

Carnegie  
Mellon  
University

Intro to Database  
Systems (15-445/645)

Lecture #06

# Memory & Disk I/O Management

SPRING 2024 >> Prof. Jignesh Patel



# ADMINISTRIVIA

---

**Project #1** is due February 18<sup>th</sup> @ 11:59pm.

**Homework #2** will be posted in an hour. It is due February 16<sup>th</sup> @ 11:59pm.

# 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

---

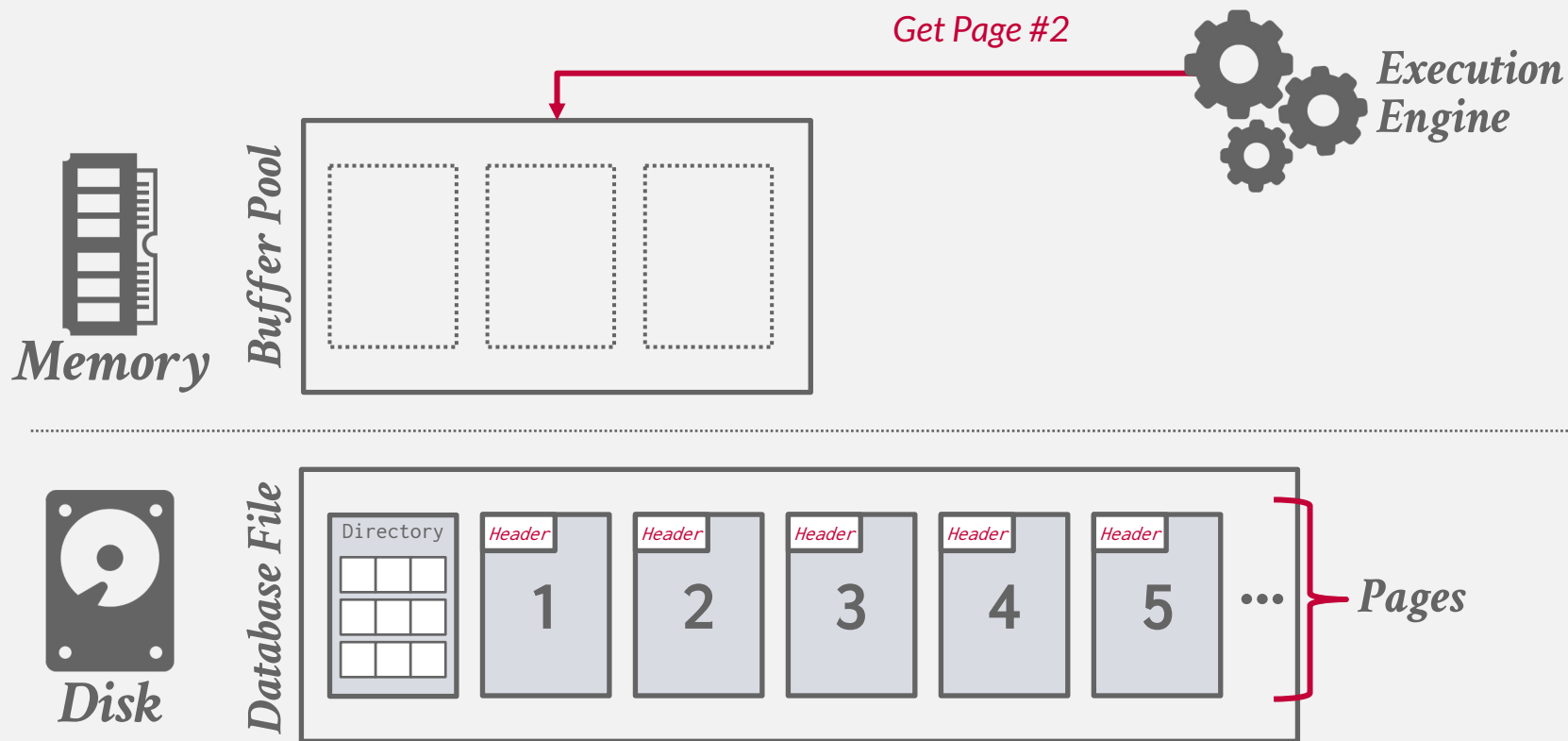
## **Spatial Control:**

- 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.

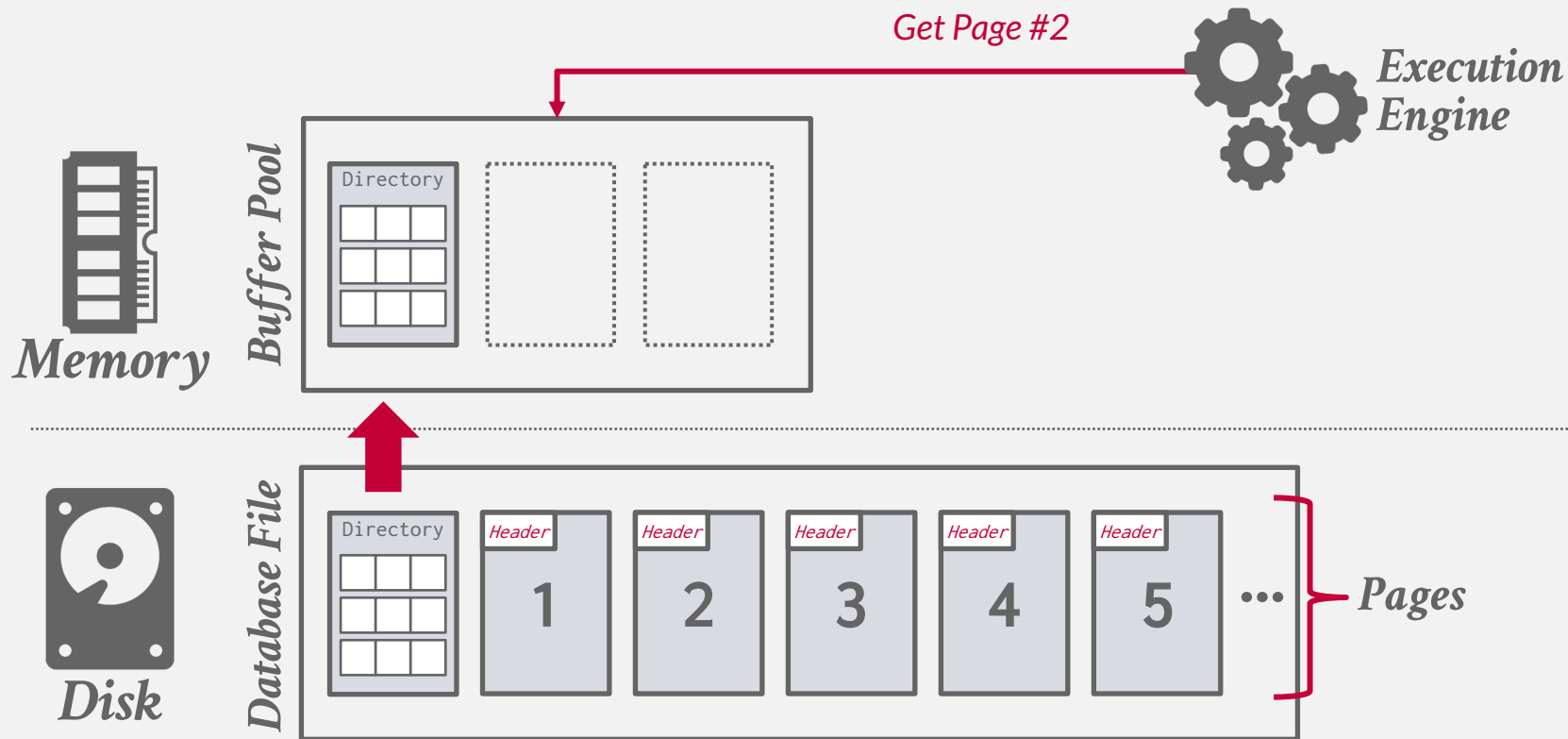
## **Temporal Control:**

- 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.

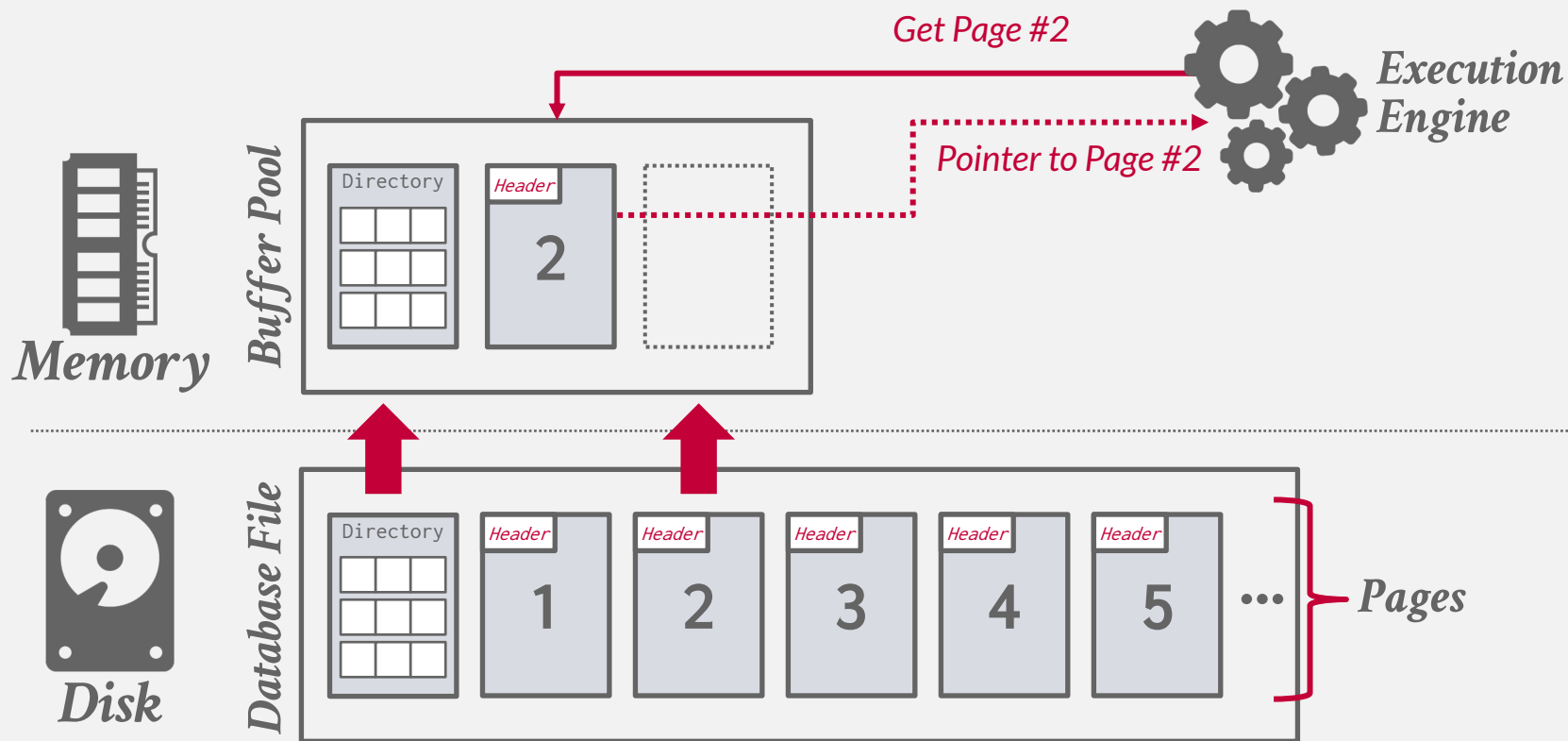
# DISK-ORIENTED DBMS



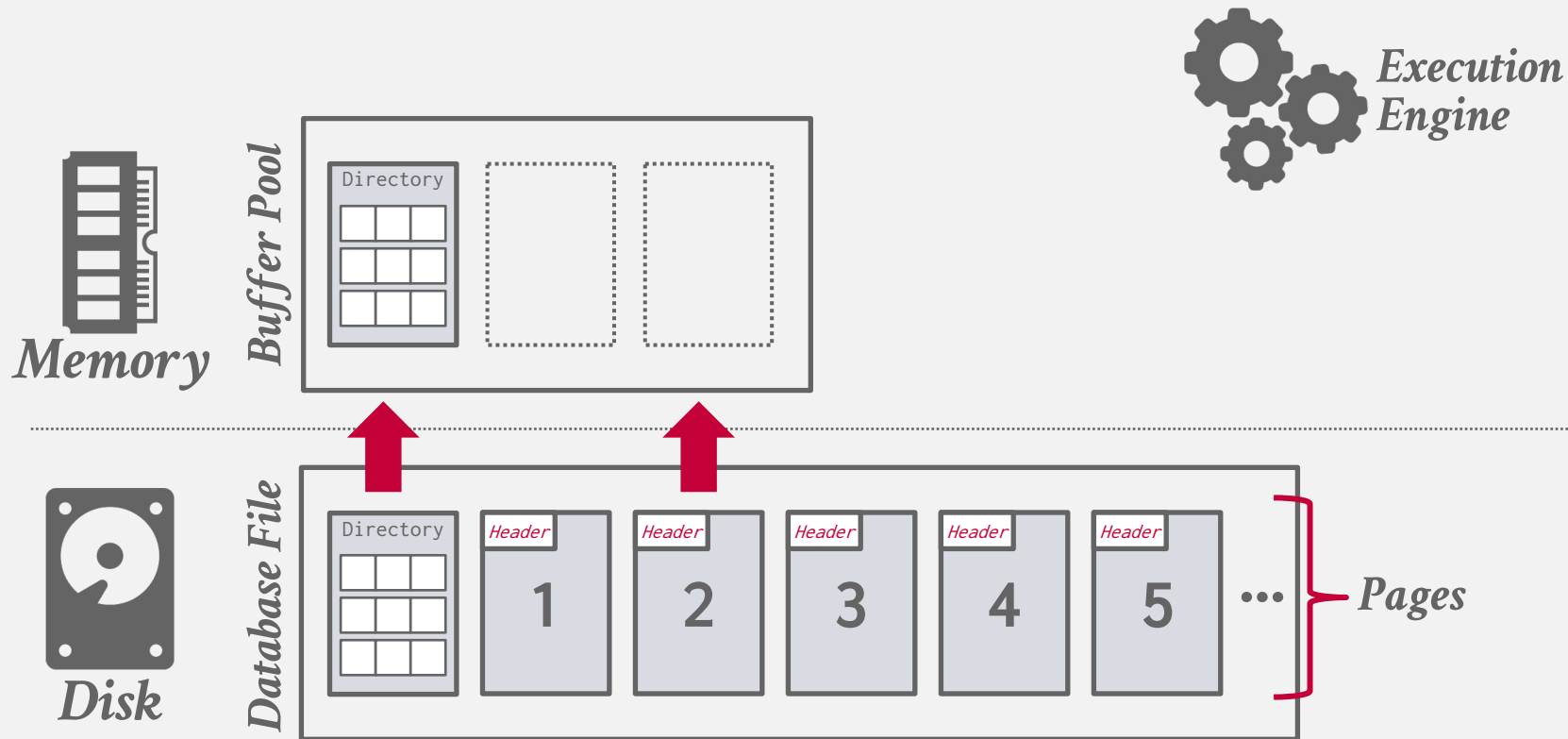
# DISK-ORIENTED DBMS



# DISK-ORIENTED DBMS

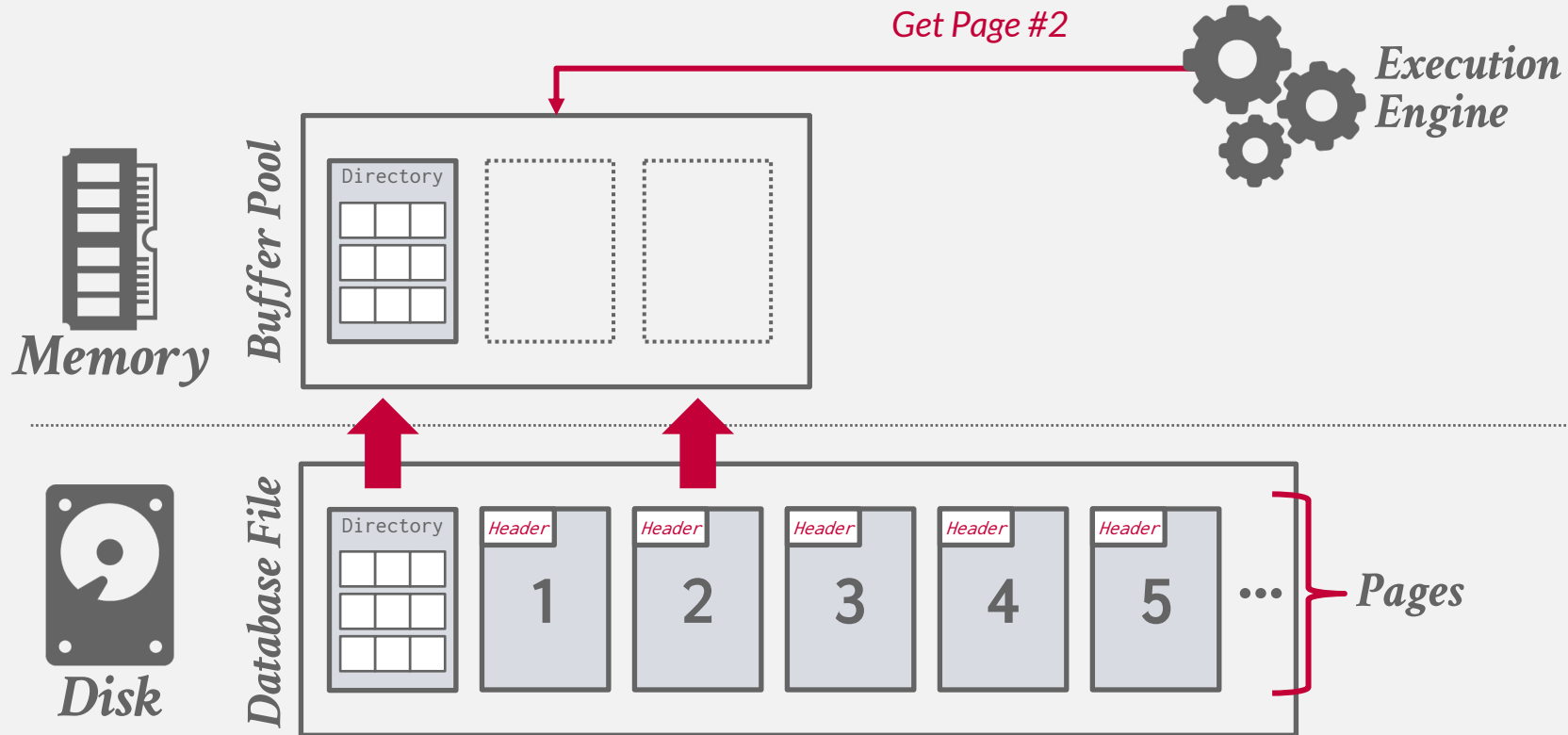


# DISK-ORIENTED DBMS

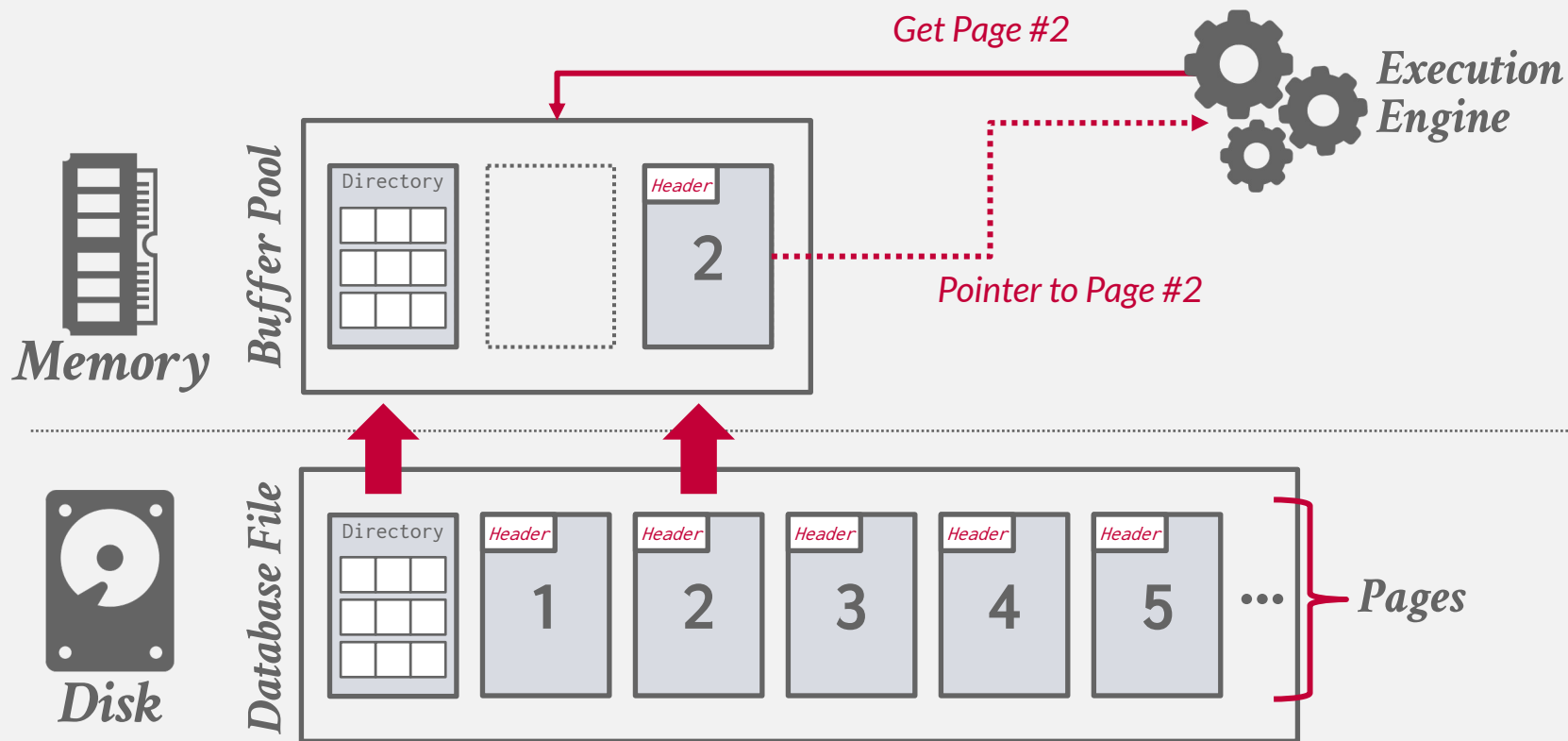




# DISK-ORIENTED DBMS



# DISK-ORIENTED DBMS



# TODAY'S AGENDA

---

Buffer Pool Manager

Disk I/O Scheduling

Replacement Policies

Other Memory Pools

# BUFFER POOL ORGANIZATION

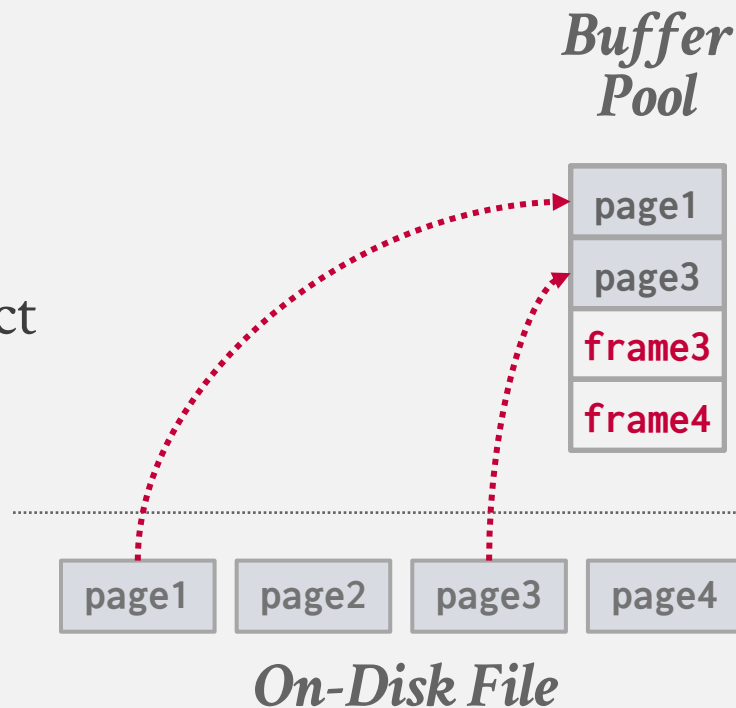
Memory region organized as an array of fixed-size pages.

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**

*Page Table*



*Buffer Pool*



*On-Disk File*

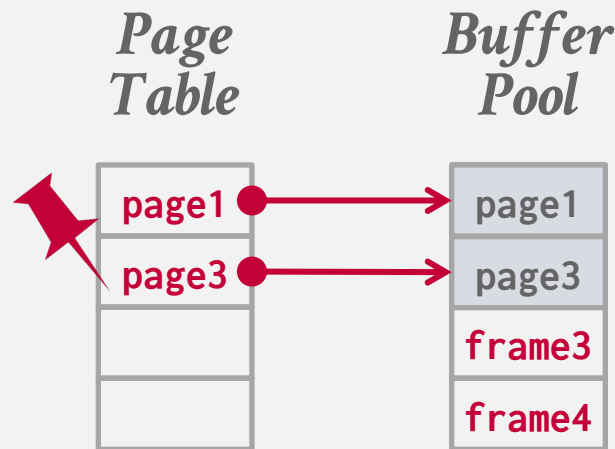
# BUFFER POOL META-DATA

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*

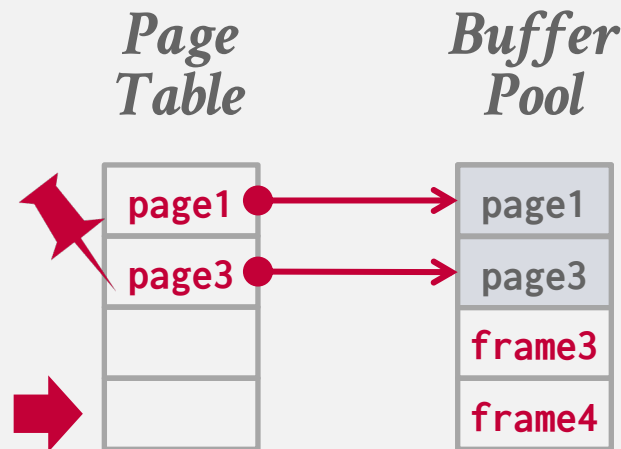
# BUFFER POOL META-DATA

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**



*On-Disk File*

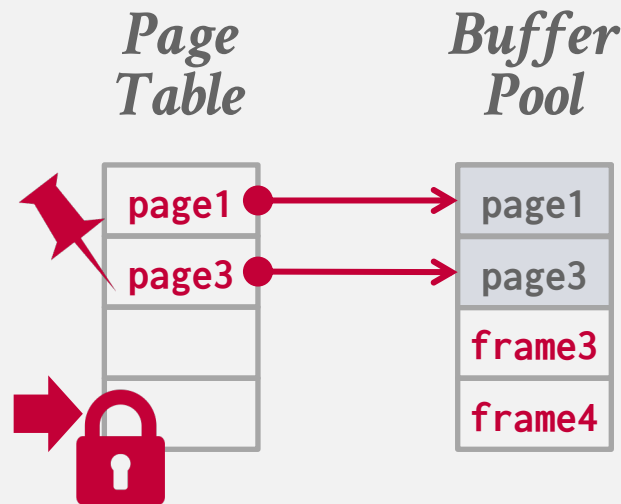
# BUFFER POOL META-DATA

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*



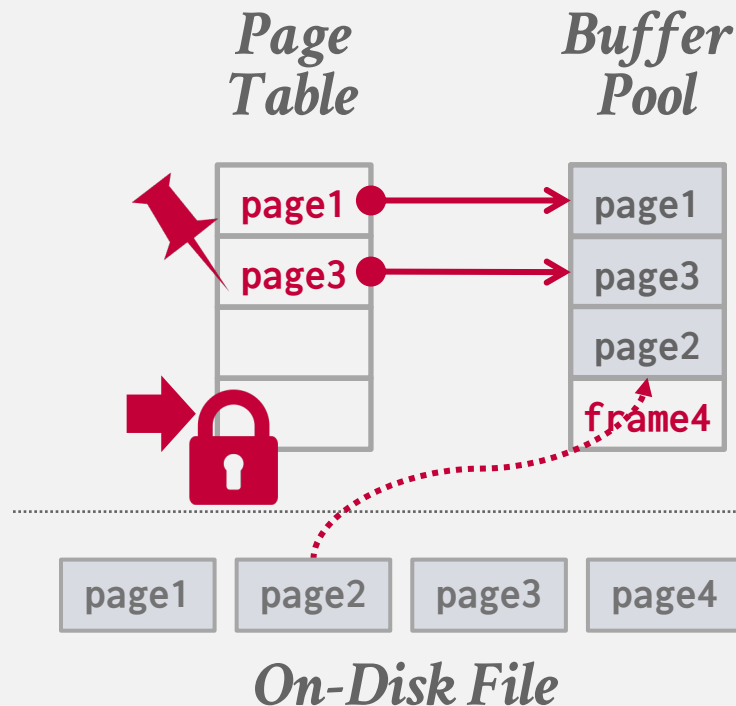
# BUFFER POOL META-DATA

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**



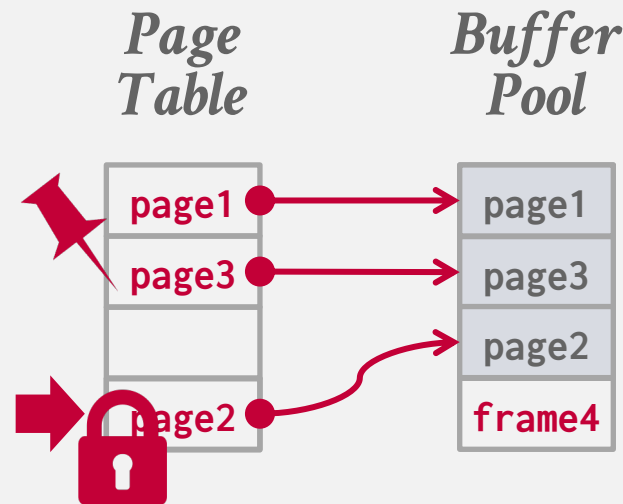
# BUFFER POOL META-DATA

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*

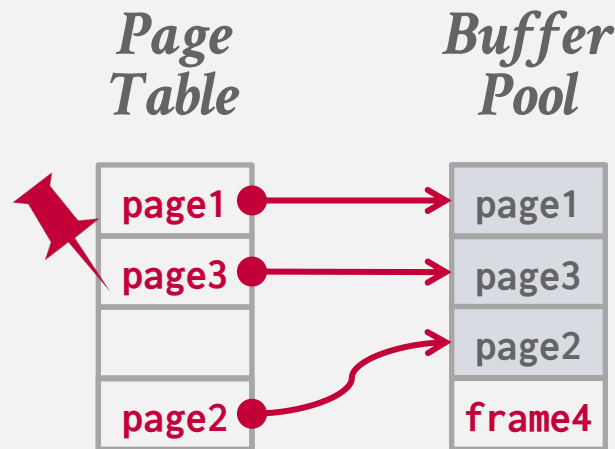
# BUFFER POOL META-DATA

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.

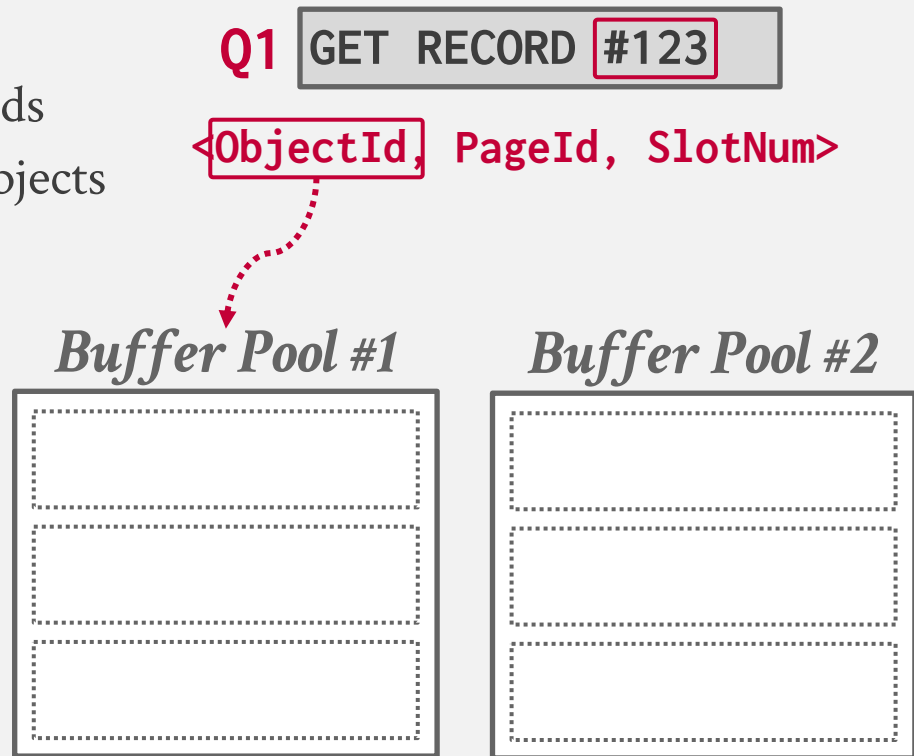




# MULTIPLE BUFFER POOLS

## 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.

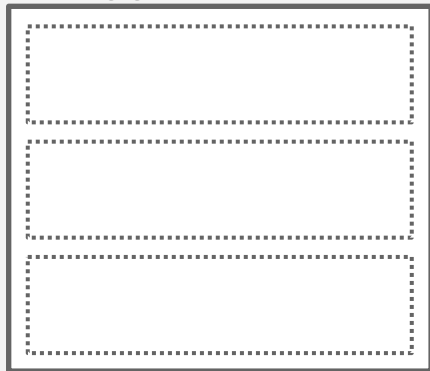
## Approach #2: Hashing

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

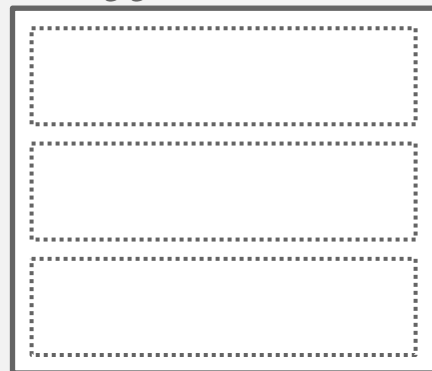
Q1 GET RECORD #123

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

*Buffer Pool #1*



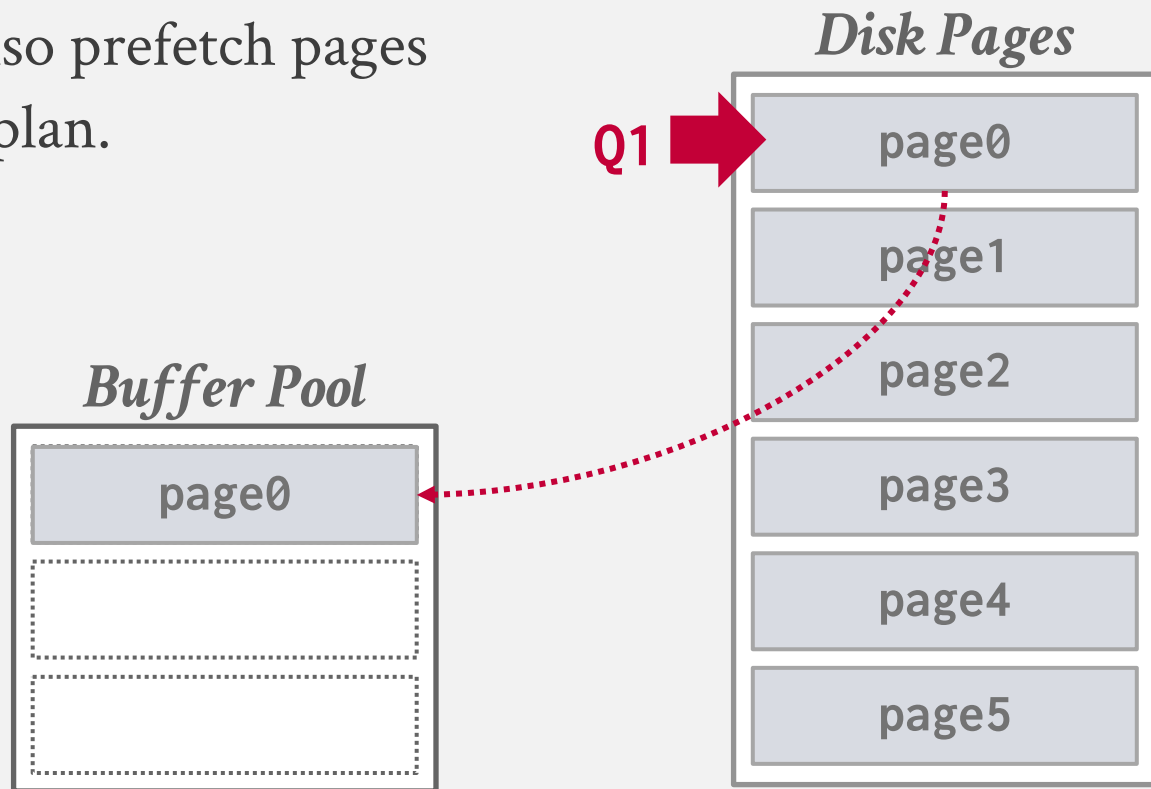
*Buffer Pool #2*



# PRE-FETCHING

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

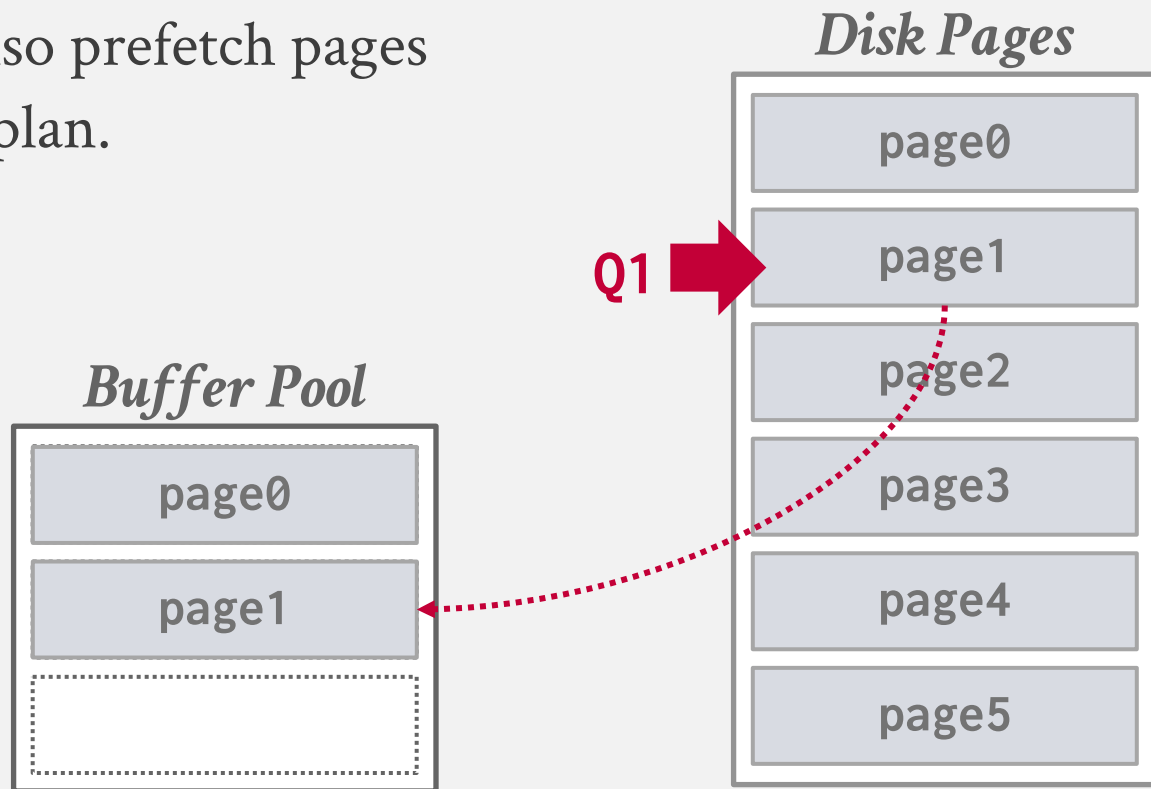
- Sequential Scans
- Index Scans



# PRE-FETCHING

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

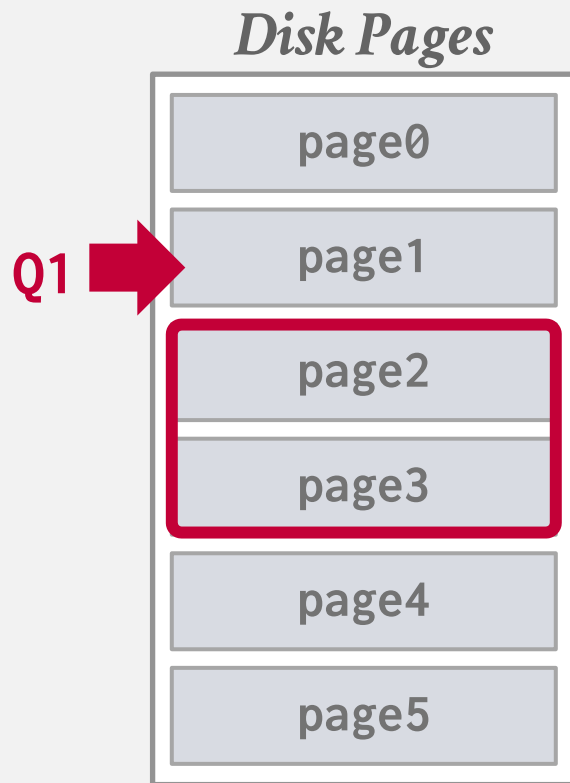
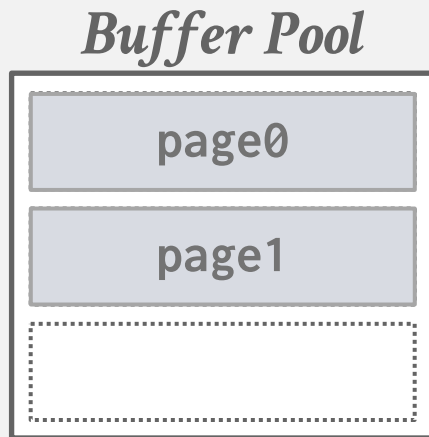
- Sequential Scans
- Index Scans



# PRE-FETCHING

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

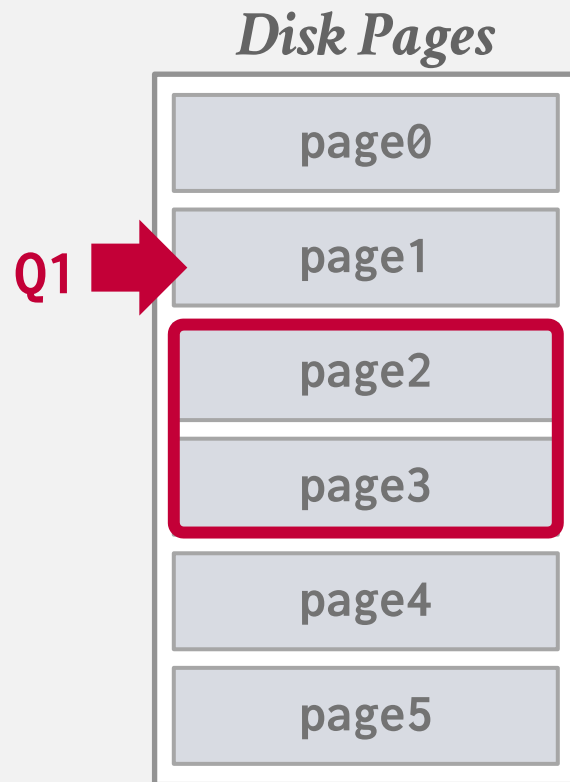
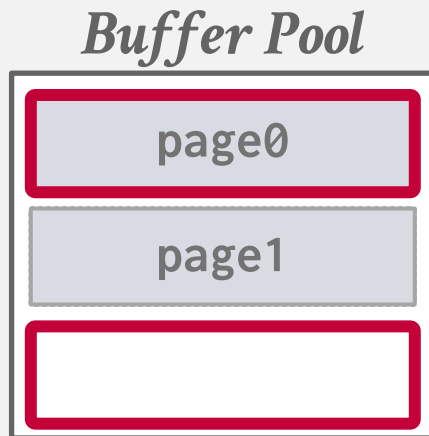
- Sequential Scans
- Index Scans



# PRE-FETCHING

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

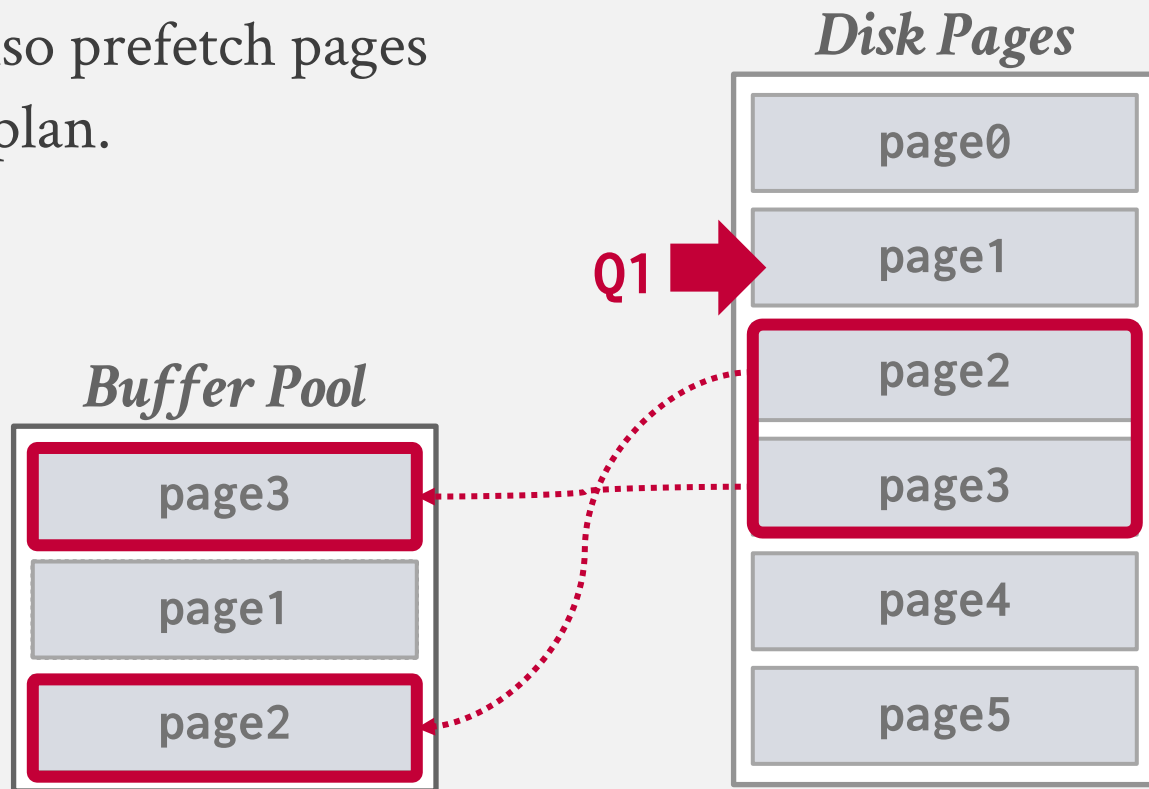
- Sequential Scans
- Index Scans



# PRE-FETCHING

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

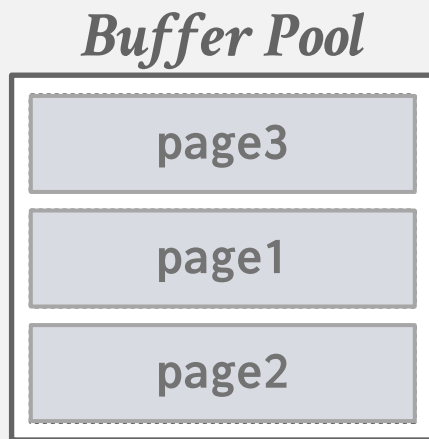
- Sequential Scans
- Index Scans



# PRE-FETCHING

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

- Sequential Scans
- Index Scans

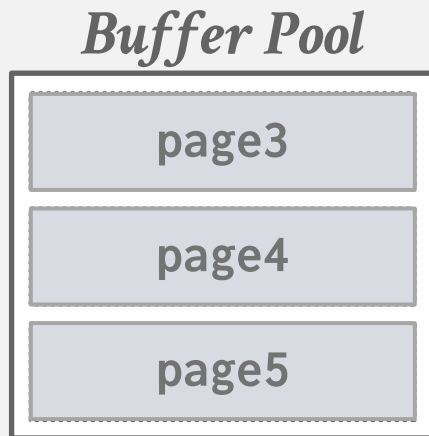




# PRE-FETCHING

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

- Sequential Scans
- Index Scans



# PRE-FETCHING

**Q1**

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

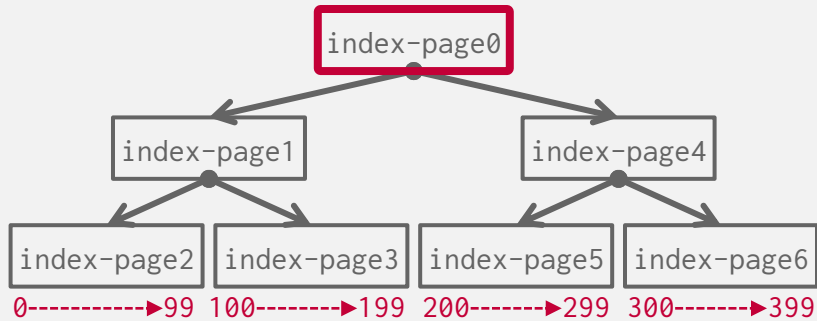
*Buffer Pool*



*Disk Pages*



# PRE-FETCHING



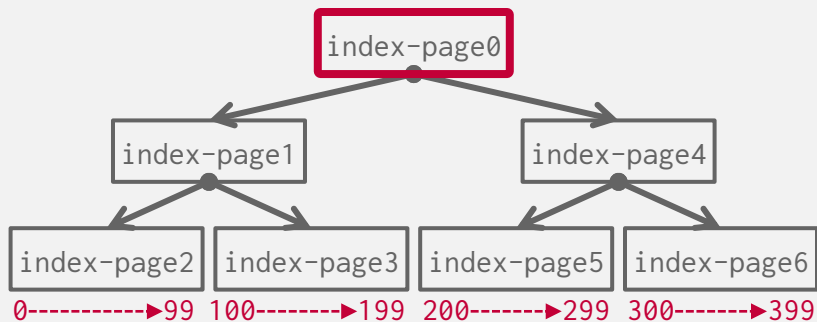
## Buffer Pool



## Disk Pages



# PRE-FETCHING



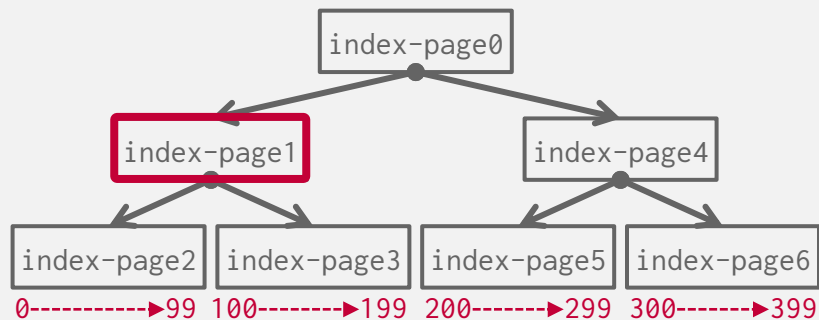
## Buffer Pool



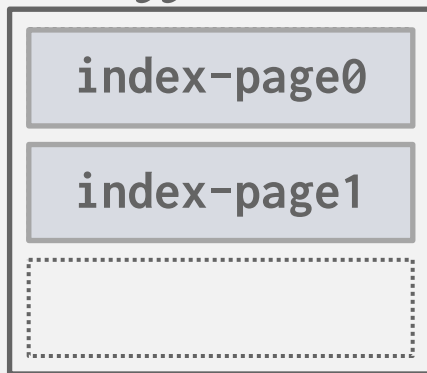
## Disk Pages



# PRE-FETCHING



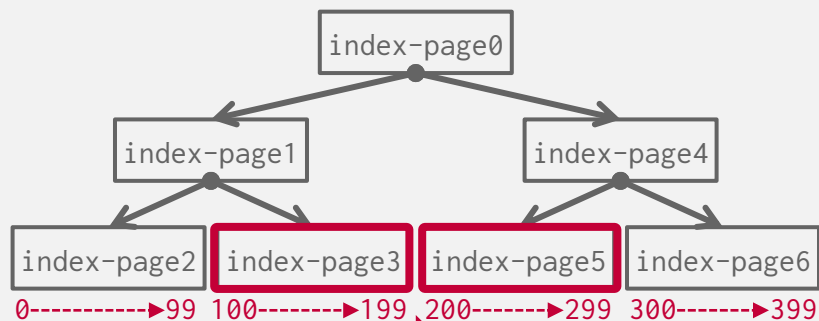
## Buffer Pool



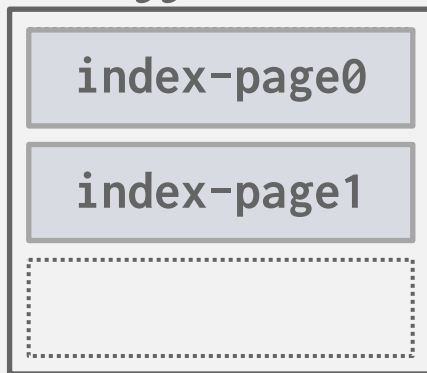
## Disk Pages



# PRE-FETCHING



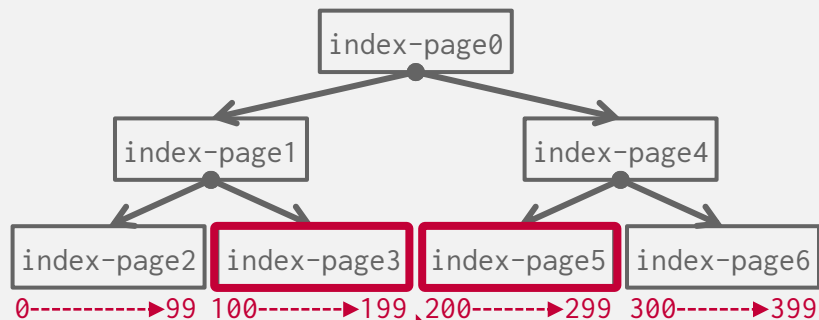
## Buffer Pool



## Disk Pages



# 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.

TERADATA

Microsoft  
SQL Server

IBM DB2

ORACLE

PostgreSQL

# 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 lines) 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.

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`

*Buffer Pool*



*Disk Pages*



# SCAN SHARING

**Q1** `SELECT SUM(val) FROM A`

*Buffer Pool*



*Disk Pages*



# SCAN SHARING

**Q1** `SELECT SUM(val) FROM A`

*Buffer Pool*

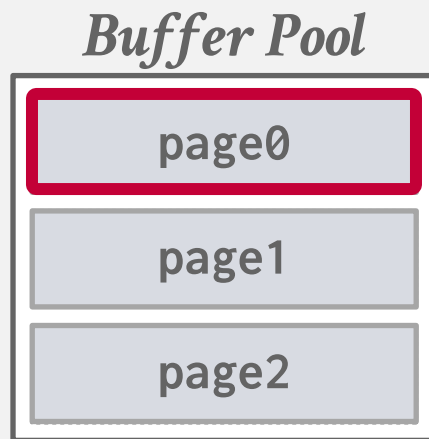


*Disk Pages*



# SCAN SHARING

**Q1** `SELECT SUM(val) FROM A`



# SCAN SHARING

**Q1** `SELECT SUM(val) FROM A`

*Buffer Pool*



*Disk Pages*





# SCAN SHARING

**Q1** SELECT SUM(val) FROM A

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

## *Buffer Pool*



## *Disk Pages*



# SCAN SHARING

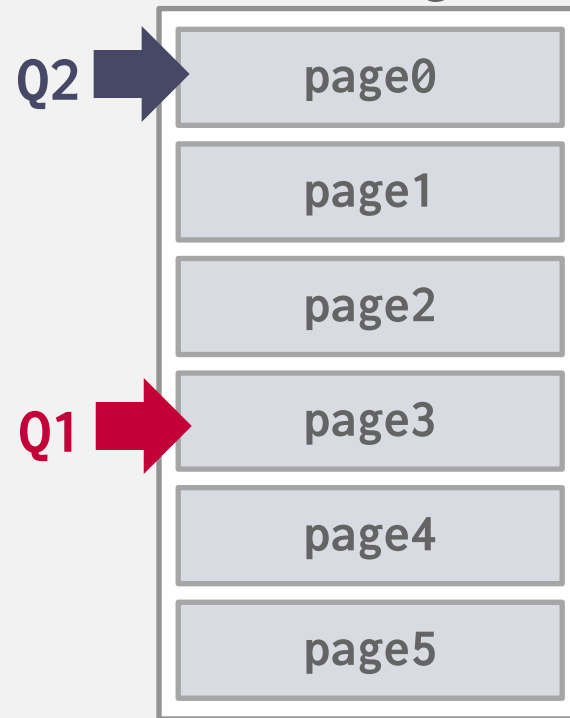
**Q1** SELECT SUM(val) FROM A

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

## *Buffer Pool*



## *Disk Pages*



# SCAN SHARING

**Q1** SELECT SUM(val) FROM A

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

## *Buffer Pool*



## *Disk Pages*



# SCAN SHARING

**Q1** SELECT SUM(val) FROM A

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

## *Buffer Pool*



## *Disk Pages*



# SCAN SHARING

**Q1** SELECT SUM(val) FROM A

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

## *Buffer Pool*



## *Disk Pages*



# SCAN SHARING

**Q1** SELECT SUM(val) FROM A

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

## *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.



# 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
- Metadata overhead

# LEAST-RECENTLY USED

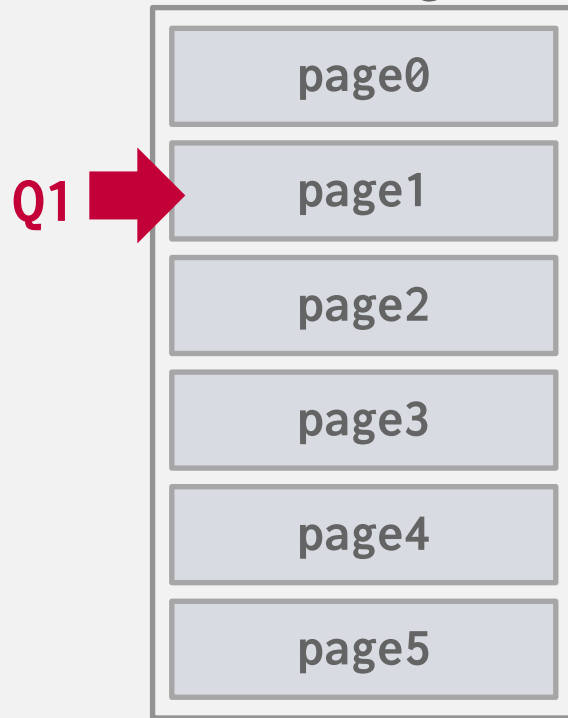
Maintain a single timestamp of when each page was last accessed. When the DBMS needs to evict a page, select the one with the oldest timestamp.

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

## *LRU List*



## *Disk Pages*



# 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.

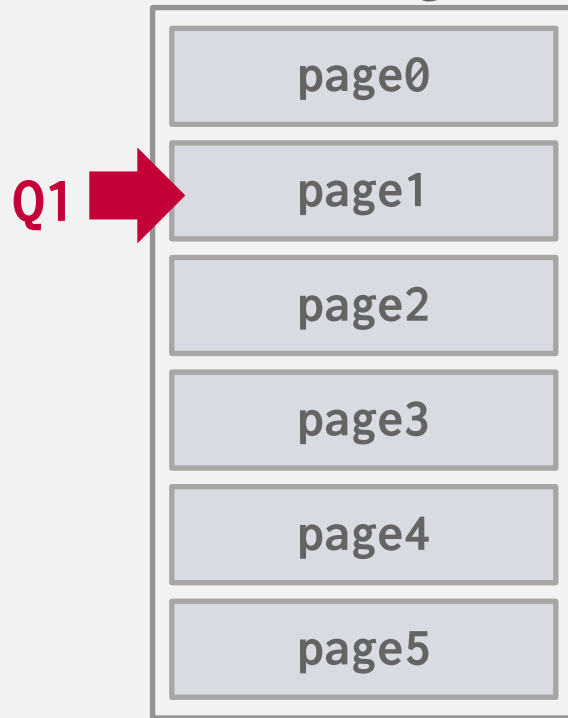
→ Keep the pages in sorted order to reduce the search time on eviction.

## *LRU List*



*Newest ← Oldest*

## *Disk Pages*



# LEAST-RECENTLY USED

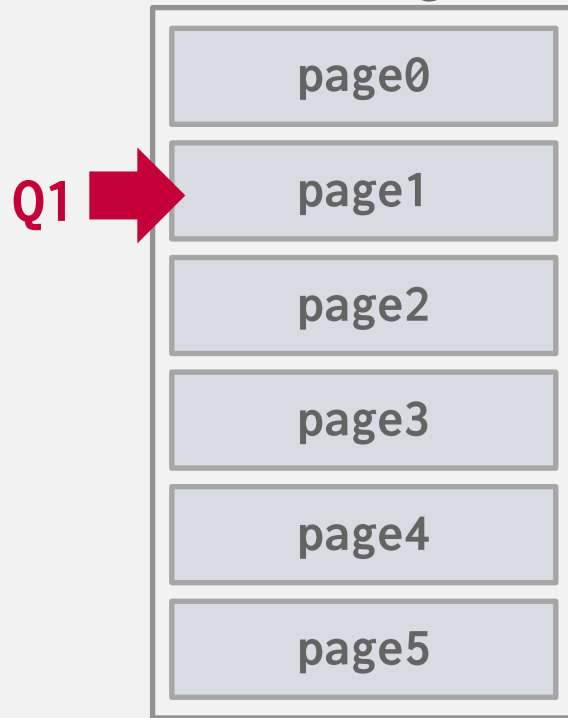
Maintain a single timestamp of when each page was last accessed. When the DBMS needs to evict a page, select the one with the oldest timestamp.

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

## *LRU List*



## *Disk Pages*



# 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.

ref=0

page1

ref=0

page4

ref=0

page2

page3

ref=0

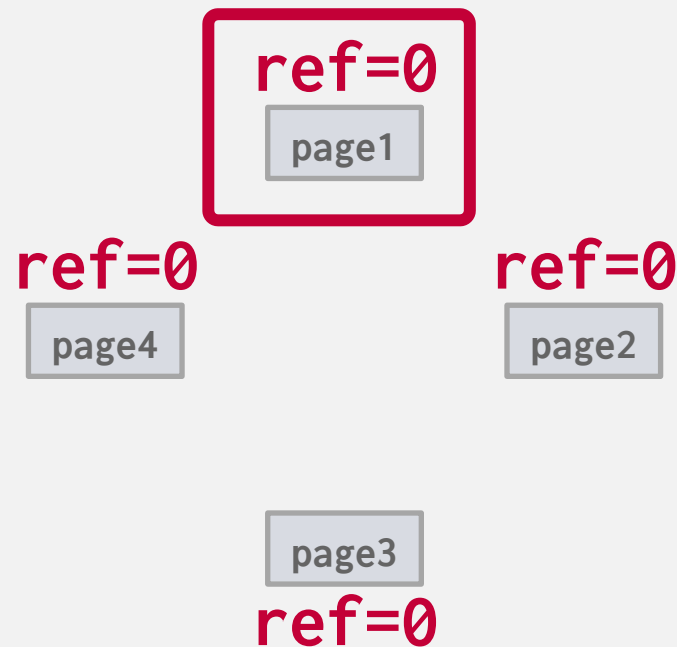
# CLOCK

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.



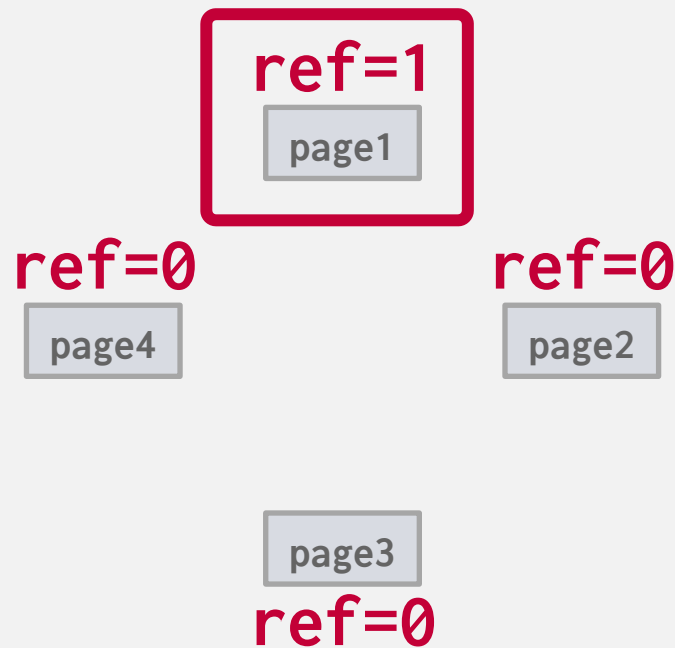
# CLOCK

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.



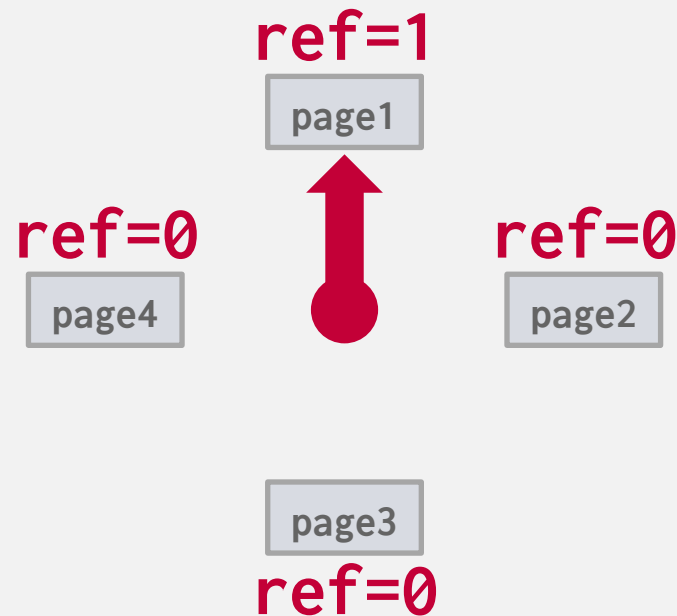
# CLOCK

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.





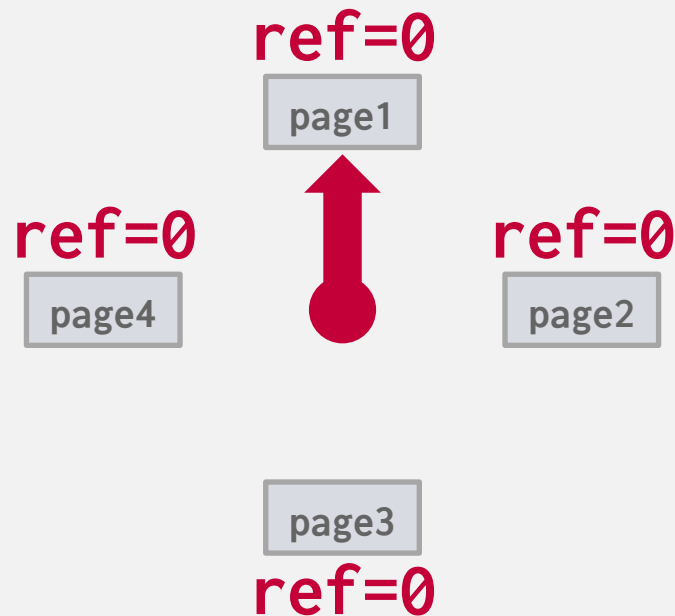
# CLOCK

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.



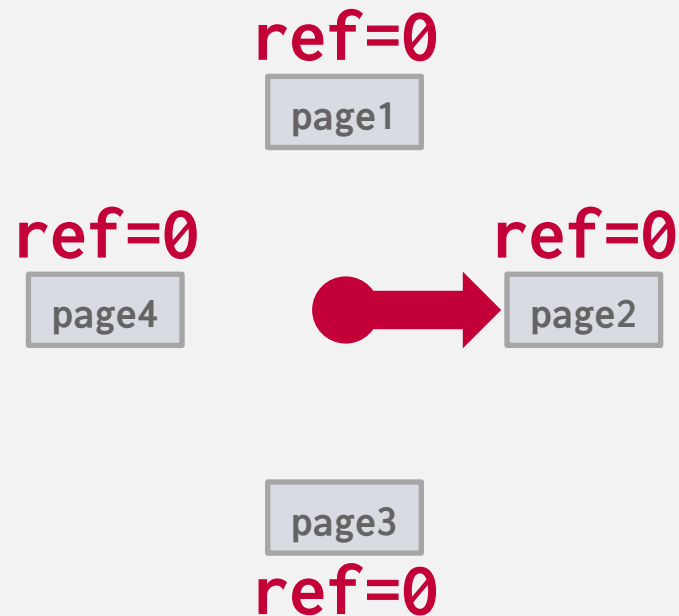
# CLOCK

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.



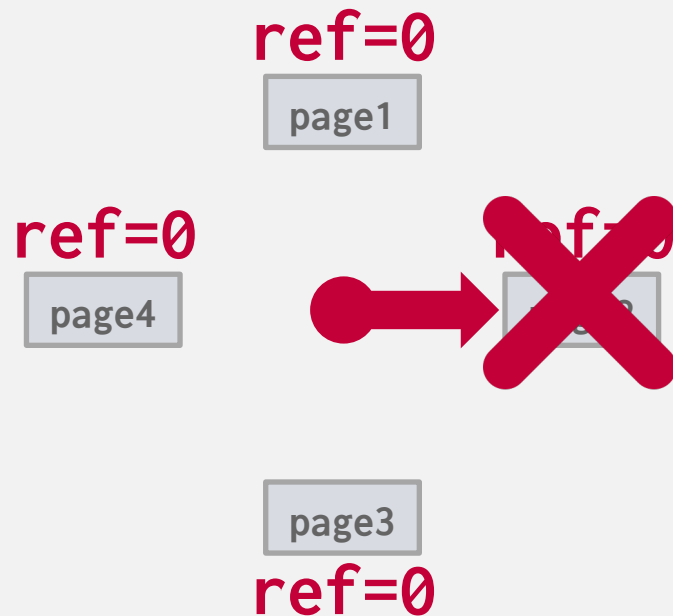
# CLOCK

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.



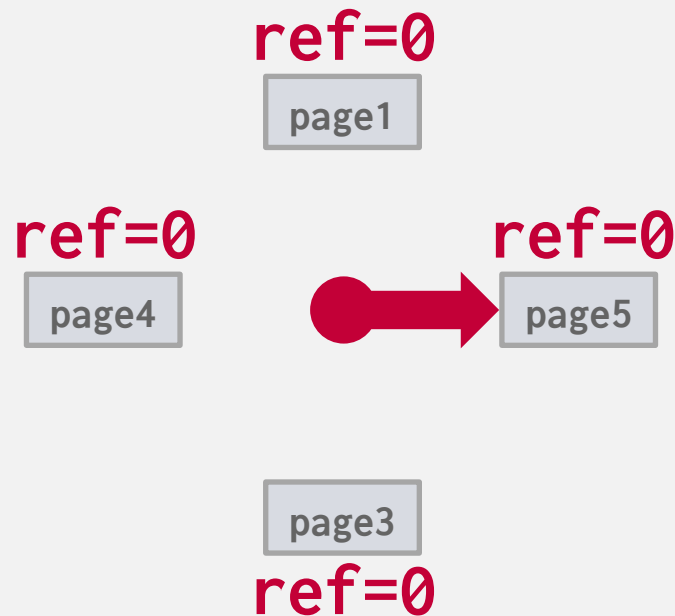
# CLOCK

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.



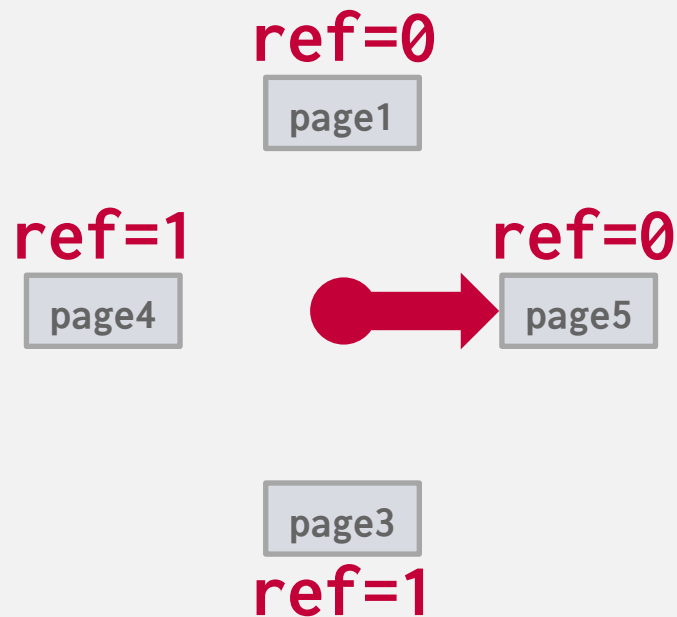
# CLOCK

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.



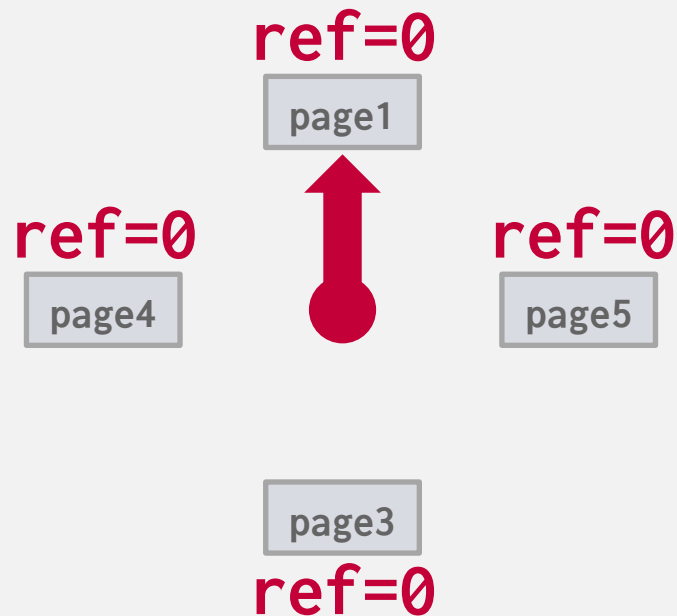
# CLOCK

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.



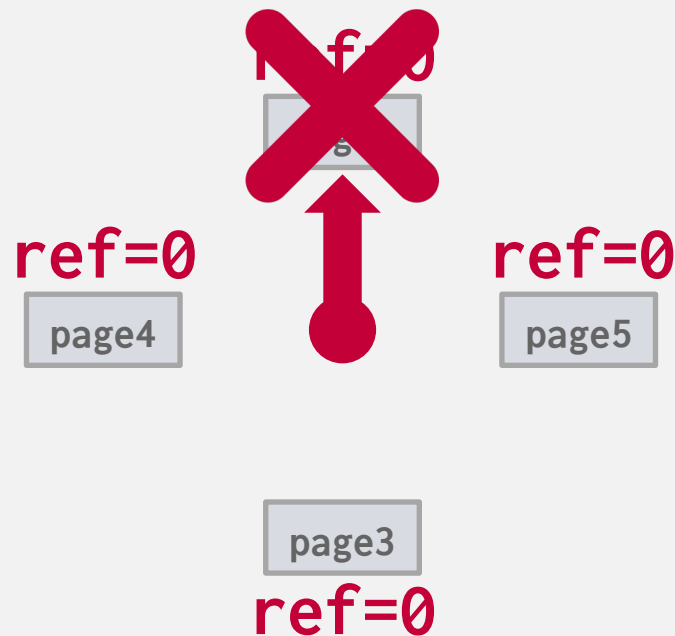
# CLOCK

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.



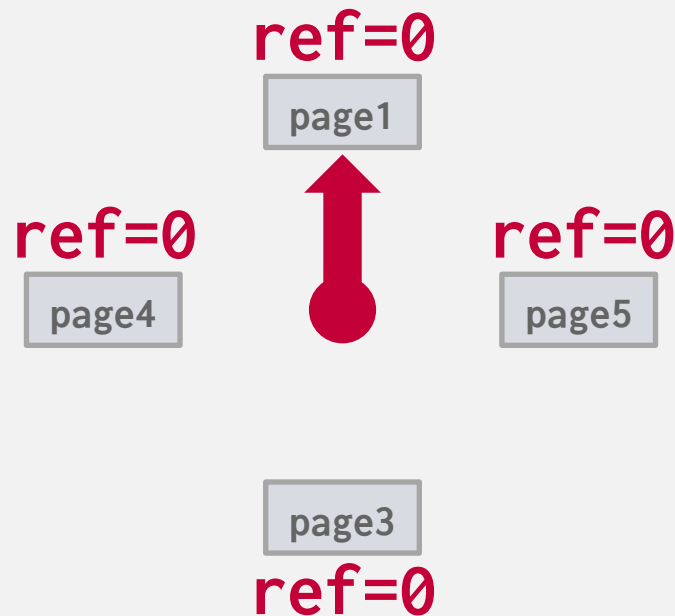
# CLOCK

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

---

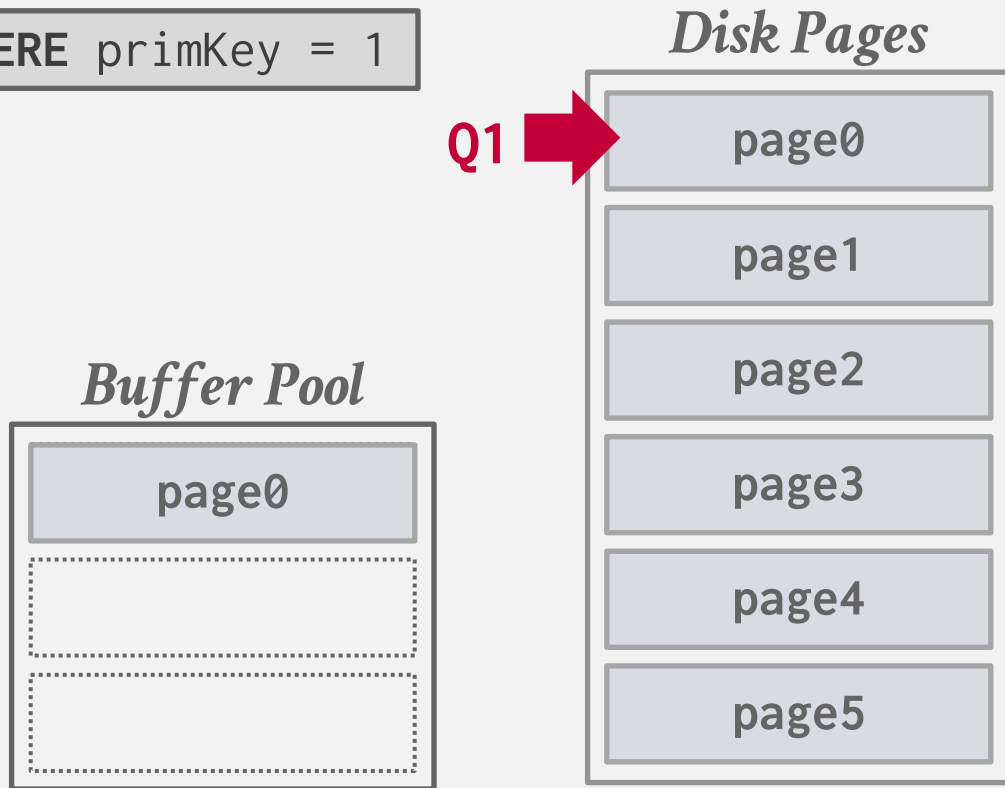
LRU + CLOCK replacement policies are susceptible to sequential flooding.

- 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

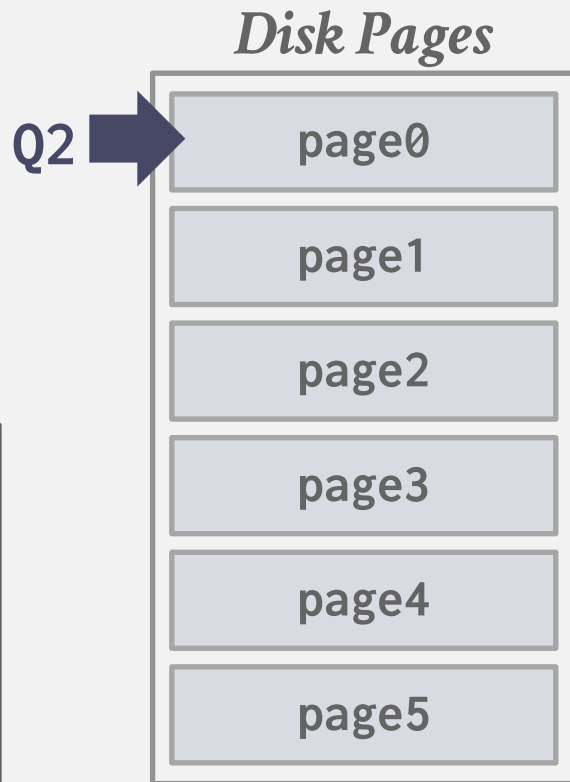
**Q1** `SELECT * FROM A WHERE primKey = 1`



# SEQUENTIAL FLOODING

**Q1** SELECT \* FROM A WHERE primKey = 1

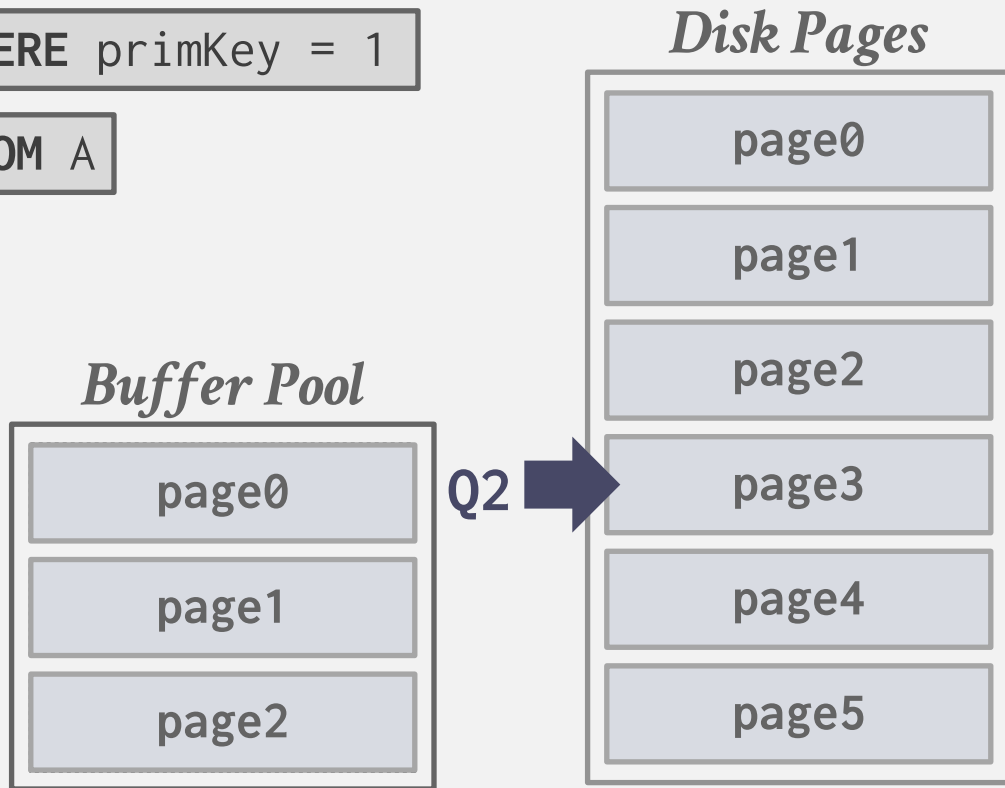
**Q2** SELECT AVG(val) FROM A



# SEQUENTIAL FLOODING

**Q1** SELECT \* FROM A WHERE primKey = 1

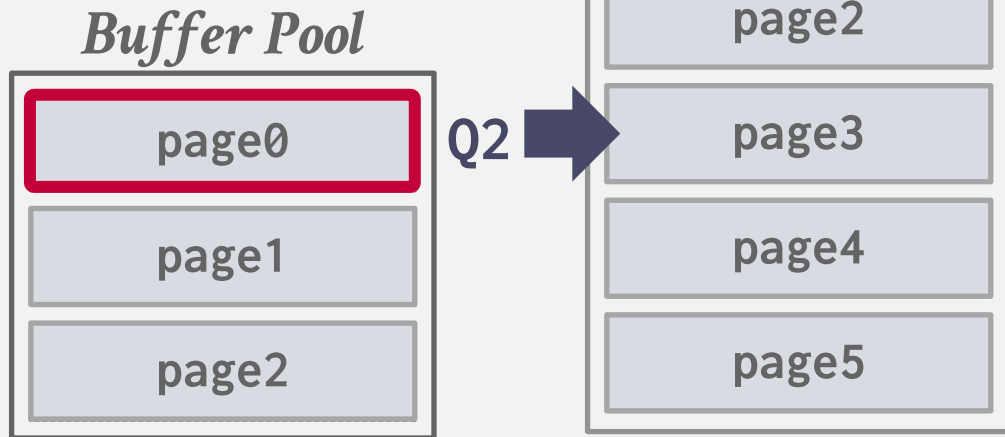
**Q2** SELECT AVG(val) FROM A



# SEQUENTIAL FLOODING

**Q1** SELECT \* FROM A WHERE primKey = 1

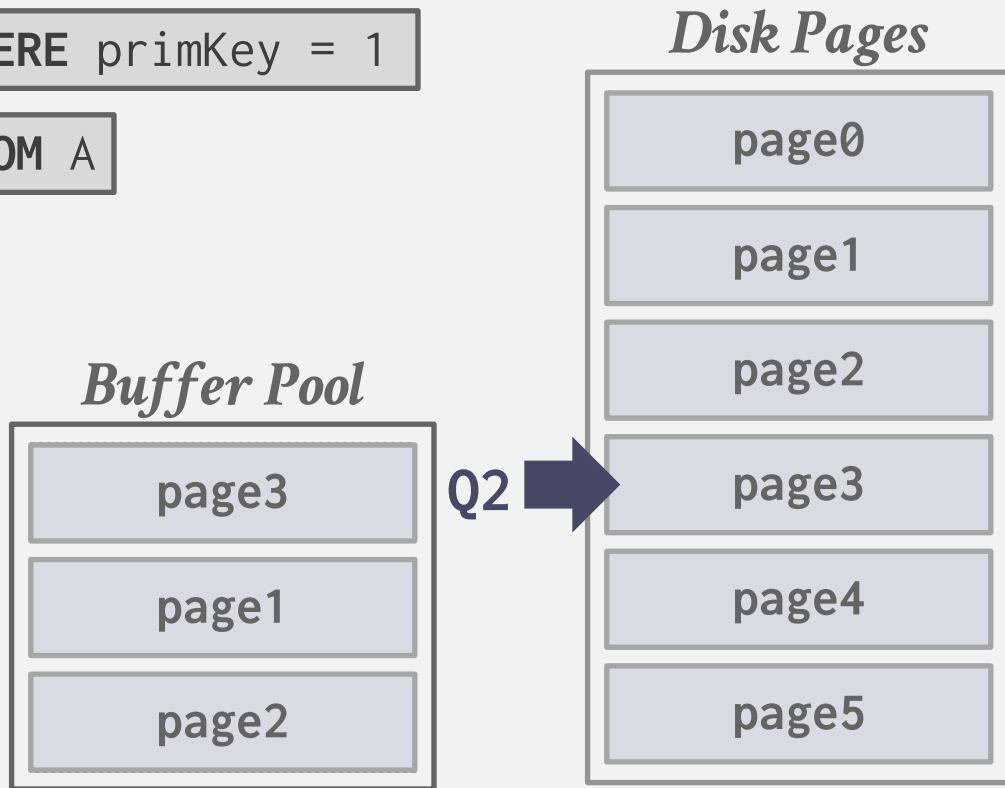
**Q2** SELECT AVG(val) FROM A



# SEQUENTIAL FLOODING

**Q1** SELECT \* FROM A WHERE primKey = 1

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

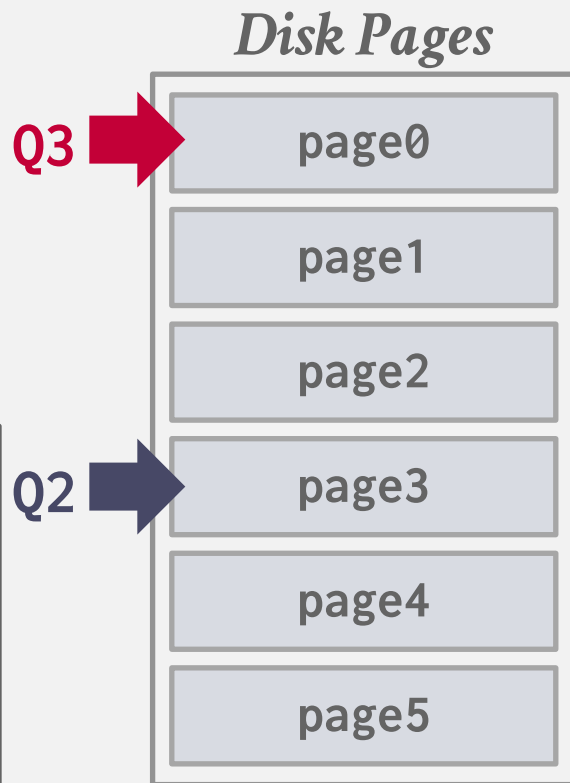
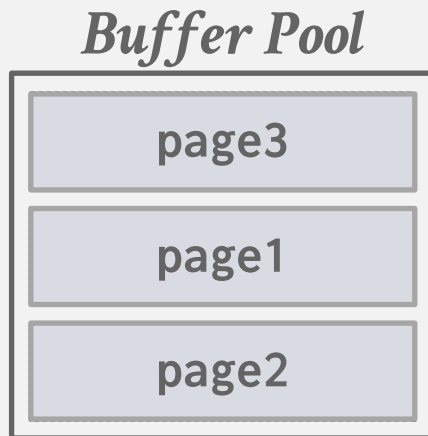


# SEQUENTIAL FLOODING

**Q1** SELECT \* FROM A WHERE primKey = 1

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

**Q3** SELECT \* FROM A WHERE primKey = 1

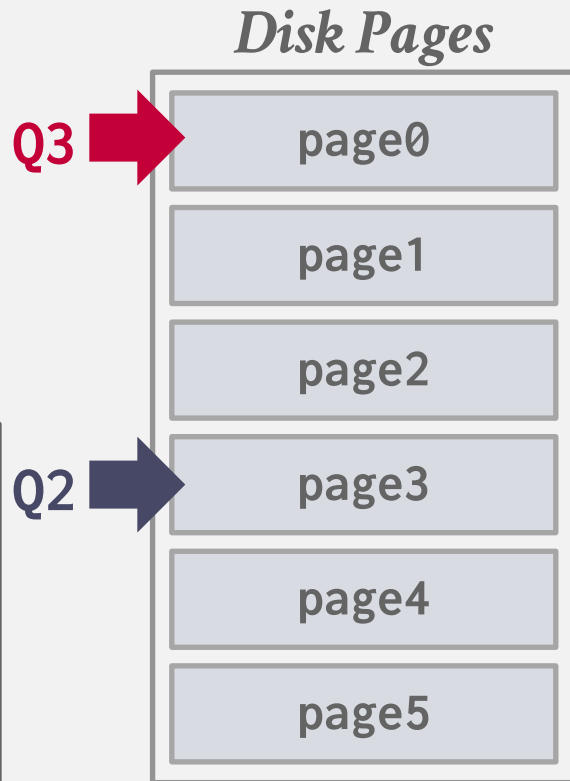
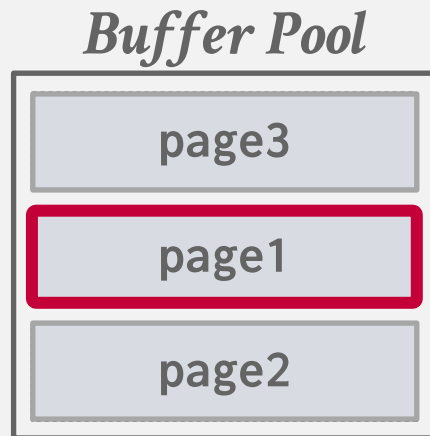


# SEQUENTIAL FLOODING

**Q1** SELECT \* FROM A WHERE primKey = 1

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

**Q3** SELECT \* FROM A WHERE primKey = 1





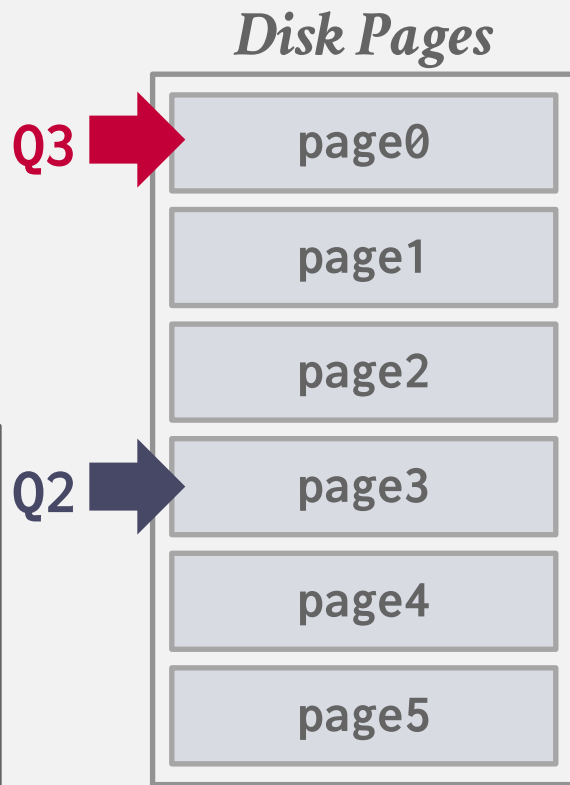
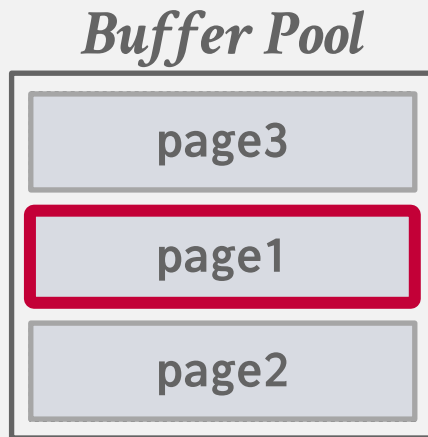
# SEQUENTIAL FLOODING

**Q1** SELECT \* FROM A WHERE primaryKey = 1

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

**Q3** SELECT \* FROM A WHERE primaryKey = 1

Sequential flooding can occur when a table is scanned multiple times within one query, such as with a Nested-Blocks Join.



# BETTER POLICIES: LRU-K

Track the last  $K$  references to each page and compute the interval between subsequent accesses.

→ Can get fancy with distinguishing between reference types.

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

→ Replace the page with the oldest “ $K$ -th” access.

→ A balance between recency and frequency of access.

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

## The LRU-K Page Replacement Algorithm For Database Disk Buffering

Elizabeth J. O’Neil<sup>1</sup>, Patrick E. O’Neil<sup>1</sup>, Gerhard Weikum<sup>2</sup>

<sup>1</sup> Department of Mathematics and Computer Science  
University of Massachusetts at Boston  
Harbor Campus  
Boston, MA 02125-3393

<sup>2</sup> Department of Computer Science  
ETH Zurich  
CH-8092 Zurich  
Switzerland

E-mail: eoneil@cs.umb.edu, pooneil@cs.umb.edu, weikum@inf.ethz.ch

### ABSTRACT

This paper introduces a new approach to database disk buffering, called the LRU-K method. The basic idea of LRU-K is to keep track of the times of the last  $K$  reference to popular database pages, using this information to statistically estimate the interarrival times of references on a page by page basis. Although the LRU-K approach performs optimal statistical inference under relatively standard assumptions, it is fairly simple and incurs little bookkeeping overhead. As we demonstrate with simulation experiments, the LRU-K algorithm surpasses conventional buffering algorithms in discriminating between frequently and infrequently referenced pages. In fact, LRU-K can approach the behavior of buffering algorithms in which page sets with known access frequencies are manually assigned to different buffer pools of specifically tuned sizes. Unlike such customized buffering algorithms however, the LRU-K method is self-tuning, and does not rely on external hints about workload characteristics. Furthermore, the LRU-K algorithm adapts in real time to changing patterns of access.

### 1. Introduction

#### 1.1 Problem Statement

All database systems retain disk pages in memory buffers for a period of time after they have been read in from disk and accessed by a particular application. The purpose is to keep popular pages memory resident and reduce disk I/O. In their ‘Five Minute Rule’, Gray and Putzolu pose the following tradeoff: We are willing to pay more for memory buffers up to a certain point, in order to reduce the cost of disk arms for a system (GRRAPYPT), see also (CKS). The critical buffering decision arises when a new buffer slot is needed for a page about to be read in from disk, and all current buffers are in use: What current page should be dropped from buffer? This is known as the page replacement policy, and the different buffering algorithms take their name from the type of replacement policy they impose: one, for example, (COFFENEN), (EFFHARR).

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

SIGMOD ’83/Washington, DC USA  
© 1983 ACM 0-89791-532-5/83/0005-0297...\$1.50

The algorithm utilized by almost all commercial systems is known as LRU, for Least Recently Used. When a new buffer is needed, the LRU policy drops the page from buffer that has not been accessed for the longest time. LRU buffering was developed originally for patterns of use in instruction logic (for example, (DENNING), (COFFENEN)), and does not always fit well into the database environment, as was noted also in (REITER), (STON), (SACSCHE), and (CHODURA). In fact, the LRU buffering algorithm has a problem which is addressed by the current paper: that it decides what page to drop from buffer based on too little information, limiting itself to only the time of last reference. Specifically, LRU is unable to differentiate between pages that have relatively frequent references and pages that have very infrequent references until the system has wasted a lot of resources keeping infrequently referenced pages in buffer for an extended period.

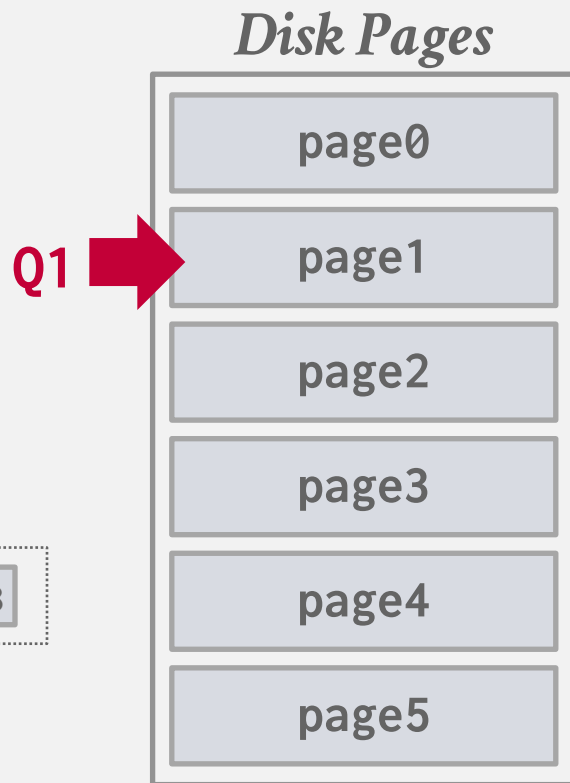
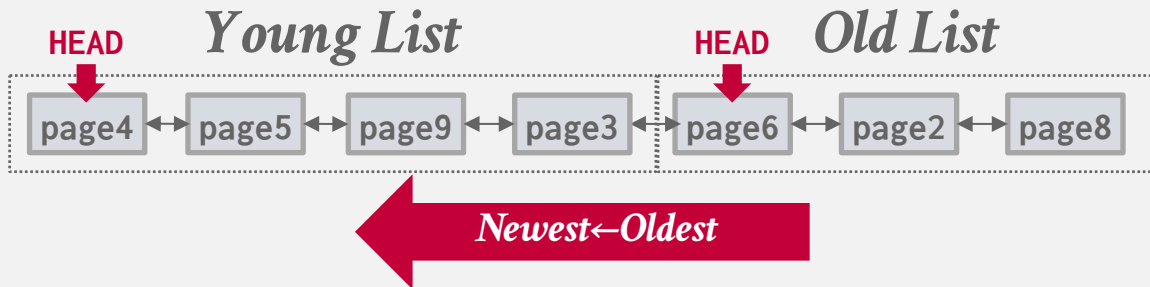
**Example 1.1.** Consider a multi-user database application, which references randomly chosen customer records through a clustered B-tree indexed key, CUST-ID, to retrieve desired information (cf. (TPC-A)). Assume simplistically that 20,000 customers exist, that a customer record is 2000 bytes in length, and that space needed for the B-tree index at the leaf level, (free space included), is 20 bytes for each key entry. Then if disk pages contain 4000 bytes of usable space and can be packed full, we require 100 pages to hold the leaf level nodes of the B-tree index (there is a single B-tree root node), and 10,000 pages to hold the records. The pattern of reference to these pages (ignoring the B-tree root node) is clearly: 11, R1, R2, R2, R3, R3, ..., alternate references to random index leaf pages and record pages. If we can only afford to buffer 101 pages in memory for this application, the B-tree root node is automatic; we should buffer all the B-tree leaf pages, since each of them is referenced with a probability of .005 (once in each 200 general page references), while it is clearly wasteful to displace one of these leaf pages with a data page; since data pages have only .00005 probability of reference (once in each 20,000 general page references). Using the LRU algorithm, however, the pages held in memory buffers will be the least recently referenced ones. To a first approximation, this means 50 leaf pages and 50 data pages. Given that a page gets no extra credit for being referenced twice in the recent past and that this is more likely to happen with B-tree leaf pages, there will even be slightly more data



# MYSQL APPROXIMATE LRU-K

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

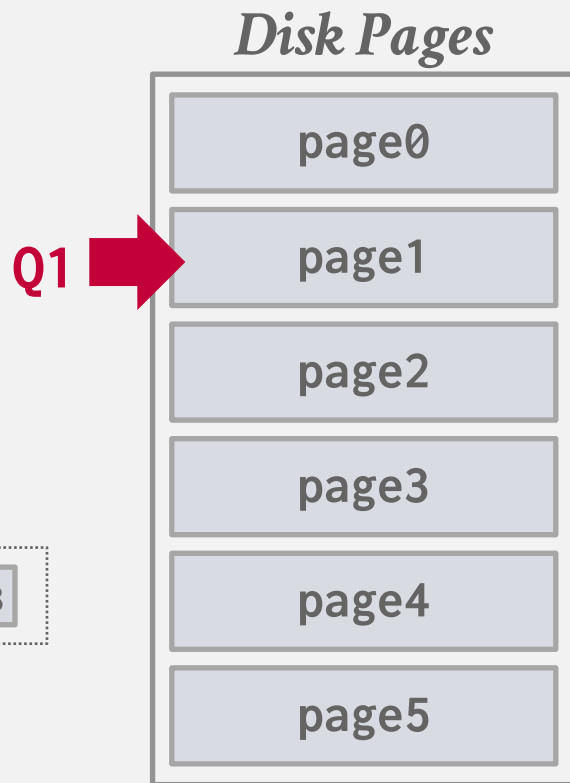
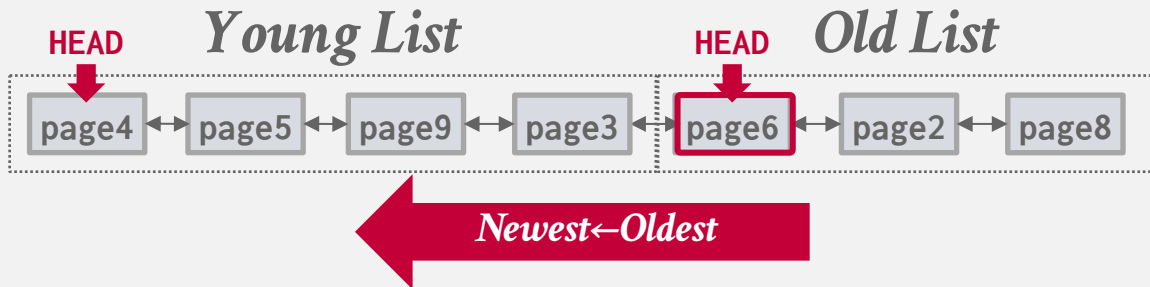
- New pages are always inserted to the head of the old list.
- If pages in the old list is accessed again, then insert into the head of the young list.



# MYSQL APPROXIMATE LRU-K

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

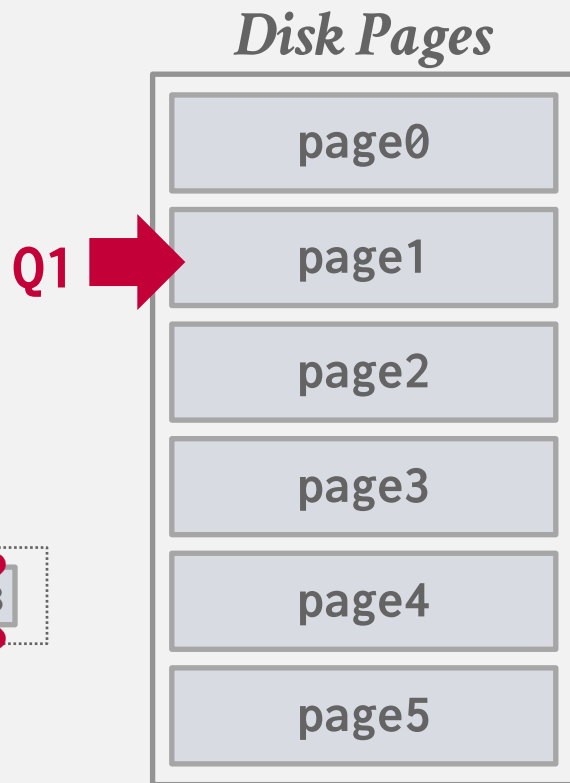
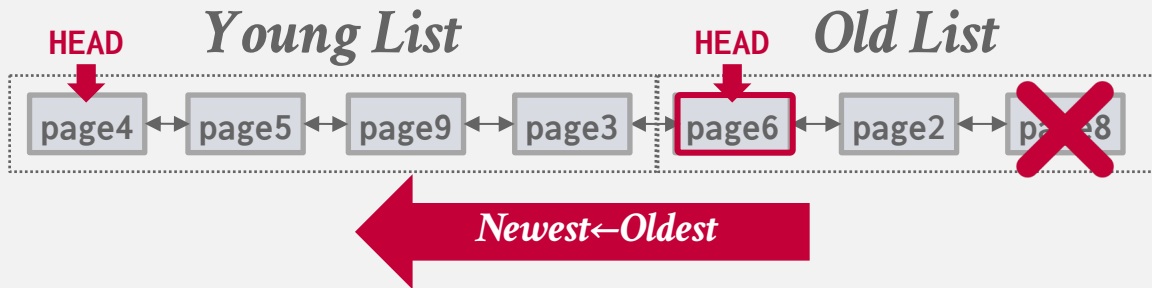
- New pages are always inserted to the head of the old list.
- If pages in the old list is accessed again, then insert into the head of the young list.



# MYSQL APPROXIMATE LRU-K

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

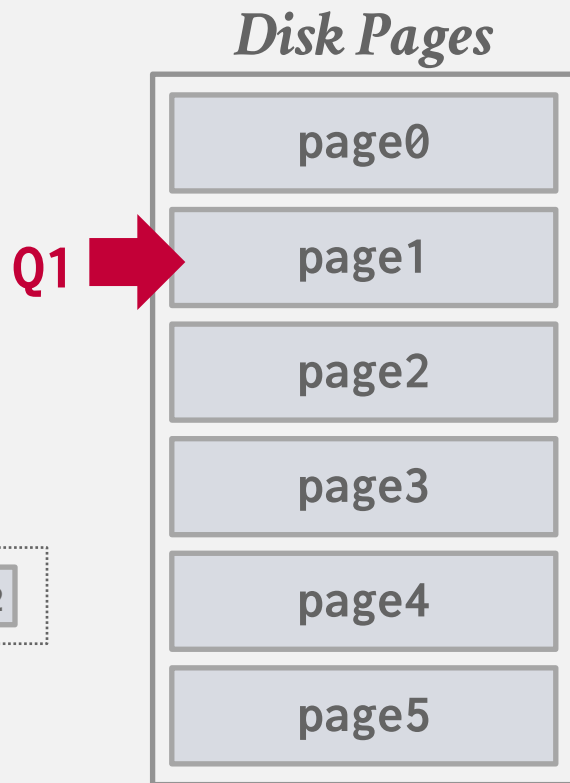
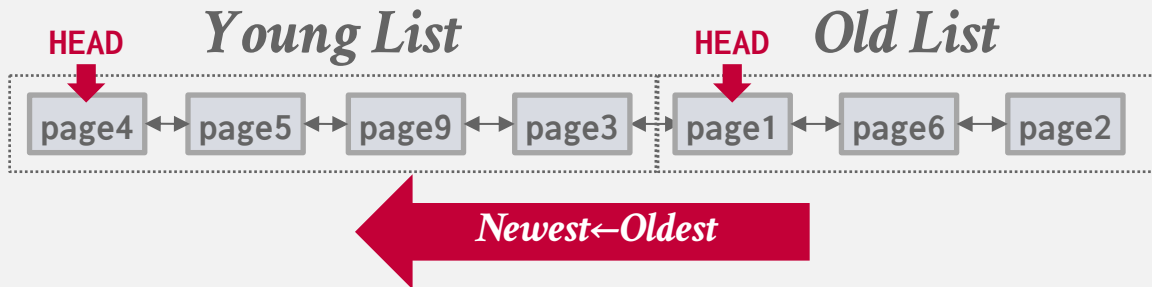
- New pages are always inserted to the head of the old list.
- If pages in the old list is accessed again, then insert into the head of the young list.



# MYSQL APPROXIMATE LRU-K

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

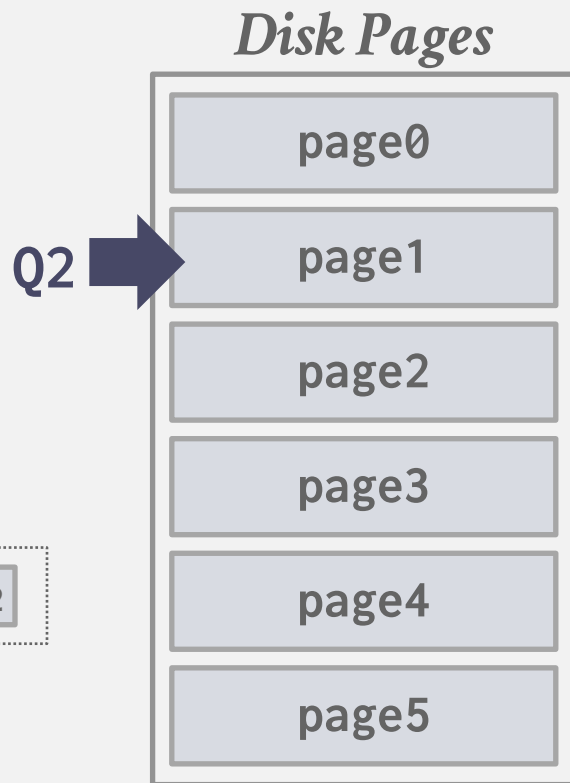
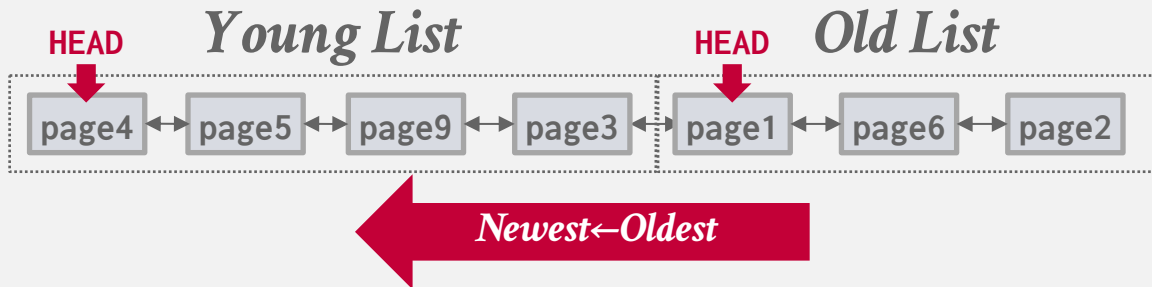
- New pages are always inserted to the head of the old list.
- If pages in the old list is accessed again, then insert into the head of the young list.



# MYSQL APPROXIMATE LRU-K

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

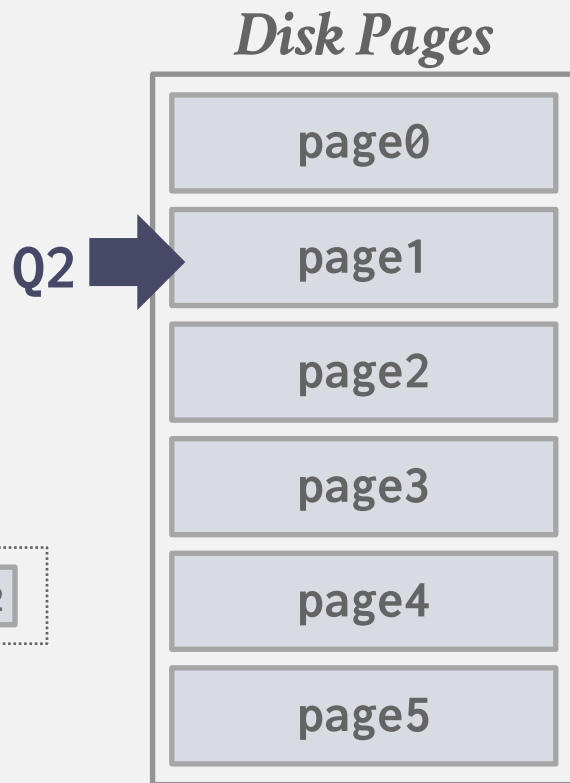
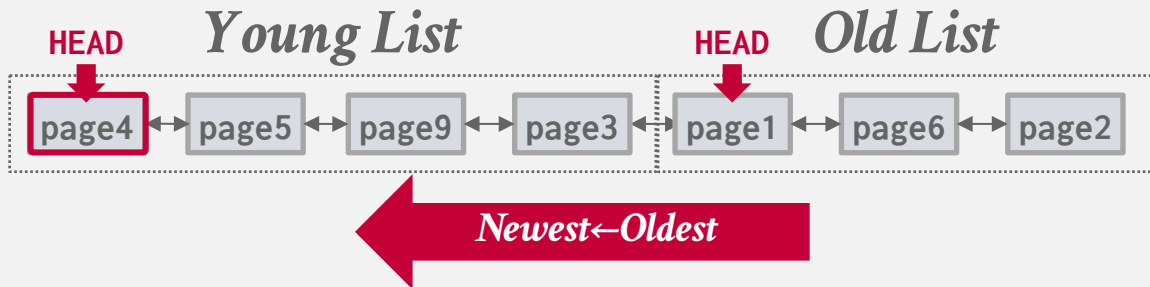
- New pages are always inserted to the head of the old list.
- If pages in the old list is accessed again, then insert into the head of the young list.



# 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.

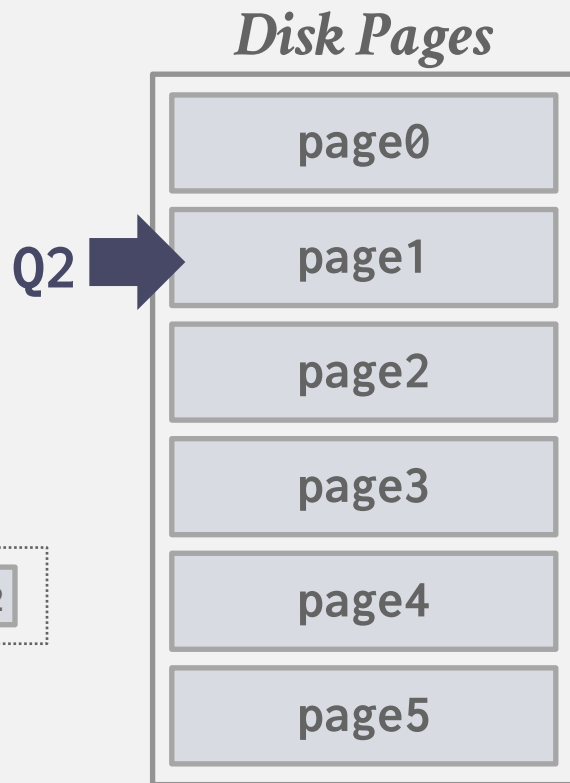
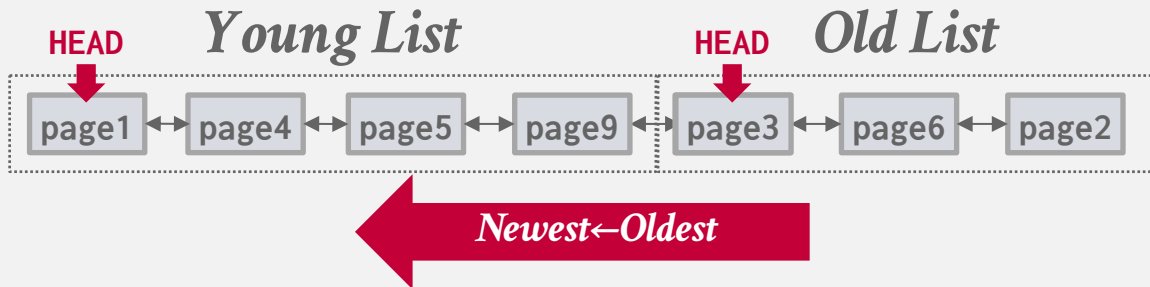




# MYSQL APPROXIMATE LRU-K

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.

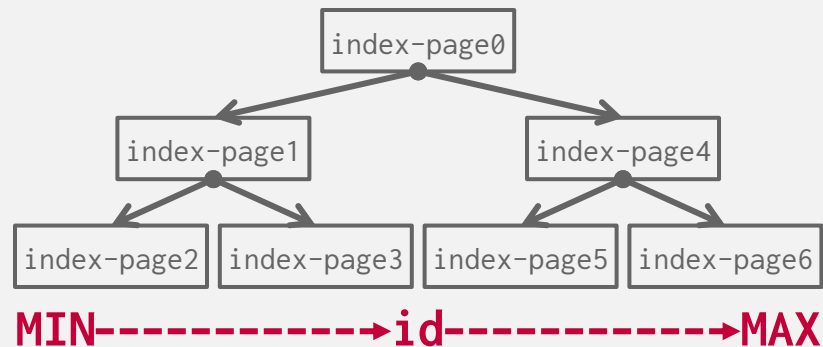
Example: Postgres maintains a small ring buffer that is private to the query.

# 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++*)

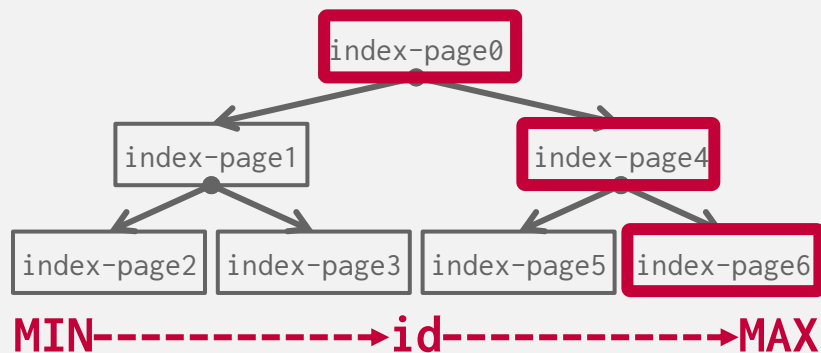


# BETTER POLICIES: PRIORITY HINTS

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++)`



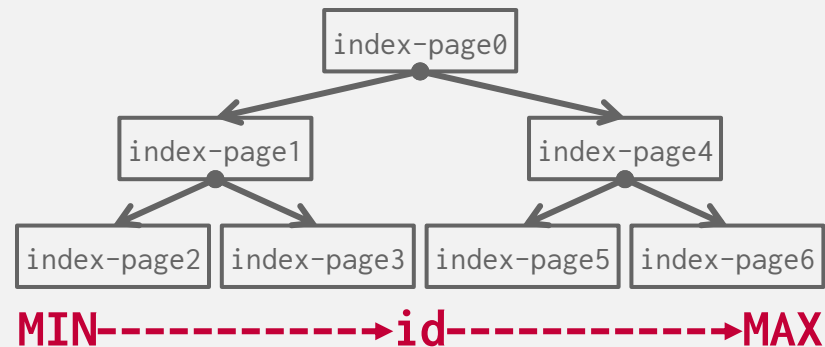
# BETTER POLICIES: PRIORITY HINTS

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 = ?



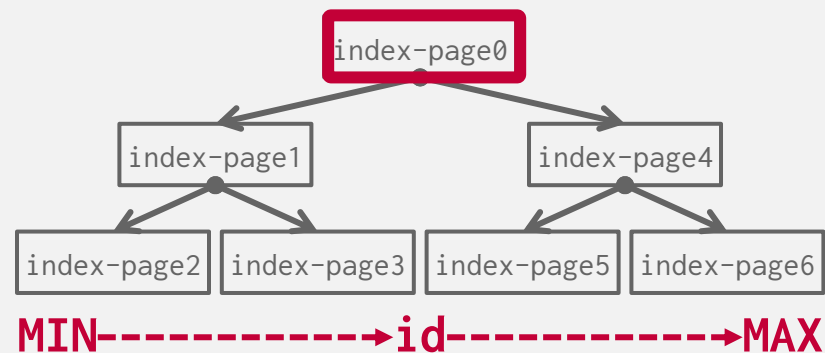
# BETTER POLICIES: PRIORITY HINTS

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

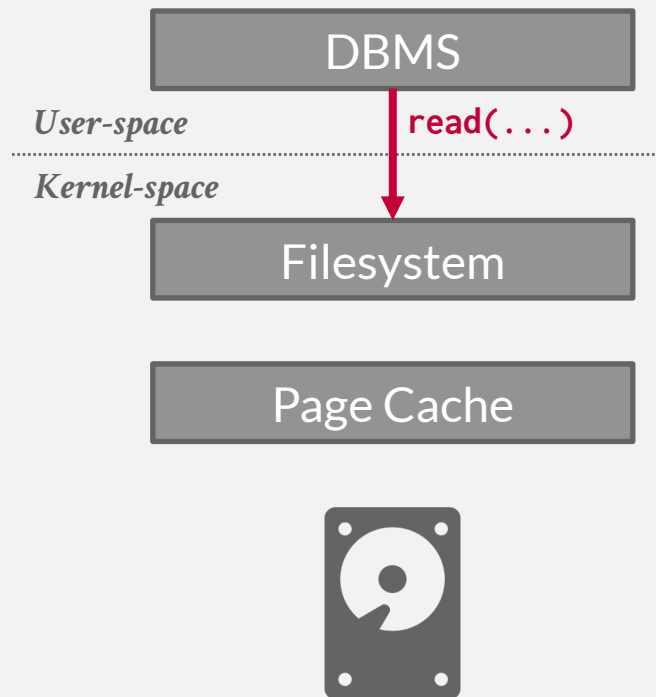
The OS doesn't know these things and is going to get into the way...

# 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.

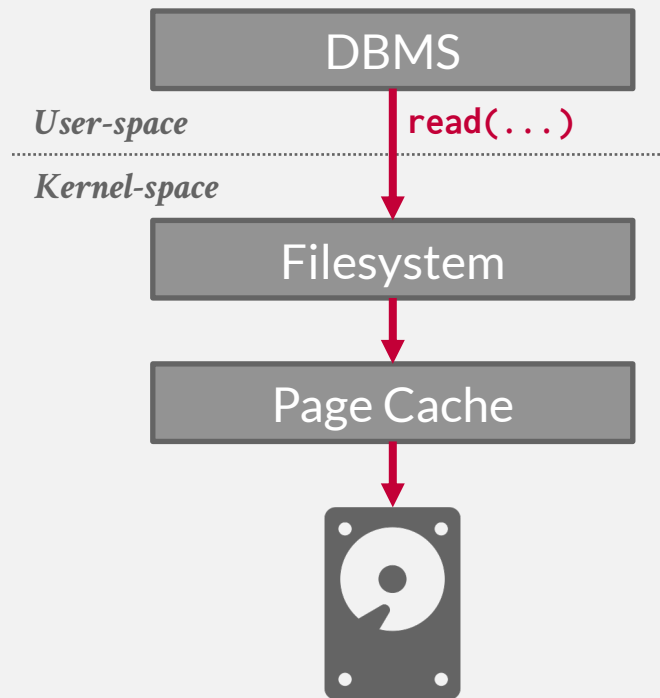


# OS PAGE CACHE

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.

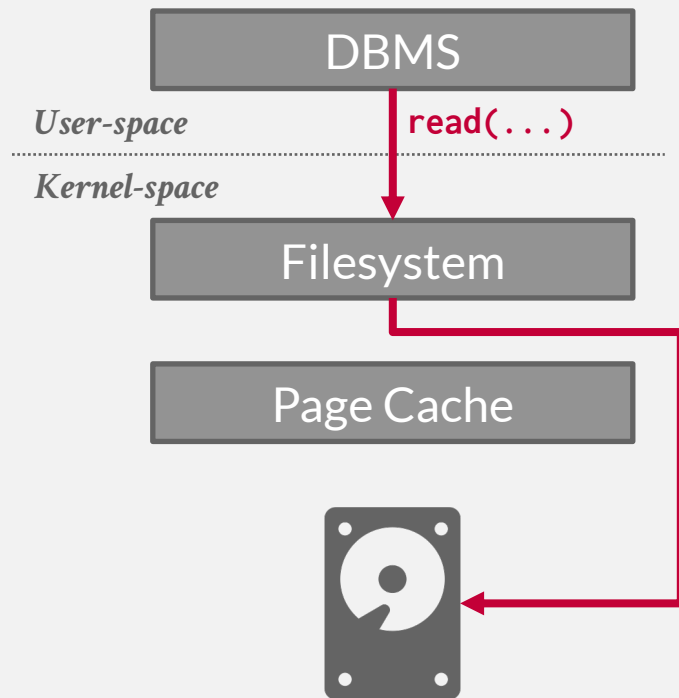


# OS PAGE CACHE

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 **fwrite**, what happens?

If the DBMS calls **fsync**, what happens?

If **fsync** fails (EIO), what happens?

- Linux marks the dirty pages as clean.
- If the DBMS calls **fsync** again, then Linux tells you that the flush was successful. 💀

# FSYNC PROBLEMS

If the DBMS calls **f**

If the DBMS calls **f**

If **fsync** fails (EIO)

→ Linux marks the di

→ If the DBMS calls **f**  
the flush was succe

The screenshot shows a Wikipedia article titled "Fsync Errors". The article discusses the current status, history, and OS differences related to the `fsync()` reliability issues, particularly around the "fsyncgate 2018". It mentions that PostgreSQL 12 will now PANIC on `fsync()` failure, which was backported to PostgreSQL 11, 10, 9.6, 9.5, and 9.4. The article also notes that Linux kernel 4.13 improved `fsync()` error handling and the man page for `fsync()` is somewhat improved. It lists several particularly significant 4.13 commits, including new infrastructure for writeback error handling, use of `errseq_t` based error handling, and documentation updates. The article concludes with thanks to Jeff Layton for his work and mentions similar changes in InnoDB/MySQL, WiredTiger/MongoDB, and other software. A proposed follow-up change to PostgreSQL is also discussed, noting that a patch that was committed did not incorporate some additional safeguards.

**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 `errseq_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 `errseq_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 or 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

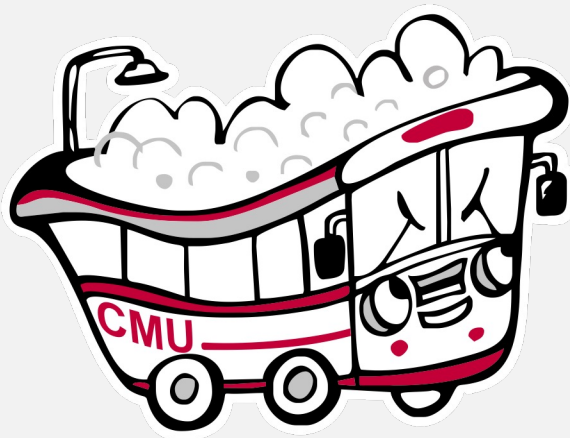
# PROJECT #1

---

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 Feb 18<sup>th</sup> @ 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**.

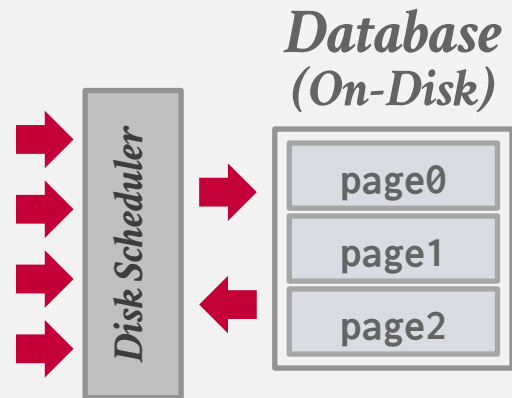
## TASK #2 - DISK SCHEDULER

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!

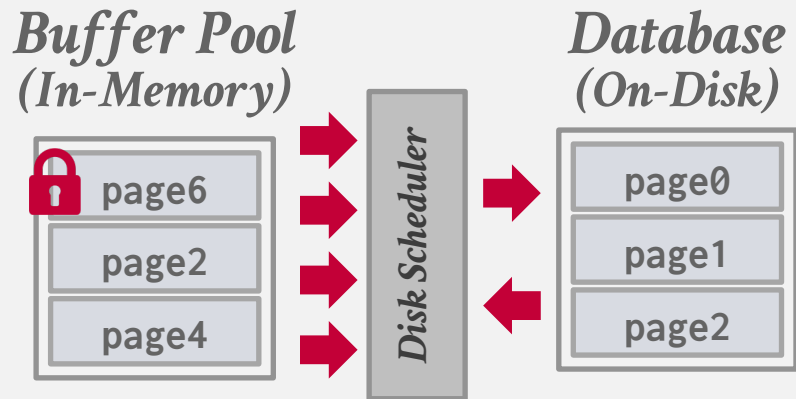


# TASK #3 - BUFFER POOL MANAGER

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 besides the ones 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

The 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.