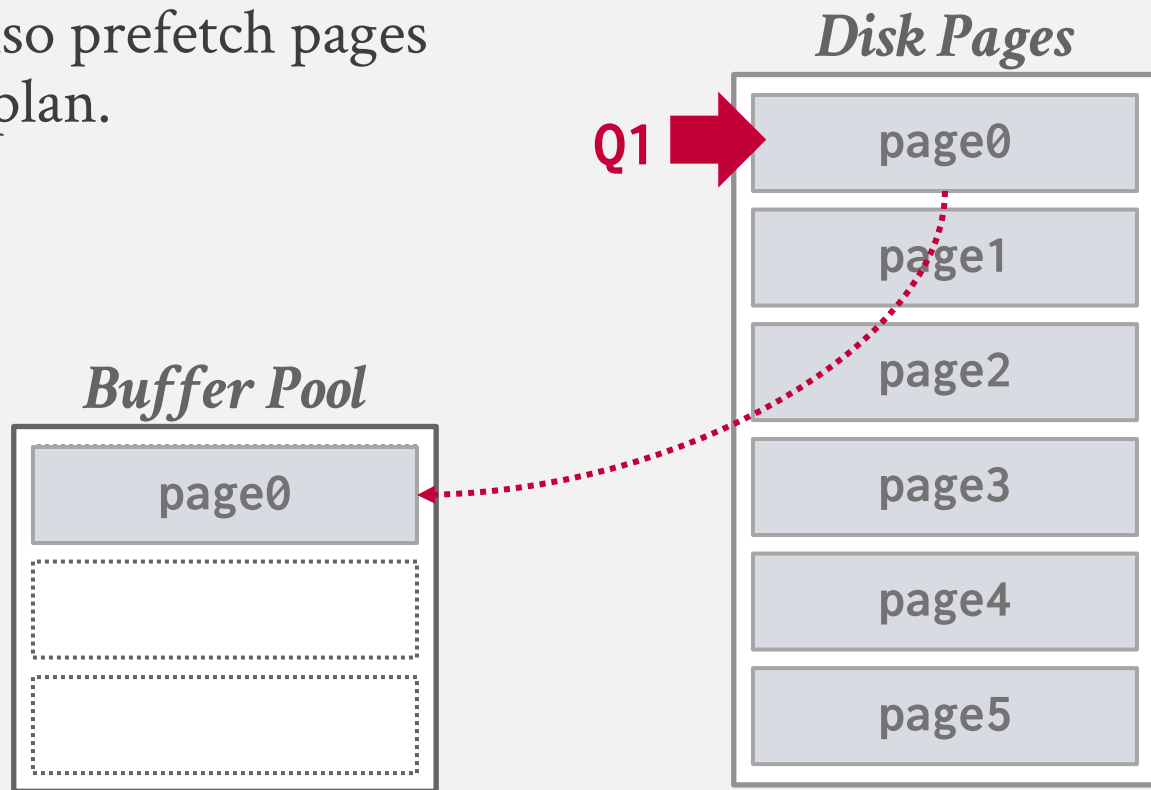


PRE-FETCHING

The DBMS can also prefetch pages based on a query plan.

→ Sequential Scans

→ Index Scans

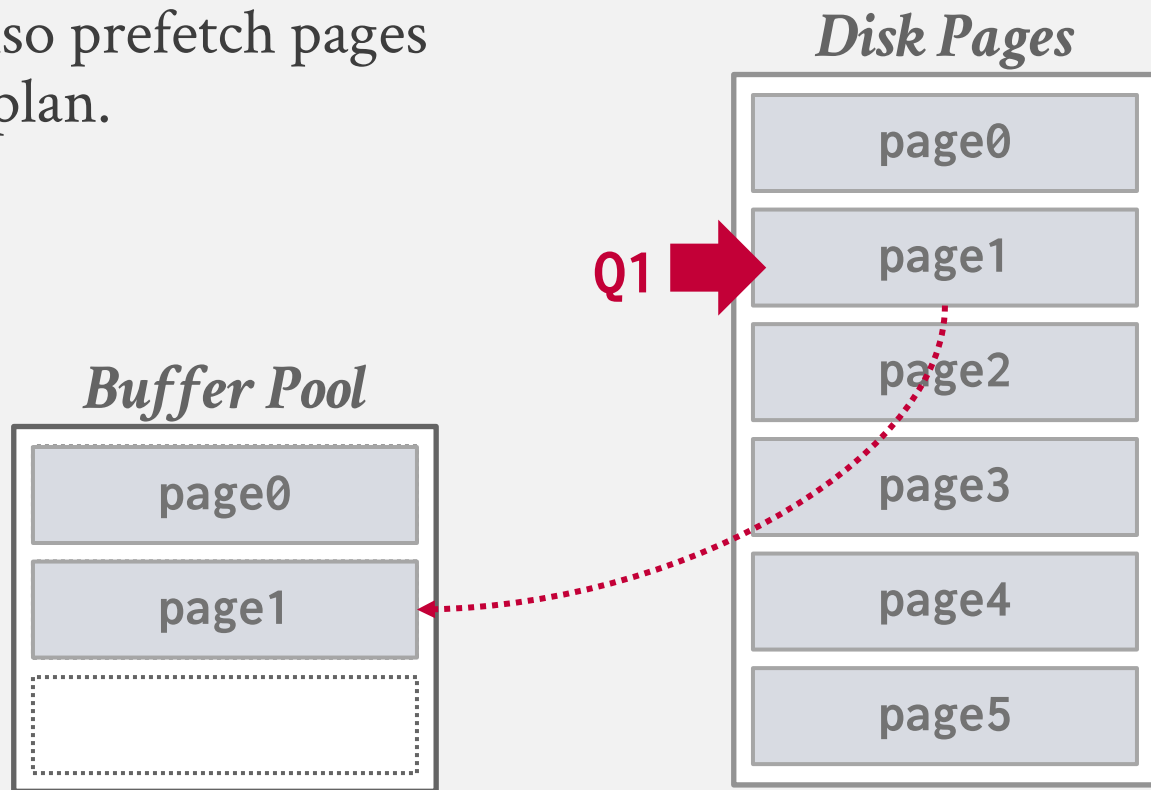


PRE-FETCHING

The DBMS can also prefetch pages based on a query plan.

→ Sequential Scans

→ Index Scans

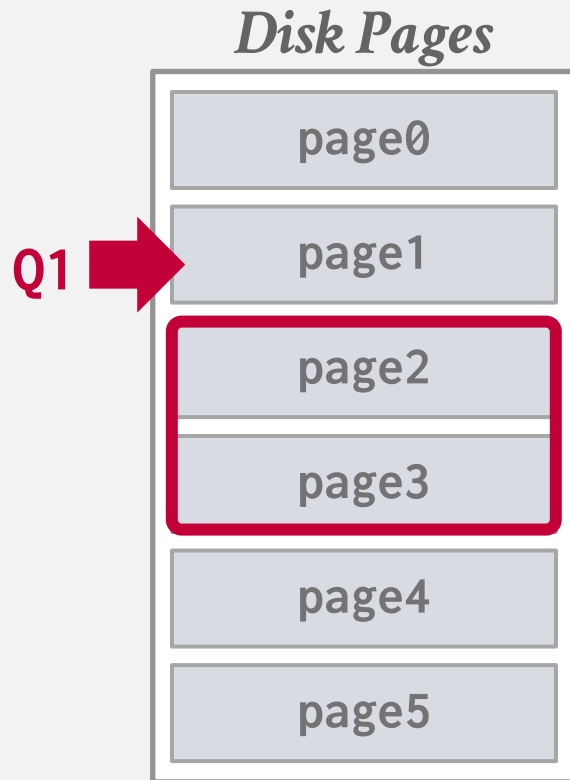
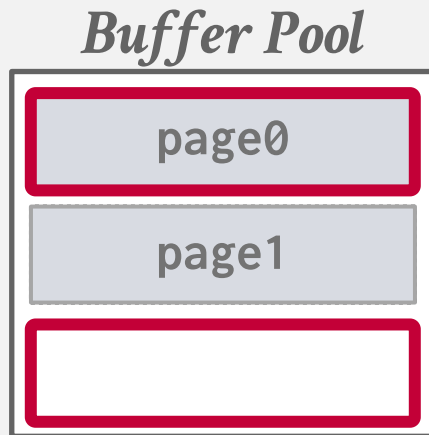


PRE-FETCHING

The DBMS can also prefetch pages based on a query plan.

→ Sequential Scans

→ Index Scans

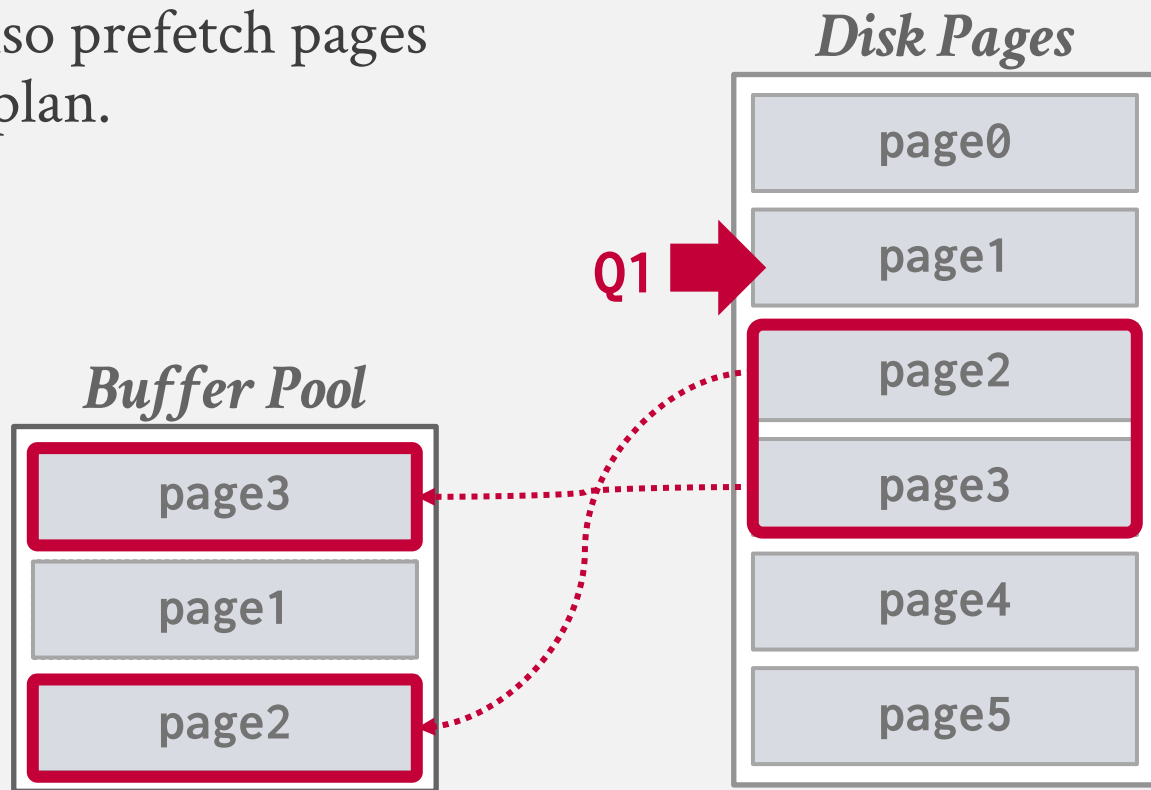


PRE-FETCHING

The DBMS can also prefetch pages based on a query plan.

→ Sequential Scans

→ Index Scans

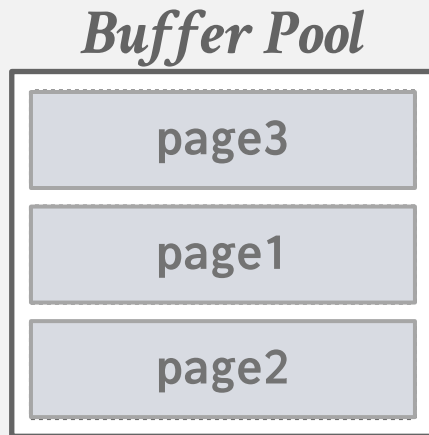


PRE-FETCHING

The DBMS can also prefetch pages based on a query plan.

→ Sequential Scans

→ Index Scans



PRE-FETCHING

The DBMS can also prefetch pages based on a query plan.

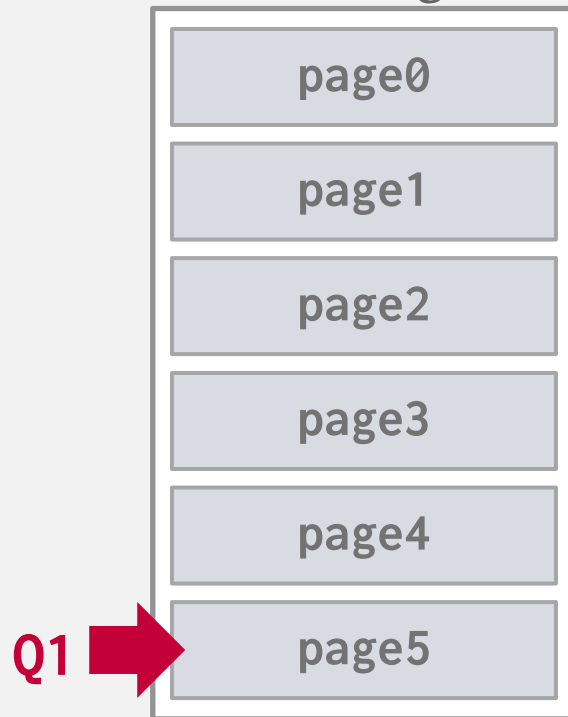
→ Sequential Scans

→ Index Scans

Buffer Pool



Disk Pages



PRE-FETCHING

Q1

```
SELECT * FROM A
WHERE val BETWEEN 100 AND 250
```

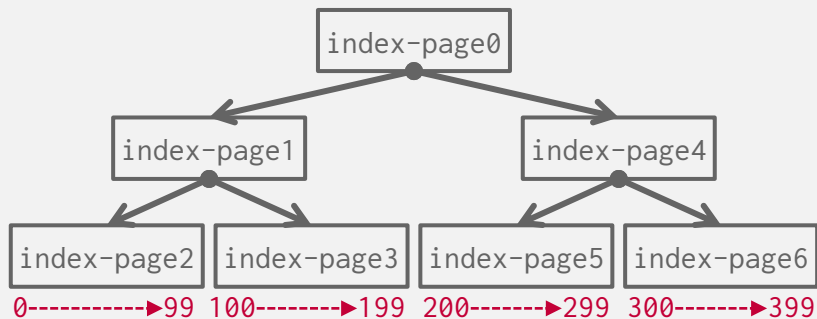
Buffer Pool



Disk Pages



PRE-FETCHING



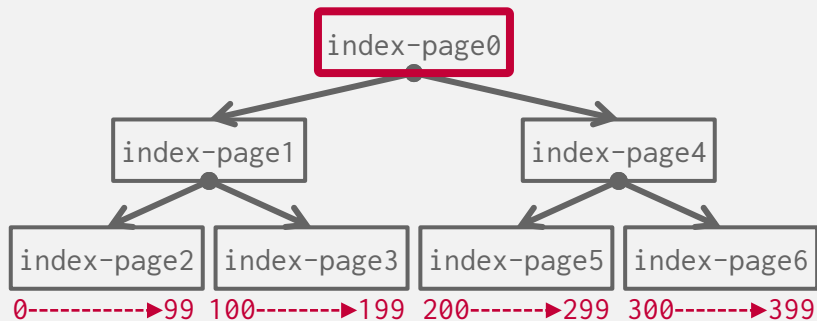
Buffer Pool



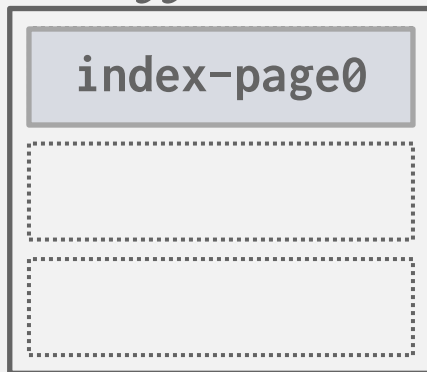
Disk Pages



PRE-FETCHING



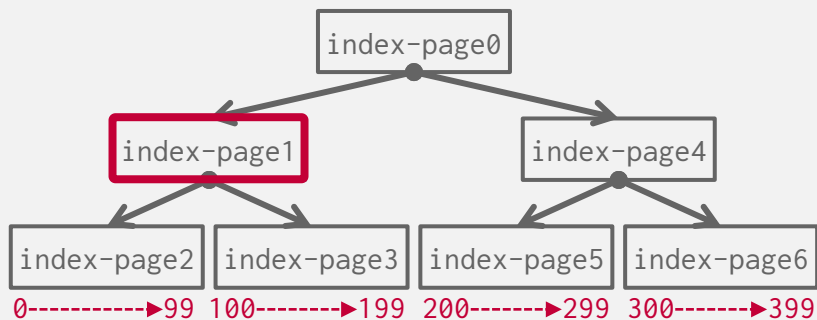
Buffer Pool



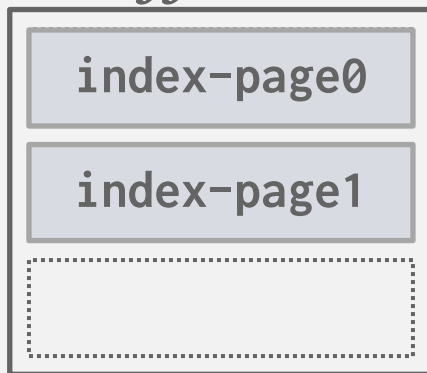
Disk Pages



PRE-FETCHING



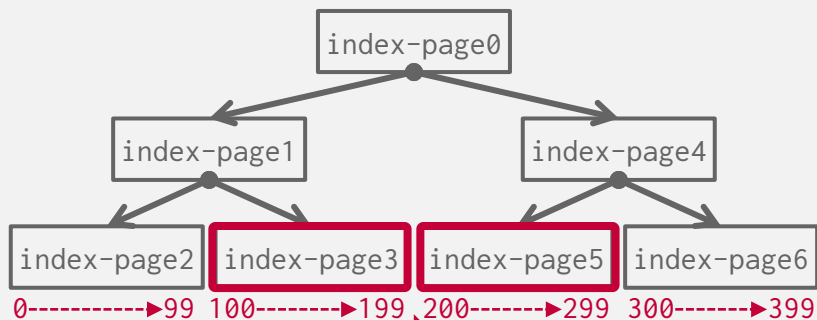
Buffer Pool



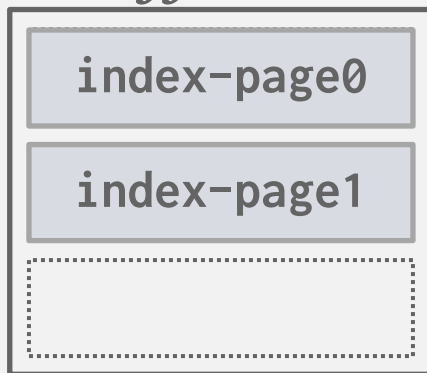
Disk Pages



PRE-FETCHING



Buffer Pool



Disk Pages



SCAN SHARING

Queries can reuse data retrieved from storage or operator computations.

- Also called *synchronized scans*.
- This is different from result caching.

Allow multiple queries to attach to a single cursor that scans a table.

- Queries do not have to be the same.
- Can also share intermediate results.

SCAN SHARING

If a query wants to scan a table and another query is already doing this, then the DBMS will attach the second query's cursor to the existing cursor.

Examples:

- Fully supported in DB2, MSSQL, Teradata, and Postgres.
- Oracle only supports cursor sharing for identical queries.

The logo for Teradata, featuring the word "TERADATA" in a bold, orange, sans-serif font.The logo for Microsoft SQL Server, featuring a red and white globe icon to the left of the text "Microsoft SQL Server" in a black, sans-serif font.The logo for IBM DB2, featuring the IBM logo (a black square with white horizontal stripes) to the left of the text "DB2" in a white, sans-serif font on a green background.The logo for Oracle, featuring the word "ORACLE" in a bold, red, sans-serif font.The logo for PostgreSQL, featuring a blue elephant icon to the left of the text "PostgreSQL" in a blue, sans-serif font.


SCAN SHARING

If a query wants to scan a table and another query is already doing this, then the DBMS will attach the second query's cursor to the existing cursor.

Examples:

For a textual match to occur, the text of the SQL statements or PL/SQL blocks must be character-for-character identical, including spaces, case, and comments. For example, the following statements cannot use the same shared SQL area:

```
SELECT * FROM employees;  
SELECT * FROM Employees;  
SELECT * FROM employees;
```

 Copy

reSQL

SCAN SHARING

Q1 SELECT SUM(val) FROM A

Q2 SELECT AVG(val) FROM A **LIMIT 100**

Buffer Pool



Disk Pages



CONTINUOUS SCAN SHARING

Instead of trying to be clever, the DBMS continuously scans the database files repeatedly.

- One continuous cursor per table.
- Queries "hop" on board the cursor while it is running and then disconnect once they have enough data.

Not viable if you pay per IOP.

Only done in academic prototypes.



Q1



BUFFER POOL BYPASS

The sequential scan operator will not store fetched pages in the buffer pool to avoid overhead.

- Memory is local to running query.
- Works well if operator needs to read a large sequence of pages that are contiguous on disk.
- Can also be used for temporary data (sorting, joins).

Called "Light Scans" in Informix.

ORACLE®

Microsoft®
SQL Server®

PostgreSQL

Informix®

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.

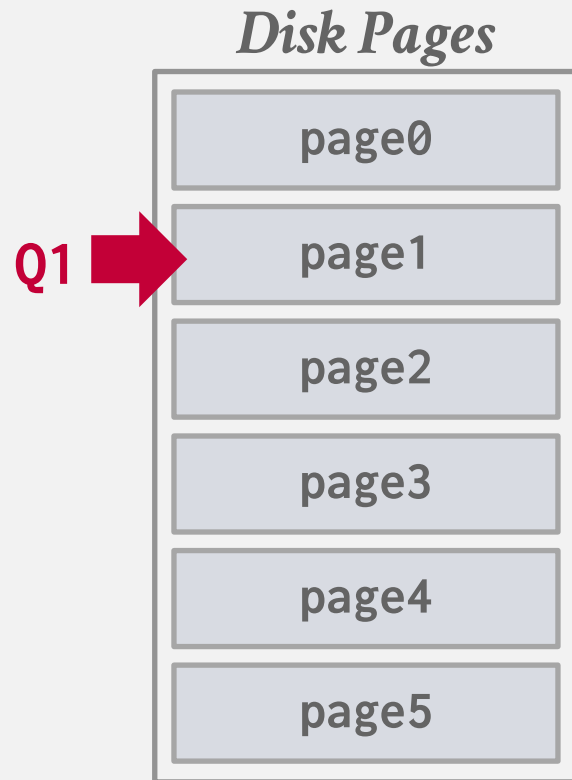
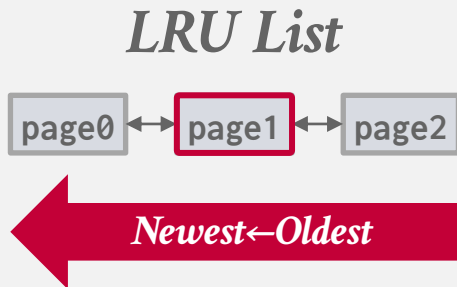
Goals:

- Correctness
- Accuracy
- Speed
- Meta-data overhead

LEAST-RECENTLY USED

Maintain a single timestamp of when each page was last accessed. When the DBMS needs to evict a page, select the one with the oldest timestamp.

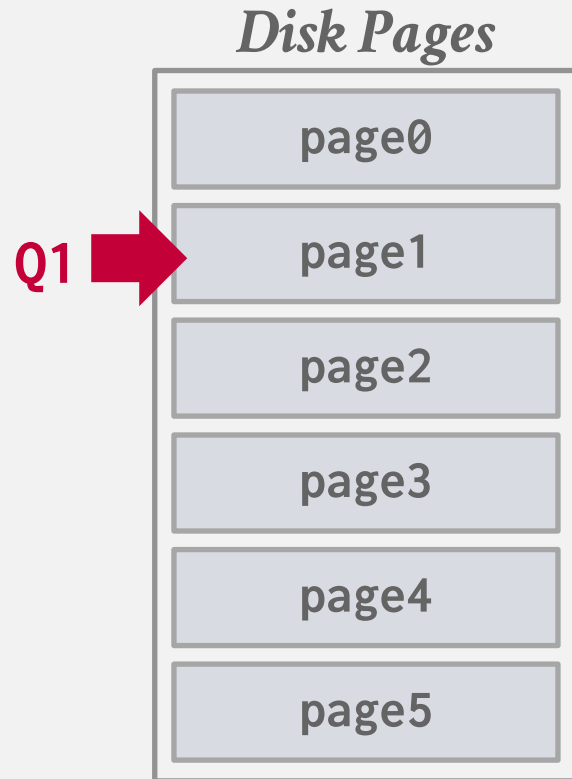
→ Keep the pages in sorted order to reduce the search time on eviction.



LEAST-RECENTLY USED

Maintain a single timestamp of when each page was last accessed. When the DBMS needs to evict a page, select the one with the oldest timestamp.

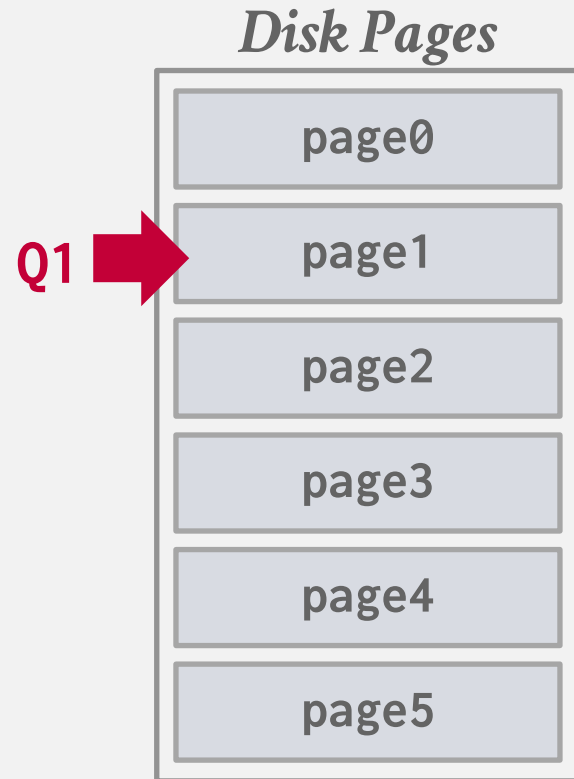
→ Keep the pages in sorted order to reduce the search time on eviction.



LEAST-RECENTLY USED

Maintain a single timestamp of when each page was last accessed. When the DBMS needs to evict a page, select the one with the oldest timestamp.

→ Keep the pages in sorted order to reduce the search time on eviction.



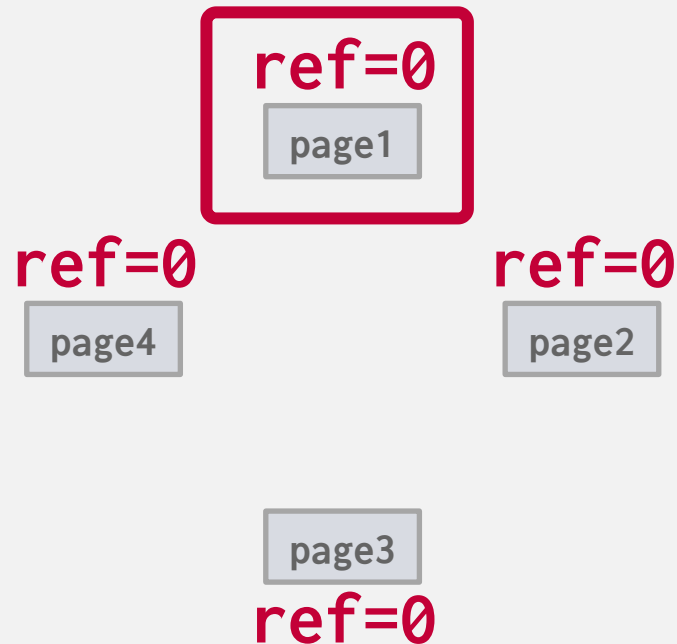
CLOCK

Approximation of LRU that does not need a separate timestamp per page.

- Each page has a reference bit.
- When a page is accessed, set to 1.

Organize the pages in a circular buffer with a "clock hand":

- Upon sweeping, check if a page's bit is set to 1.
- If yes, set to zero. If no, then evict.



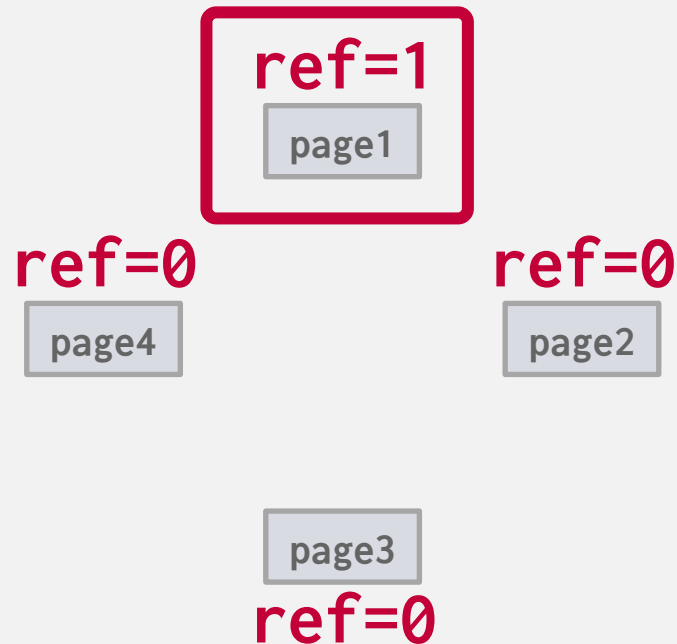
CLOCK

Approximation of LRU that does not need a separate timestamp per page.

- Each page has a reference bit.
- When a page is accessed, set to 1.

Organize the pages in a circular buffer with a "clock hand":

- Upon sweeping, check if a page's bit is set to 1.
- If yes, set to zero. If no, then evict.



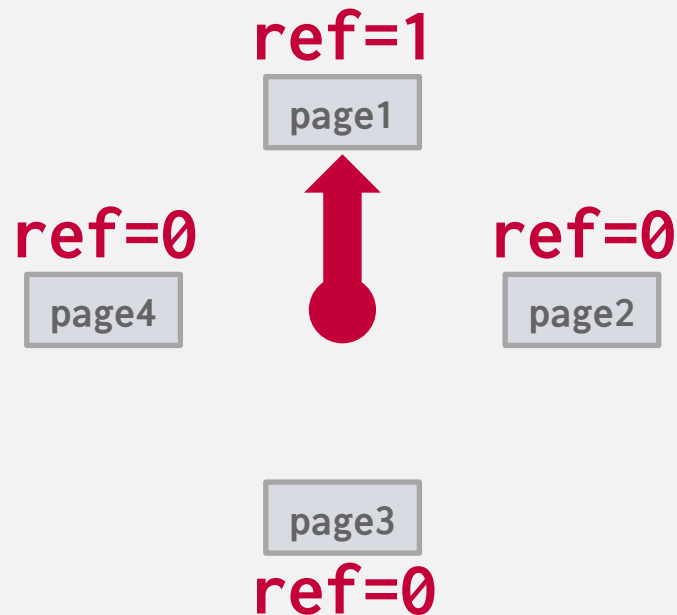
CLOCK

Approximation of LRU that does not need a separate timestamp per page.

- Each page has a reference bit.
- When a page is accessed, set to 1.

Organize the pages in a circular buffer with a "clock hand":

- Upon sweeping, check if a page's bit is set to 1.
- If yes, set to zero. If no, then evict.



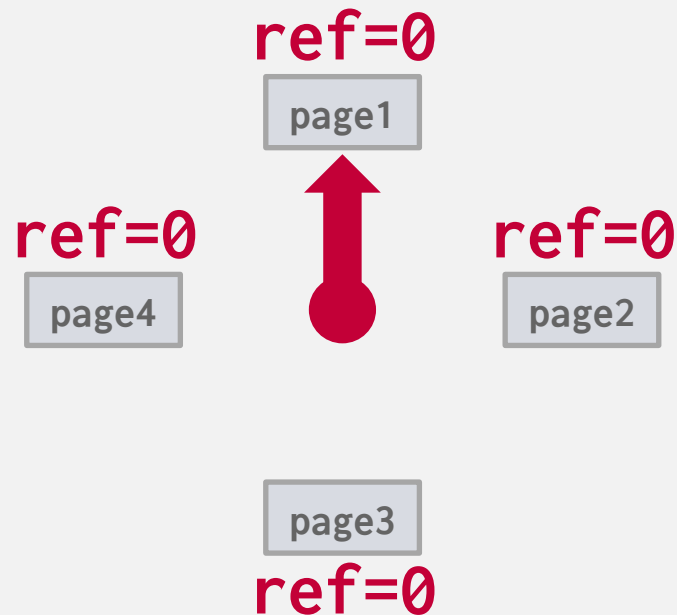
CLOCK

Approximation of LRU that does not need a separate timestamp per page.

- Each page has a reference bit.
- When a page is accessed, set to 1.

Organize the pages in a circular buffer with a "clock hand":

- Upon sweeping, check if a page's bit is set to 1.
- If yes, set to zero. If no, then evict.



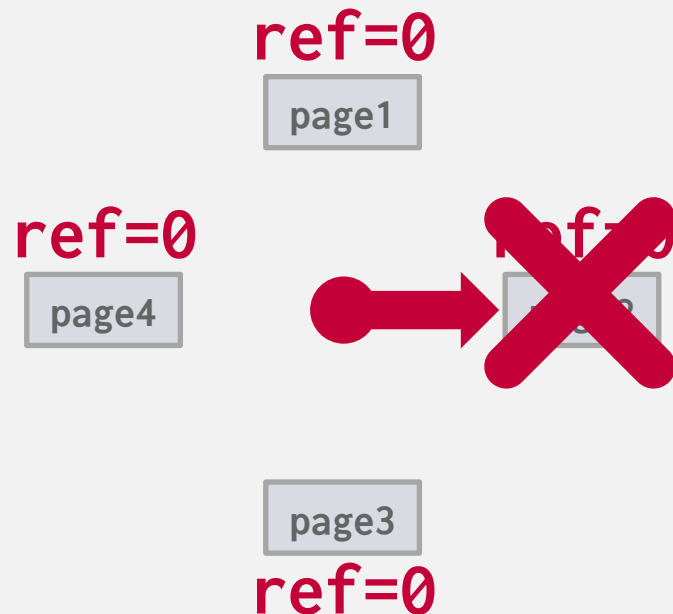
CLOCK

Approximation of LRU that does not need a separate timestamp per page.

- Each page has a reference bit.
- When a page is accessed, set to 1.

Organize the pages in a circular buffer with a "clock hand":

- Upon sweeping, check if a page's bit is set to 1.
- If yes, set to zero. If no, then evict.



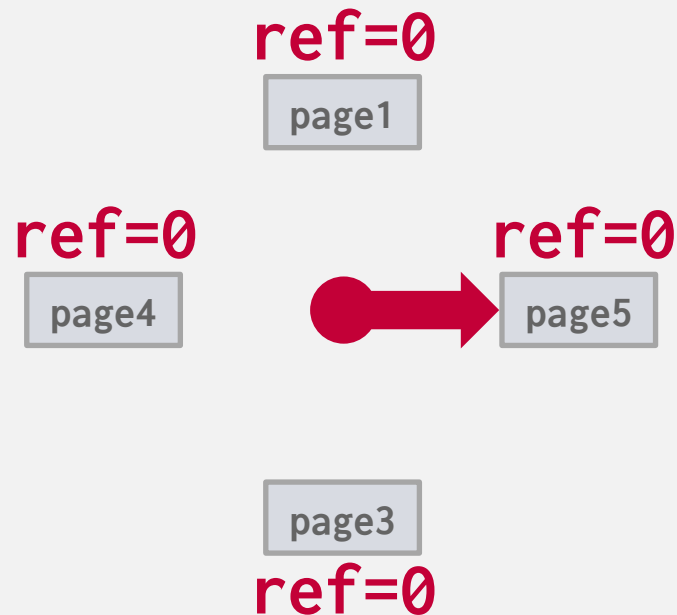
CLOCK

Approximation of LRU that does not need a separate timestamp per page.

- Each page has a reference bit.
- When a page is accessed, set to 1.

Organize the pages in a circular buffer with a "clock hand":

- Upon sweeping, check if a page's bit is set to 1.
- If yes, set to zero. If no, then evict.



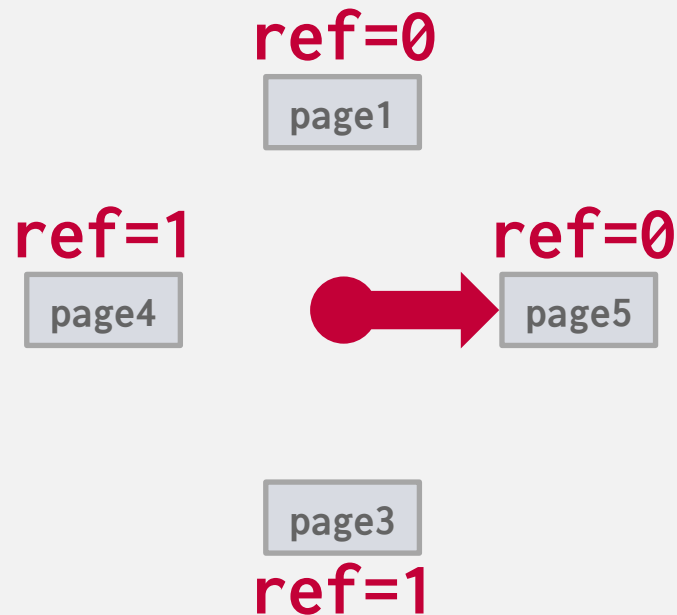
CLOCK

Approximation of LRU that does not need a separate timestamp per page.

- Each page has a reference bit.
- When a page is accessed, set to 1.

Organize the pages in a circular buffer with a "clock hand":

- Upon sweeping, check if a page's bit is set to 1.
- If yes, set to zero. If no, then evict.



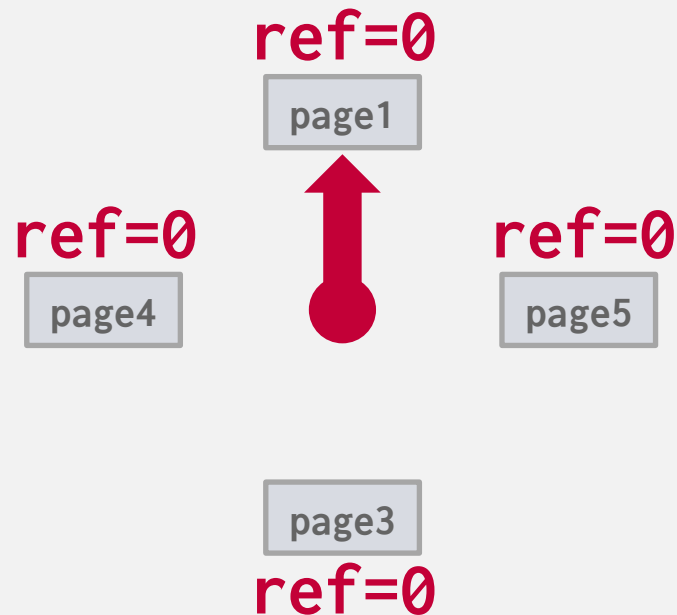
CLOCK

Approximation of LRU that does not need a separate timestamp per page.

- Each page has a reference bit.
- When a page is accessed, set to 1.

Organize the pages in a circular buffer with a "clock hand":

- Upon sweeping, check if a page's bit is set to 1.
- If yes, set to zero. If no, then evict.



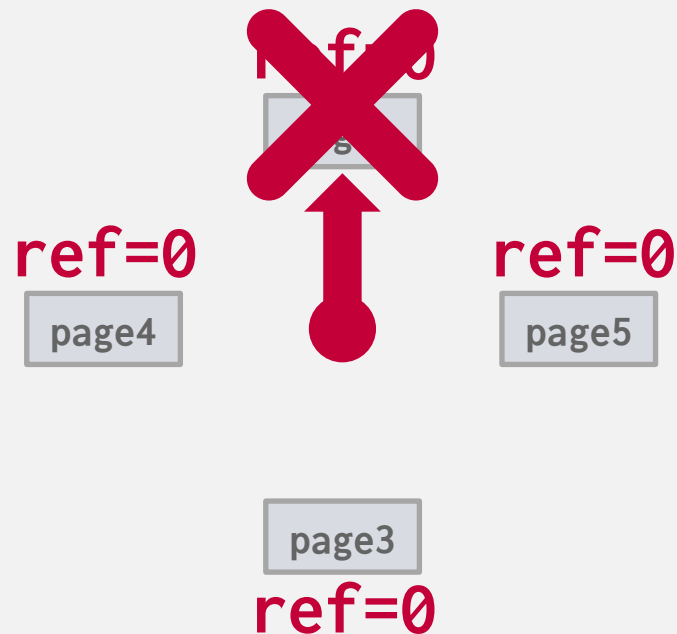
CLOCK

Approximation of LRU that does not need a separate timestamp per page.

- Each page has a reference bit.
- When a page is accessed, set to 1.

Organize the pages in a circular buffer with a "clock hand":

- Upon sweeping, check if a page's bit is set to 1.
- If yes, set to zero. If no, then evict.



OBSERVATION

LRU + CLOCK replacement policies are susceptible to sequential flooding.

- A query performs a sequential scan that reads every page.
- This pollutes the buffer pool with pages that are read once and then never again.
- In OLAP workloads, the *most recently used* page is often the best page to evict.

LRU + CLOCK only tracks when a page was last accessed, but not how often a page is accessed.

SEQUENTIAL FLOODING

Q1 SELECT * FROM A WHERE id = 1

Q2 SELECT AVG(val) FROM A

Q3 SELECT * FROM A WHERE id = 1

Buffer Pool



Disk Pages



BETTER POLICIES: LRU-K

Track the history of last K references to each page as timestamps and compute the interval between subsequent accesses.

→ Can get fancy with distinguishing between reference type

The DBMS then uses this history to estimate the next time that page is going to be accessed.

→ Evict pages with the longest expected intervals.

→ Maintain an ephemeral in-memory cache for recently evicted pages to prevent them from always being evicted.



MYSQL APPROXIMATE LRU-K

Single LRU linked list but with two entry points ("old" vs "young").

- New pages are always inserted to the head of the old list.
- If pages in the old list is accessed again, then insert into the head of the young list.



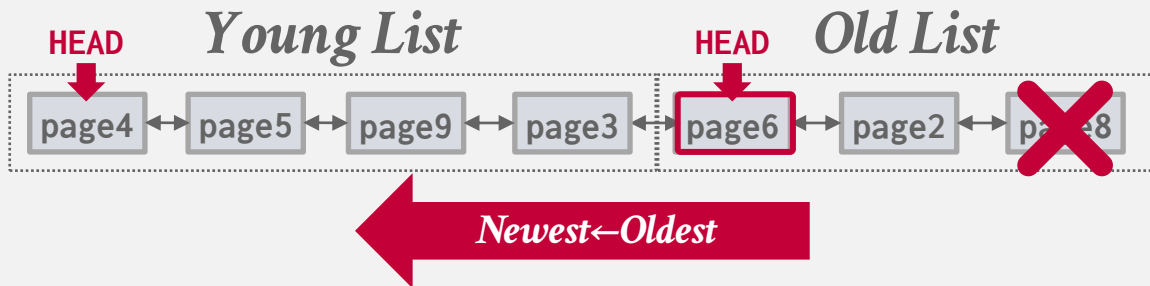
Disk Pages



MYSQL APPROXIMATE LRU-K

Single LRU linked list but with two entry points ("old" vs "young").

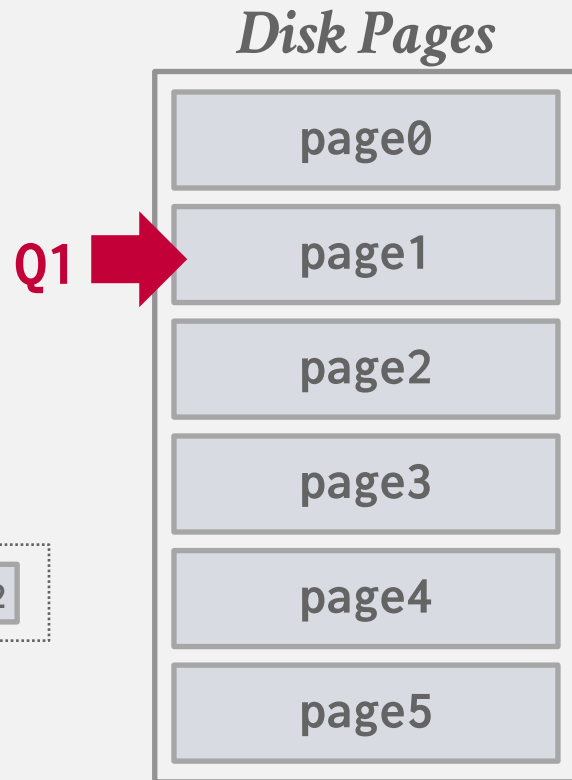
- New pages are always inserted to the head of the old list.
- If pages in the old list is accessed again, then insert into the head of the young list.



MYSQL APPROXIMATE LRU-K

Single LRU linked list but with two entry points ("old" vs "young").

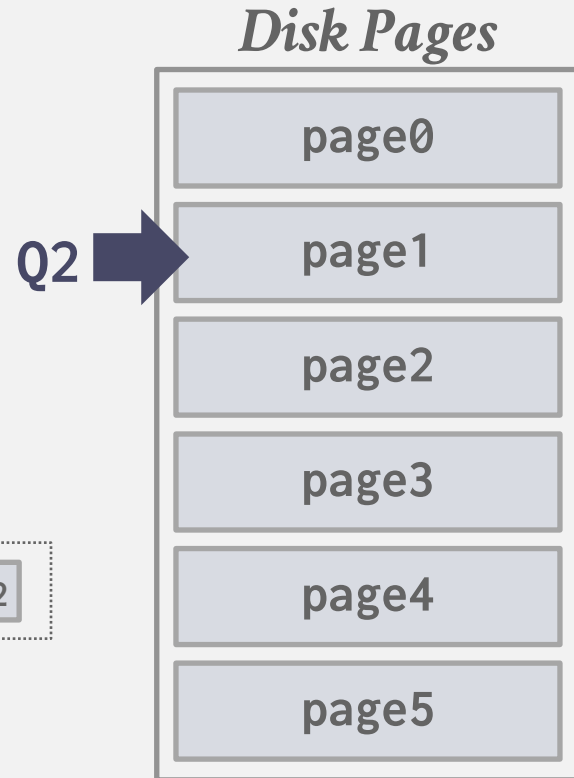
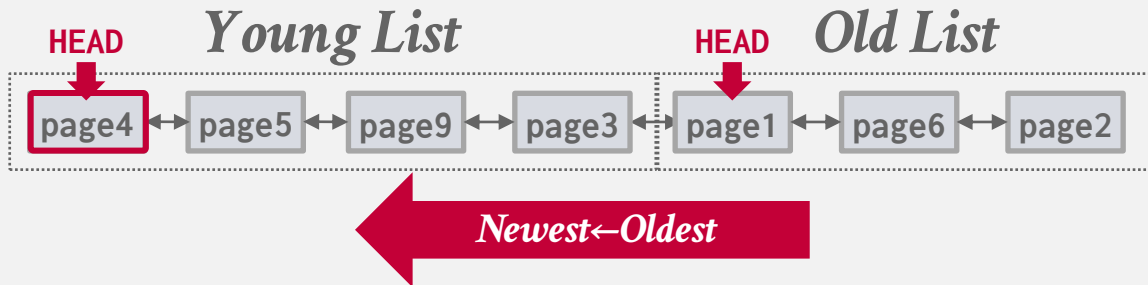
- New pages are always inserted to the head of the old list.
- If pages in the old list is accessed again, then insert into the head of the young list.



MYSQL APPROXIMATE LRU-K

Single LRU linked list but with two entry points ("old" vs "young").

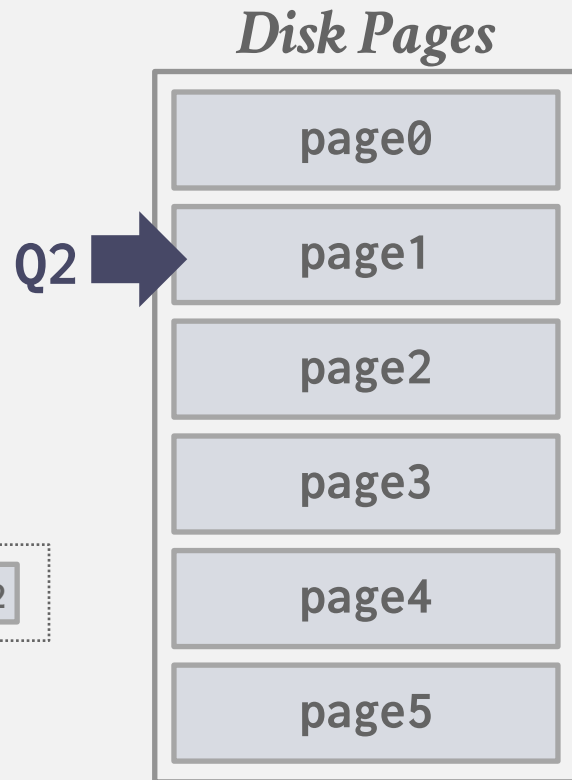
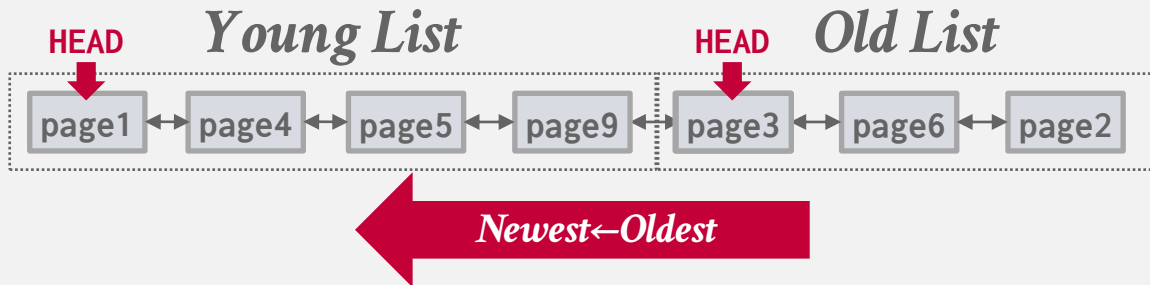
- New pages are always inserted to the head of the old list.
- If pages in the old list is accessed again, then insert into the head of the young list.



MYSQL APPROXIMATE LRU-K

Single LRU linked list but with two entry points ("old" vs "young").

- New pages are always inserted to the head of the old list.
- If pages in the old list is accessed again, then insert into the head of the young list.



BETTER POLICIES: LOCALIZATION

The DBMS chooses which pages to evict on a per query basis. This minimizes the pollution of the buffer pool from each query.

→ Keep track of the pages that a query has accessed.

Example: Postgres maintains a small ring buffer that is private to the query.

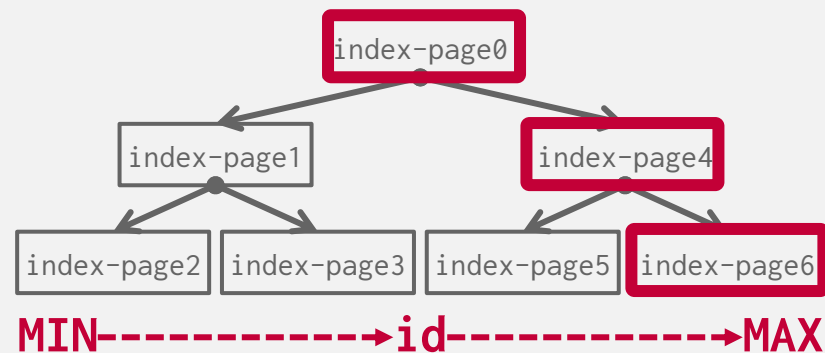
BETTER POLICIES: PRIORITY HINTS

The DBMS knows about the context of each page during query execution.

It can provide hints to the buffer pool on whether a page is important or not.

Q1 INSERT INTO A VALUES (*id++*)

Q2 SELECT * FROM A WHERE id = ?



DIRTY PAGES

Fast Path: If a page in the buffer pool is not dirty, then the DBMS can simply "drop" it.

Slow Path: If a page is dirty, then the DBMS must write back to disk to ensure that its changes are persisted.

Trade-off between fast evictions versus dirty writing pages that will not be read again in the future.

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.

Need to be careful that the system doesn't write dirty pages before their log records are written...

OBSERVATION

OS/hardware tries 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.

Many DBMSs tell you to switch Linux to use the deadline or noop (FIFO) scheduler.

→ Example: Oracle, Vertica, MySQL

DISK I/O SCHEDULING

The DBMS maintain internal queue(s) to track page read/write requests from the entire system.

Compute priorities 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

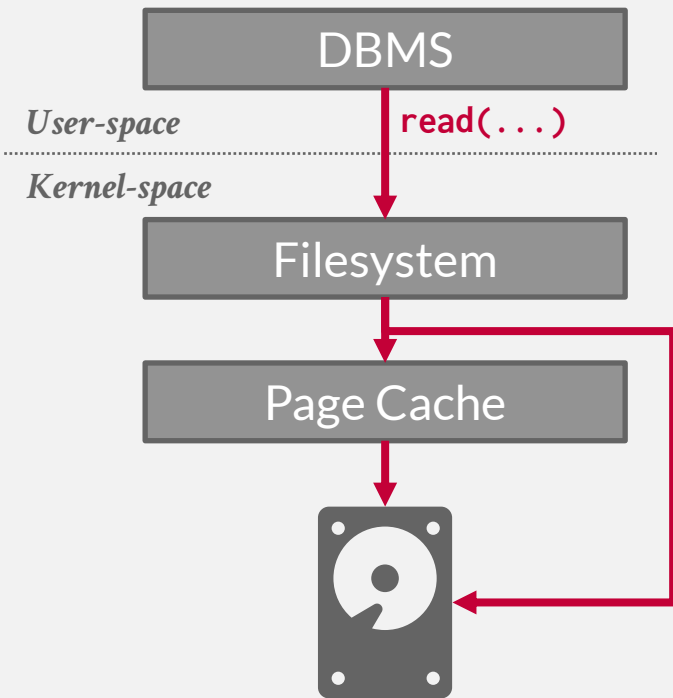
The OS doesn't know these things and is going to get into the way...

OS PAGE CACHE

Most disk operations go through the OS API. Unless the DBMS tells it not to, the OS maintains its own filesystem cache (aka page cache, buffer cache).

Most DBMSs use direct I/O (**O_DIRECT**) to bypass the OS's cache.

- Redundant copies of pages.
- Different eviction policies.
- Loss of control over file I/O.



FSYNC PROBLEMS

If the DBMS calls **fw**

If the DBMS calls **fs**

If **fsync** fails (EIO),
 → Linux marks the dirt
 → If the DBMS calls **fs**
 the flush was success

The screenshot shows a web page titled "Fsync Errors" with a navigation sidebar on the left and a main content area on the right. The sidebar includes a navigation menu with links for Main Page, Random page, Recent changes, and Help. It also has a tools section with links for What links here, Related changes, Special pages, Printable version, Permanent link, and Page information. A search box is located at the bottom of the sidebar. The main content area has a title "Fsync Errors" and a sub-header "Contents [hide]" with a list of sections: 1 Current status, 2 Articles and news, and 3 Research notes and OS differences (with sub-sections 3.1 Open source kernels, 3.2 Closed source kernels, 3.3 Special cases, and 3.4 History and notes). The "Current status" section contains text about PostgreSQL 12 commit, Linux kernel 4.13 improvements, and a list of significant 4.13 commits including changes to writeback error handling and reporting. The "Articles and news" section lists several external links related to the fsyncgate 2018 mailing list and LWN.net articles.

page discussion view source history

Fsync Errors

This article covers the current status, history, and OS and OS version differences relating to the circa 2018 fsync() reliability issues discussed on the PostgreSQL mailing list and elsewhere. It has sometimes been referred to as "fsyncgate 2018".

Contents [hide]

- 1 Current status
- 2 Articles and news
- 3 Research notes and OS differences
 - 3.1 Open source kernels
 - 3.2 Closed source kernels
 - 3.3 Special cases
 - 3.4 History and notes

Current status

As of [this PostgreSQL 12 commit](#), PostgreSQL will now PANIC on fsync() failure. It was backpatched to PostgreSQL 11, 10, 9.6, 9.5 and 9.4. Thanks to Thomas Munro, Andres Freund, Robert Haas, and Craig Ringer.

Linux kernel 4.13 improved fsync() error handling and the [man page for fsync\(\)](#) is somewhat improved as well. See:

- Kernelnewbies for 4.13
- Particularly significant 4.13 commits include:
 - "fs: new infrastructure for writeback error handling and reporting"
 - "ext4: use erseq_t based error handling for reporting data writeback errors"
 - "Documentation: flesh out the section in vfs.txt on storing and reporting writeback errors"
 - "mm: set both AS_EIO/AS_ENOSPC and erseq_t in mapping_set_error"

Many thanks to Jeff Layton for work done in this area.

Similar changes were made in [InnoDB/MySQL](#), [WiredTiger/MongoDB](#) and no doubt other software as a result of the PR around this.

A proposed follow-up change to PostgreSQL was discussed in the thread [Refactoring the checkpoint's fsync request queue](#). The [patch that was committed](#) did not incorporate the file-descriptor passing changes proposed. There is still discussion open on some additional safeguards that may use file system error counters and/or filesystem-wide flushing.

Articles and news

- The "fsyncgate 2018" mailing list thread
- LWN.net article "PostgreSQL's fsync() surprise"
- LWN.net article "Improved block-layer error handling"

OTHER MEMORY POOLS

The DBMS needs memory for things other than just tuples and indexes.

These other memory pools may not always be backed by disk. Depends on implementation.

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

CONCLUSION

The DBMS can almost always manage memory better than the OS.

Leverage the semantics about the query plan to make better decisions:

- Evictions
- Allocations
- Pre-fetching

NEXT CLASS

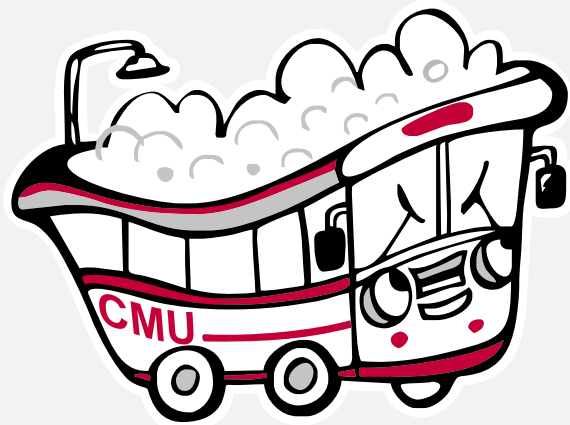
Hash Tables

PROJECT #1

You will build the first component of your storage manager.

- LRU-K Replacement Policy
- Disk Scheduler
- Buffer Pool Manager Instance

We will provide you with the basic APIs for these components.



BusTub

Due Date:
Sunday Oct 1st @ 11:59pm

TASK #1 - LRU-K REPLACEMENT POLICY

Build a data structure that tracks the usage of pages using the LRU-K policy.

General Hints:

- Your **LRUKReplacer** needs to check the "pinned" status of a **Page**.
- If there are no pages touched since last sweep, then return the lowest page id.

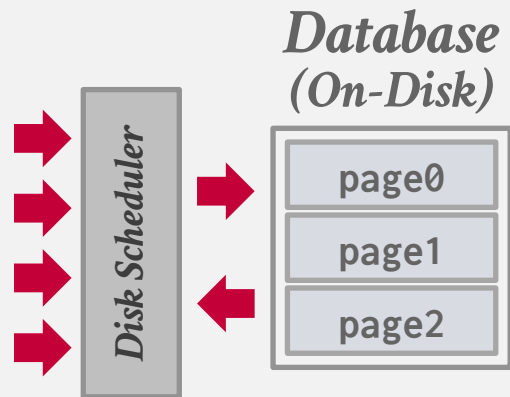
TASK #2 - DISK SCHEDULER

Create a background worker to read/write pages from disk.

- Single request queue.
- Simulates asynchronous IO using **`std::promise`** for callbacks.

It's up to you to decide how you want to batch, reorder, and issue read/write requests to the local disk.

Make sure it is thread-safe!

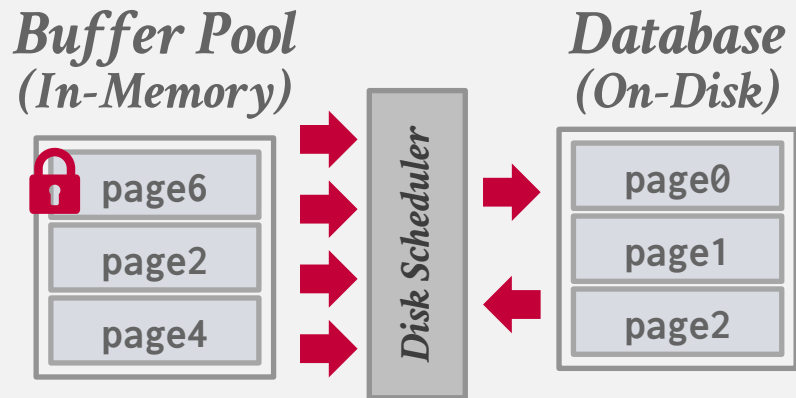


TASK #3 - BUFFER POOL MANAGER

Use your LRU-K replacer to manage the allocation of pages.

- Need to maintain internal data structures to track allocated + free pages.
- Use whatever data structure you want for the page table.

Make sure you get the order of operations correct when pinning!



THINGS TO NOTE

Do **not** change any file other than the six that you must hand in. Other changes will not be graded.

The projects are cumulative.

We will **not** be providing solutions.

Post any questions on Piazza or come to office hours, but we will **not** help you debug.

CODE QUALITY

We will automatically check whether you are writing good code.

→ [Google C++ Style Guide](#)

→ [Doxygen Javadoc Style](#)

You need to run these targets before you submit your implementation to Gradescope.

→ **make format**

→ **make check-clang-tidy-p1**

EXTRA CREDIT

Gradescope Leaderboard runs your code with a specialized in-memory version of BusTub.

The top 20 fastest implementations in the class will receive extra credit for this assignment.

- **#1**: 50% bonus points
- **#2–10**: 25% bonus points
- **#11–20**: 10% bonus points

Student with the most bonus points at the end of the semester will receive a BusTub hoodie!



PLAGIARISM WARNING



The homework and projects must be your own original work. They are **not** group assignments. You may **not** copy source code from other people or the web.

Plagiarism is **not** tolerated. You will get lit up.
→ Please ask me if you are unsure.

See [CMU's Policy on Academic Integrity](#) for additional information.