Intro to Database Systems (15-445/645)

06 Memory Management

Carnegie Mellon University

FALL 2022

Andy Pavlo

ADMINISTRIVIA

Homework #2 is due September 25th @ 11:59pm

Project #1 is due October 2nd @ 11:59pm

→ Q&A Session: **Tuesday September 22nd @ 8:00pm**
→ Special Office Hours: **Saturday October 1st @ 3pm-5pm**

DATABASE STORAGE

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

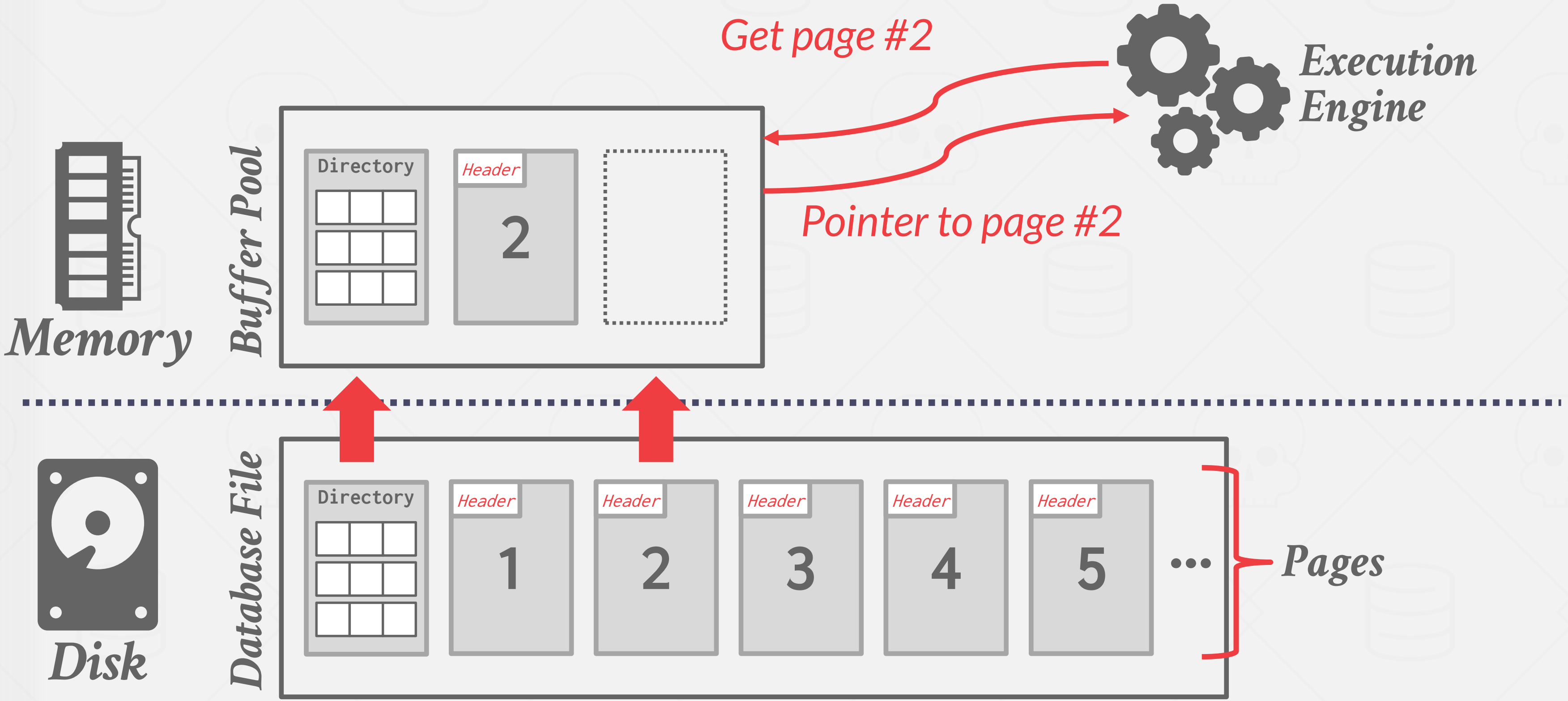
Spatial Control:

→ 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.

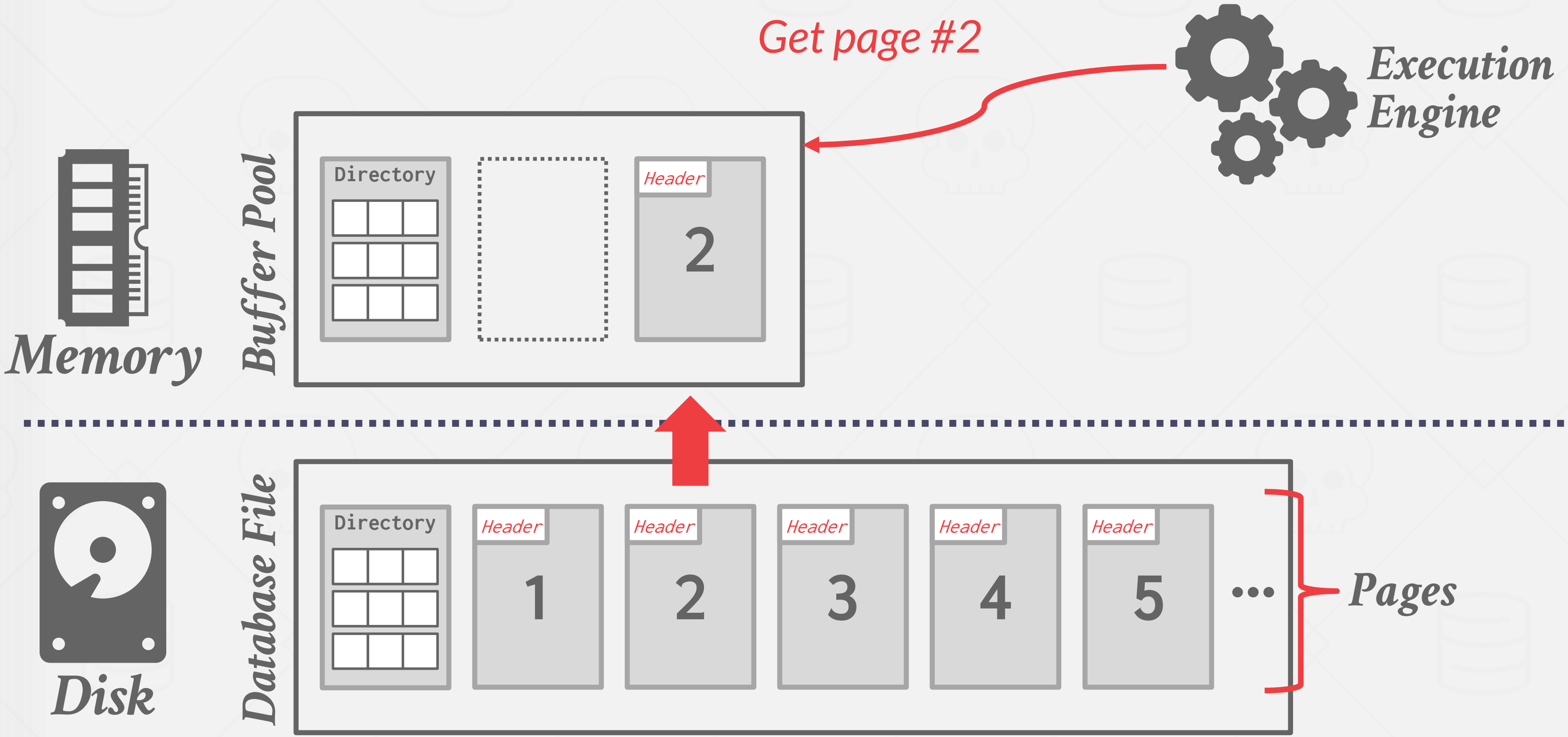
Temporal Control:

→ 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

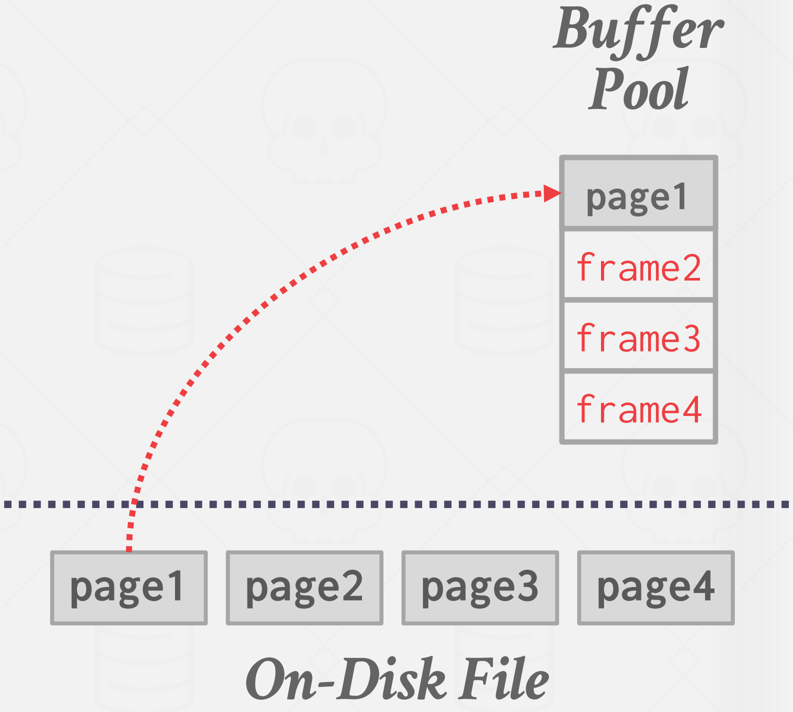
Buffer Pool Manager
Replacement Policies
Other Memory Pools

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

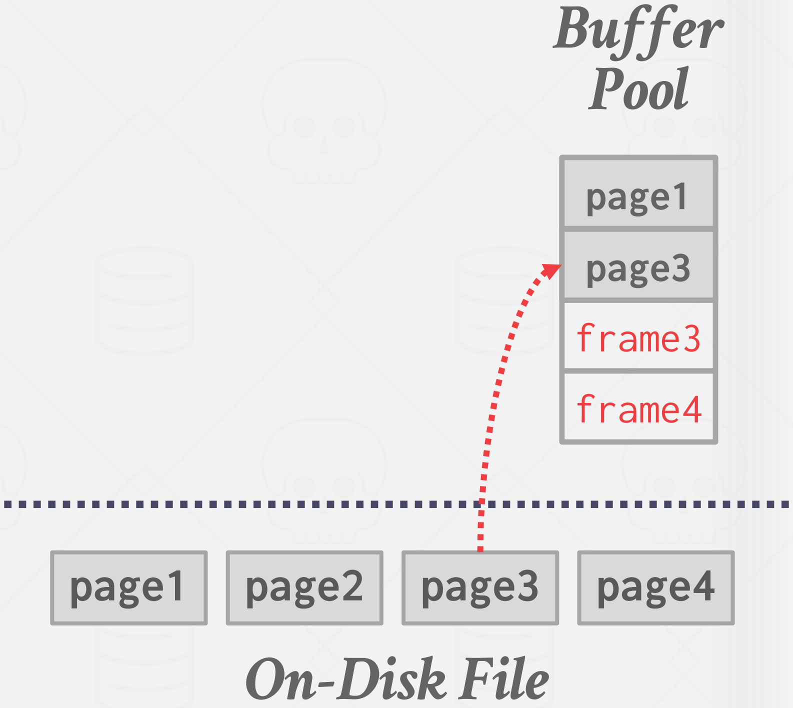


BUFFER POOL ORGANIZATION

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.

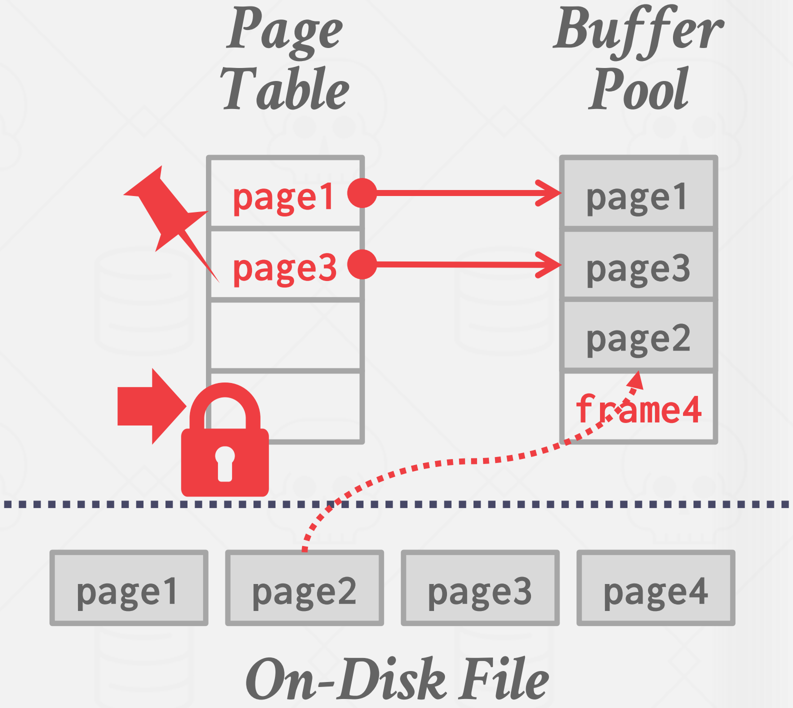
Dirty pages are buffered and not written to disk immediately
→ Write-Back Cache



BUFFER POOL META-DATA

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

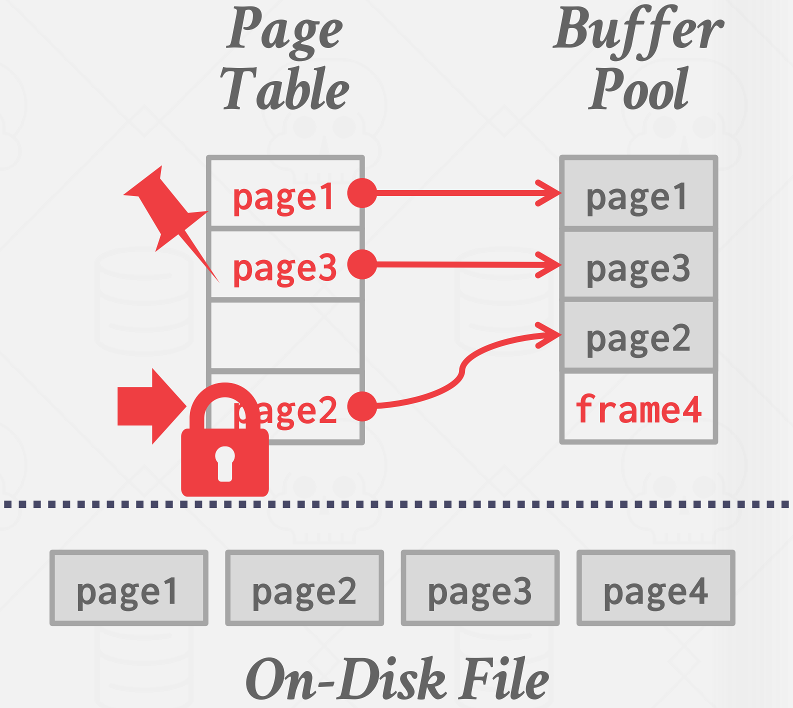
Also maintains additional meta-data per page:
→ **Dirty Flag**
→ **Pin/Reference Counter**



BUFFER POOL META-DATA

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

Also maintains additional meta-data per page:
→ **Dirty Flag**
→ **Pin/Reference Counter**



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

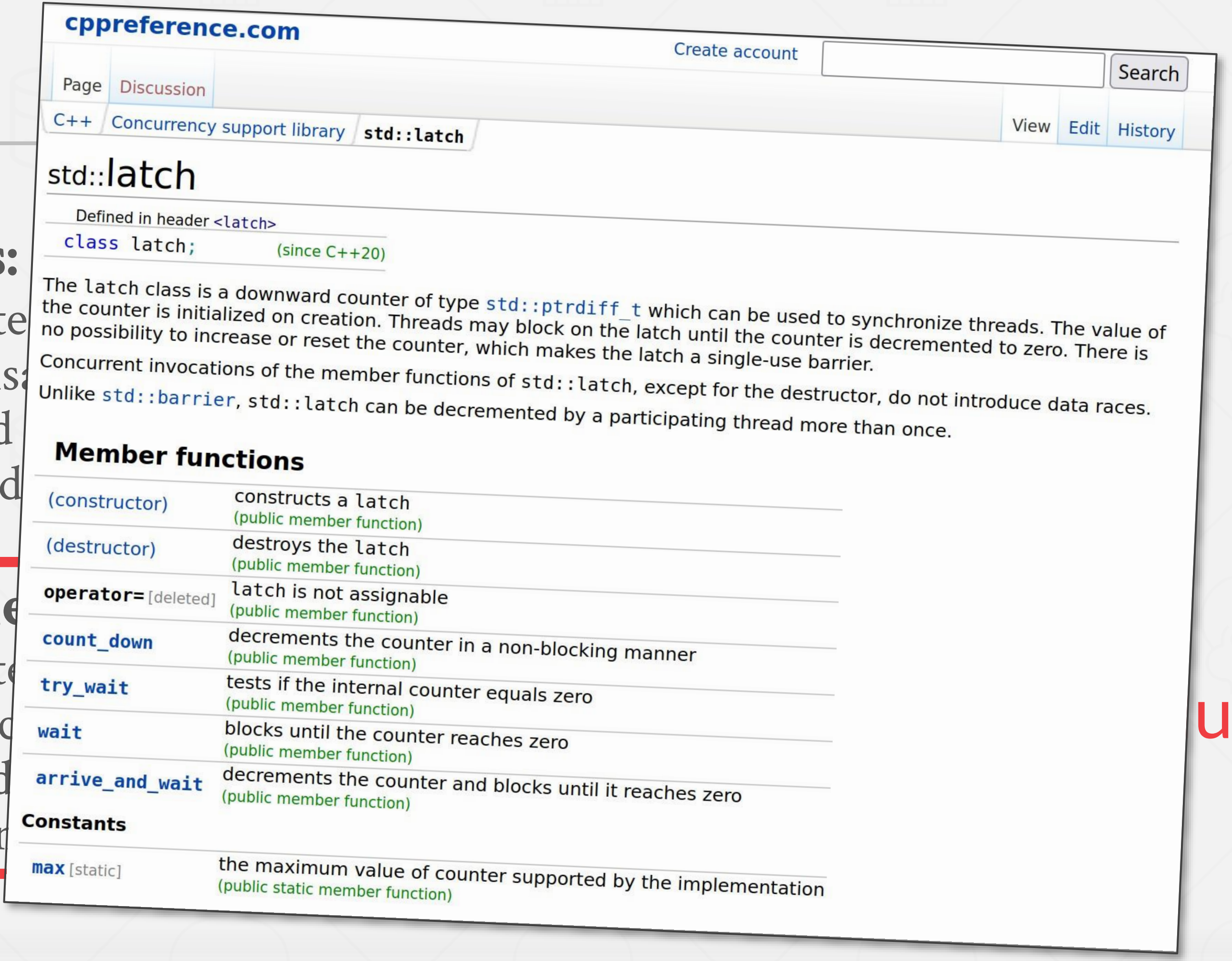
Locks:

→ Prote
trans
→ Held
→ Need

Latche

→ Prote
struc
→ Held
→ Do n

utex



cppreference.com

Create account | Search

Page | Discussion

C++ | Concurrency support library | std::latch

View | Edit | History

std::latch

Defined in header <latch>

```
class latch;
```
(since C++20)

The `latch` class is a downward counter of type `std::ptrdiff_t` which can be used to synchronize threads. The value of the counter is initialized on creation. Threads may block on the latch until the counter is decremented to zero. There is no possibility to increase or reset the counter, which makes the latch a single-use barrier.

Concurrent invocations of the member functions of `std::latch`, except for the destructor, do not introduce data races.

Unlike `std::barrier`, `std::latch` can be decremented by a participating thread more than once.

Member functions

(constructor)	constructs a `latch` (public member function)
(destructor)	destroys the `latch` (public member function)
operator= [deleted]	`latch` is not assignable (public member function)
count_down	decrements the counter in a non-blocking manner (public member function)
try_wait	tests if the internal counter equals zero (public member function)
wait	blocks until the counter reaches zero (public member function)
arrive_and_wait	decrements the counter and blocks until it reaches zero (public member function)

Constants

max [static]	the maximum value of counter supported by the implementation (public static member function)

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

DB2

```
CREATE BUFFERPOOL custom_pool
  ↳SIZE 250 PAGESIZE 8k;

CREATE TABLESPACE custom_tablespace
  ↳PAGESIZE 8k BUFFERPOOL custom_pool;

CREATE TABLE new_table
  ↳TABLESPACE custom_tablespace ( ... );
```

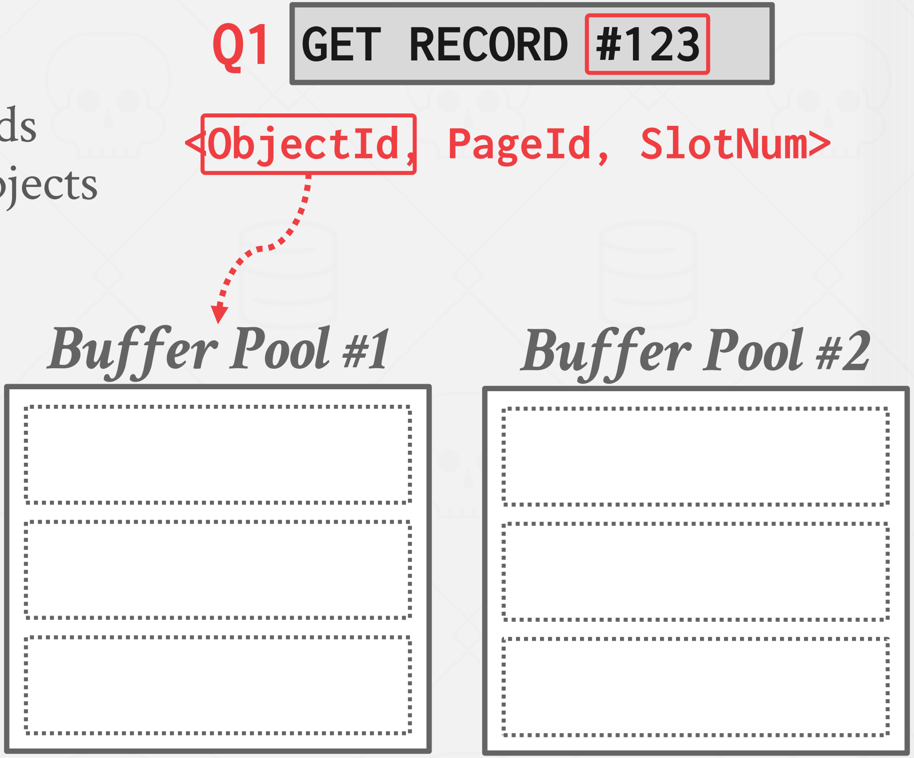
MULTIPLE BUFFER POOLS

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.



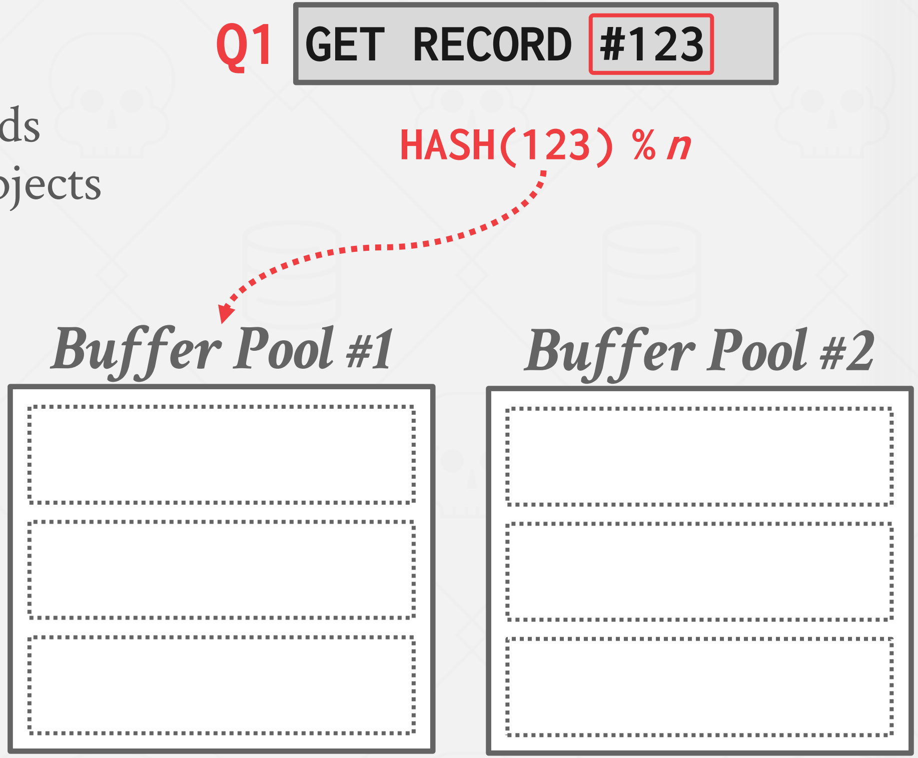
MULTIPLE BUFFER POOLS

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

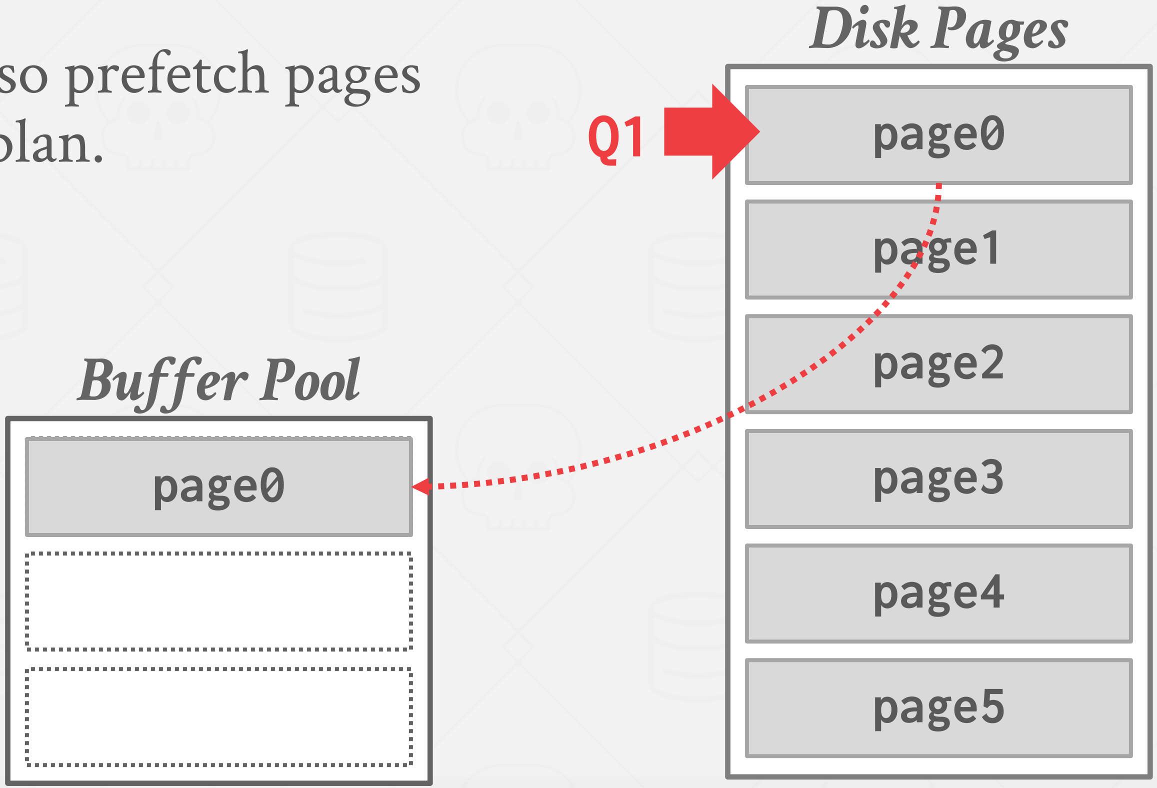
→ Hash the page id to select which buffer pool to access.



PRE-FETCHING

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

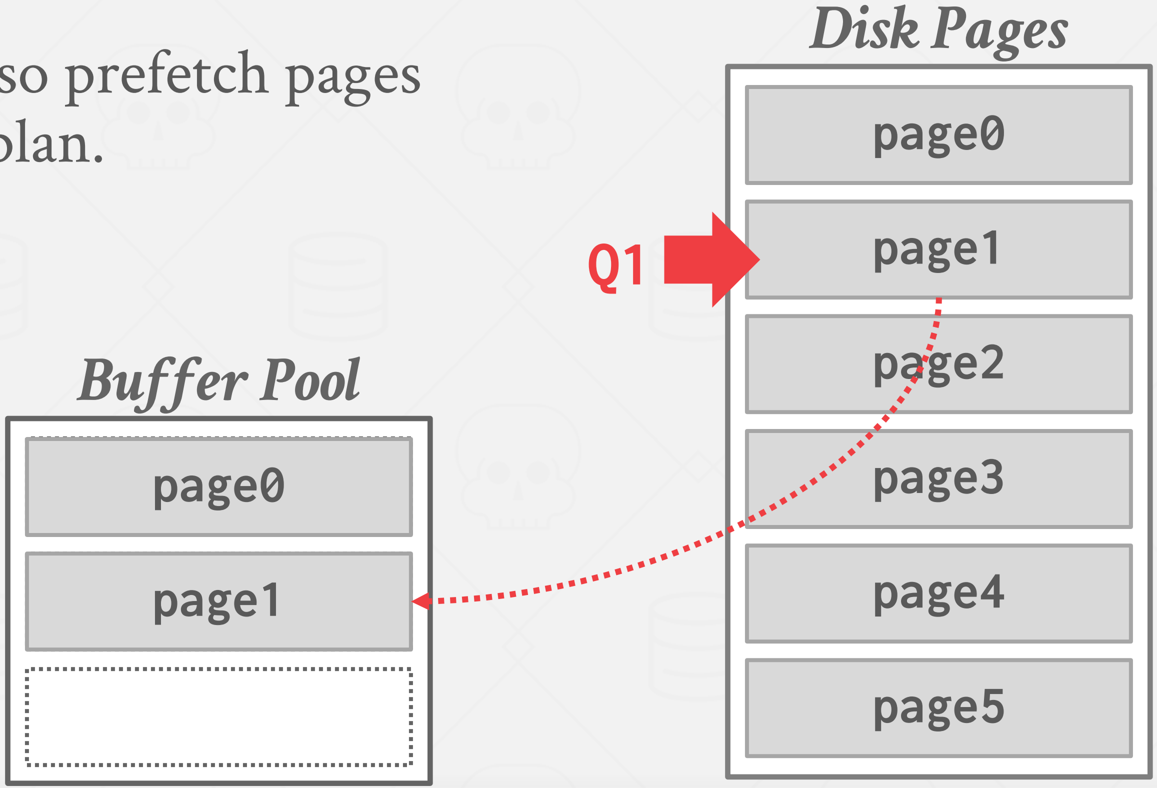
→ Sequential Scans
→ Index Scans



PRE-FETCHING

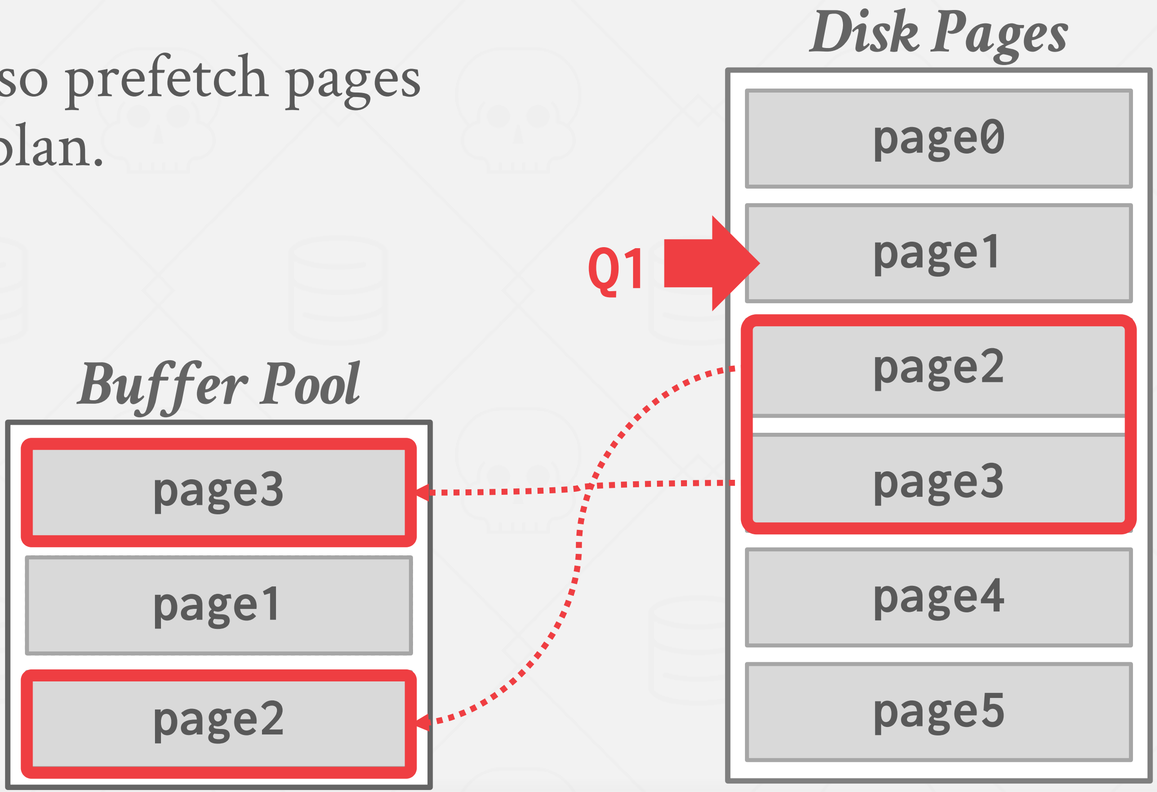
The DBMS can also prefetch pages based on a query plan.

→ Sequential Scans
→ Index Scans



PRE-FETCHING

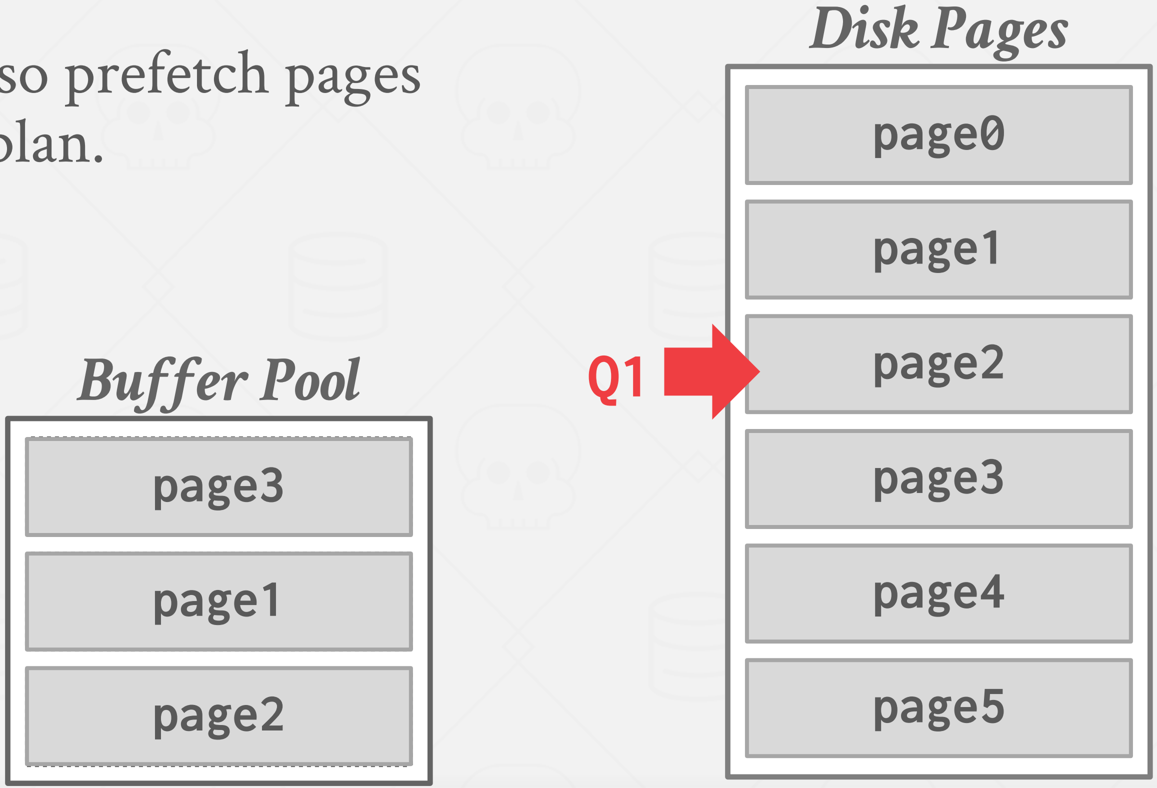
The DBMS can also prefetch pages based on a query plan.
→ Sequential Scans
→ Index Scans



PRE-FETCHING

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

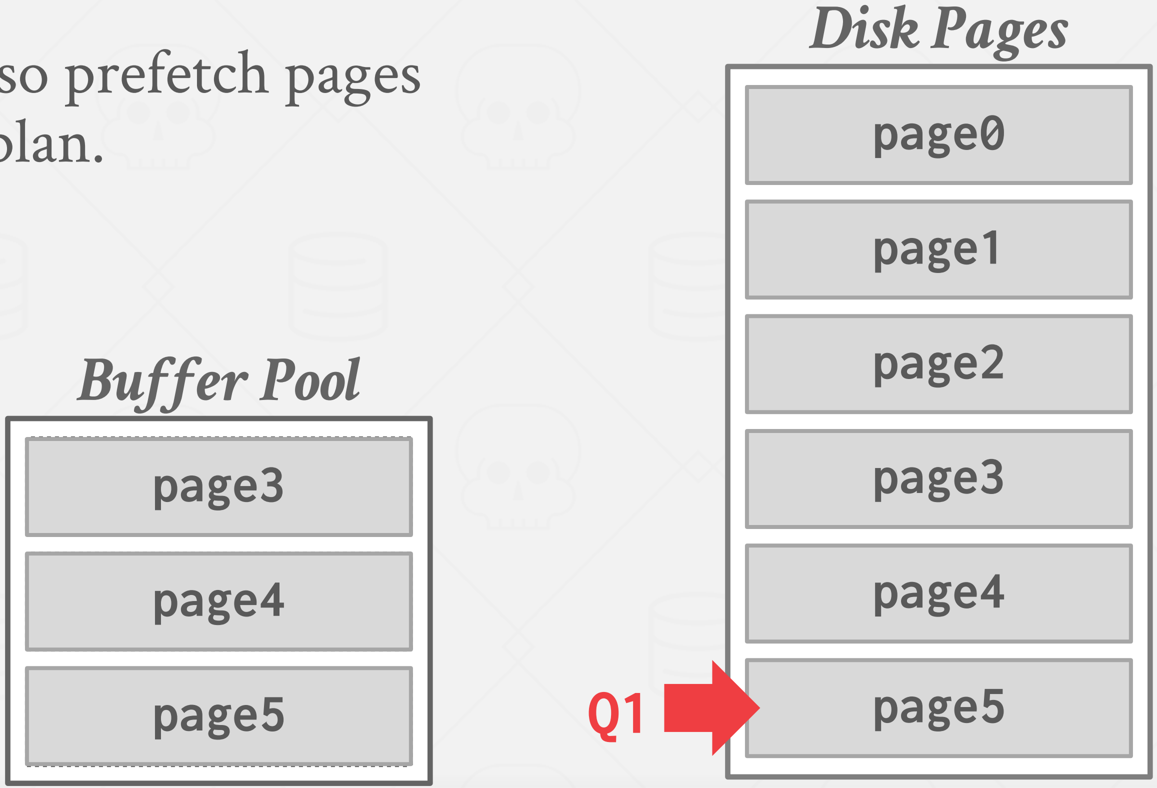
→ Sequential Scans
→ Index Scans



PRE-FETCHING

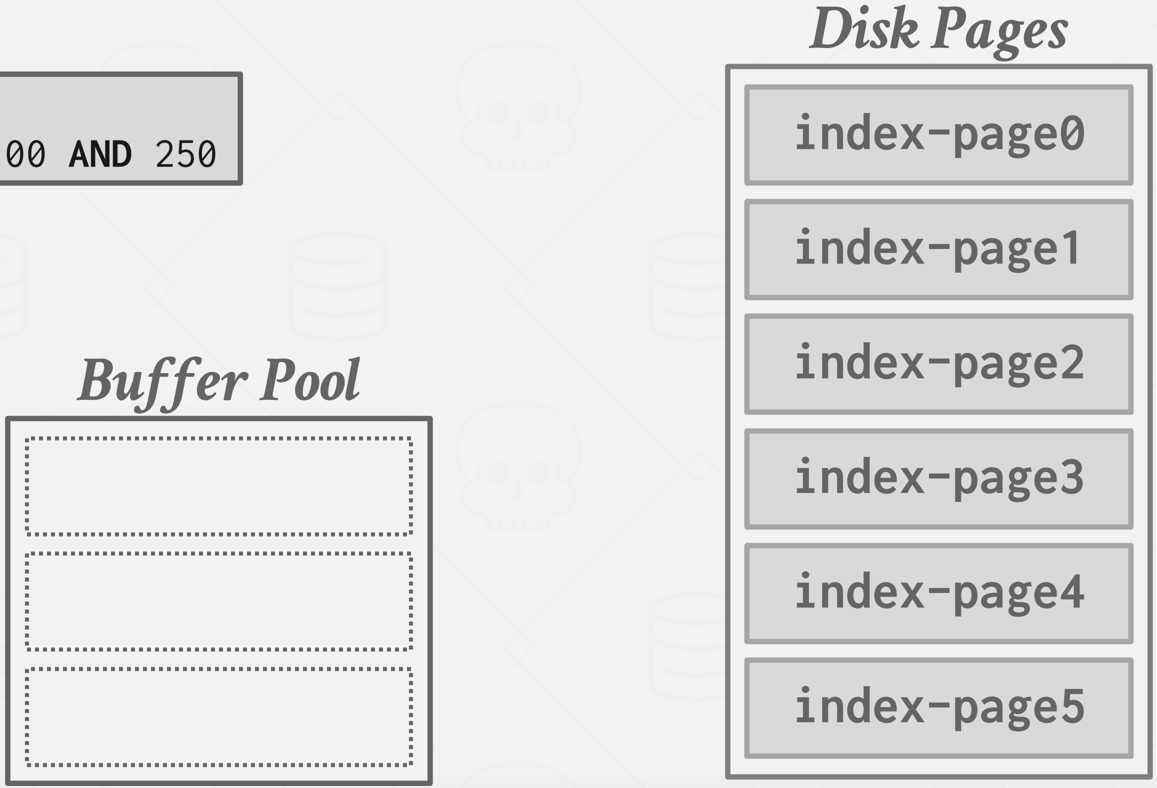
The DBMS can also prefetch pages based on a query plan.

→ Sequential Scans
→ Index Scans

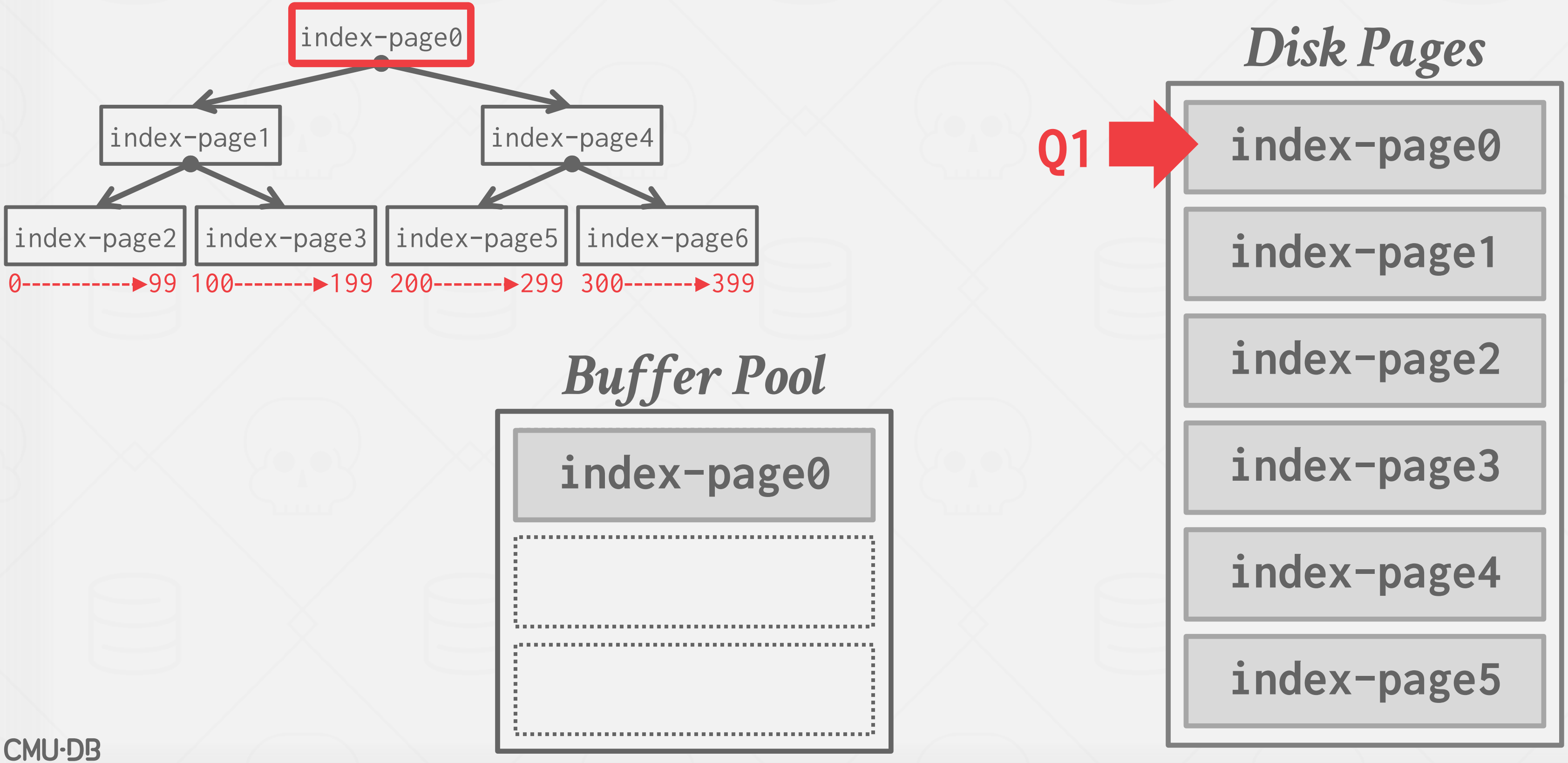


PRE-FETCHING

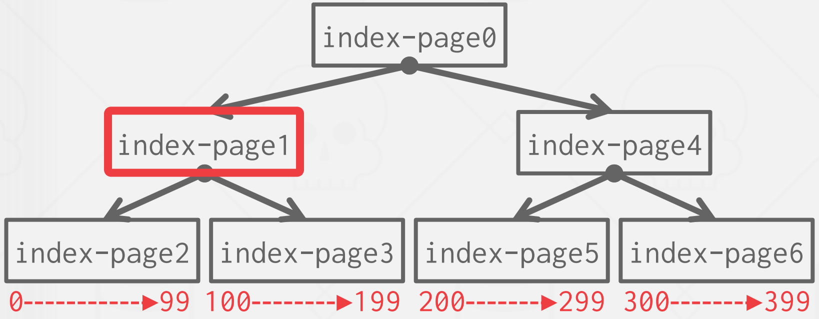
Q1
```
SELECT * FROM A
 WHERE val BETWEEN 100 AND 250
```

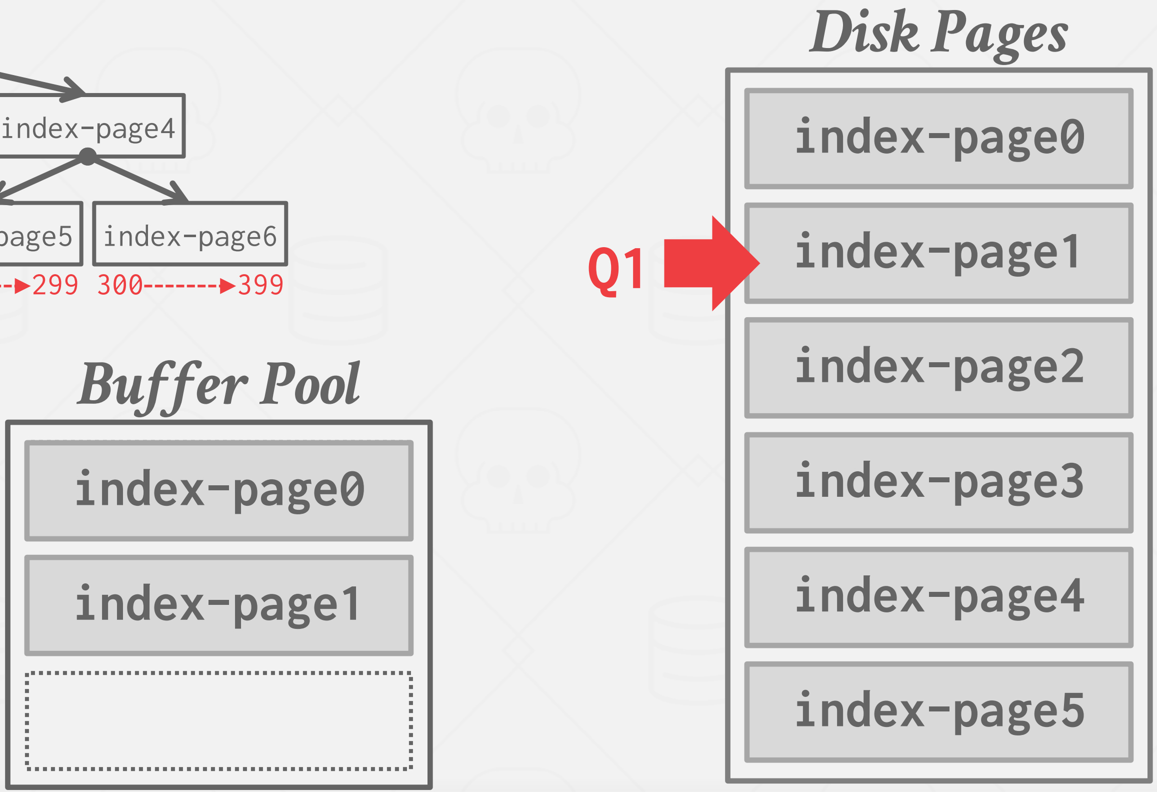


PRE-FETCHING

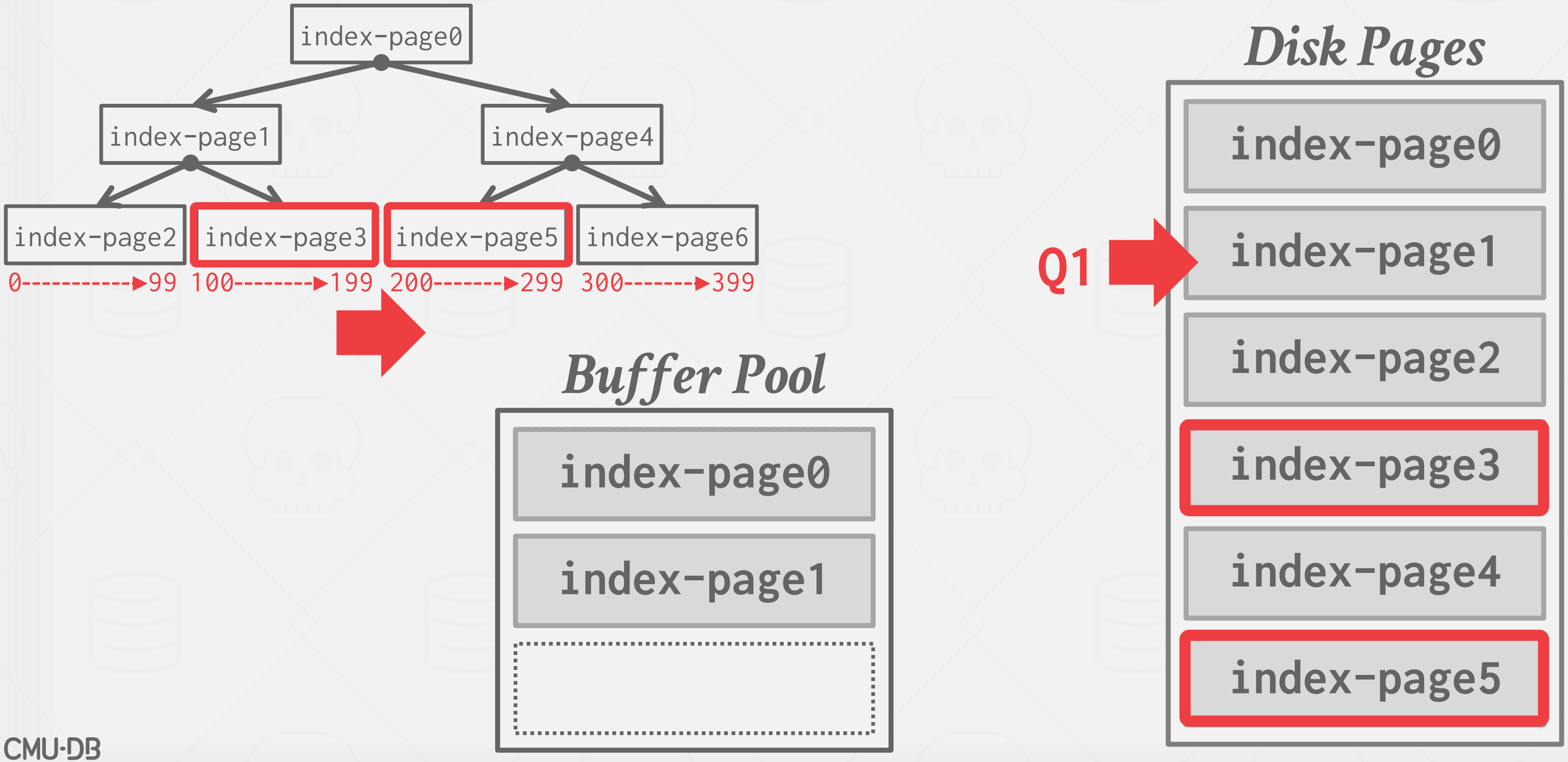


PRE-FETCHING





PRE-FETCHING



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 IBM DB2, MSSQL, and Postgres.
→ Oracle only supports cursor sharing for identical queries.

If a q
is alr
the s

Exa
→ F
→ C

Shared Scanning in Postgres

Asked 3 months ago Active 3 months ago Viewed 17 times

0

In the 11th lecture of the CMU Intro to Databases course (2020, 39:37), Andy Pavlo states that "only the high end data systems support shared buffer scanning but Postgres and MySql cannot". He does not expand and thus, I tried to find out why but couldn't find any abstracted information and wanted to ask here before I dove into the documentation. Did Andy mean that Postgres cannot support this due to its implementation, or has it simply not been implemented yet?

If it cannot be implemented, what about the Postgres design prevents it from doing so? How can this be circumvented? If it is possible, what is preventing the implementation today?

Also, do you know why everyone says that Andy smells so bad? I've heard that he smells like old Arby's beef-and-cheddar sandwiches that have been left out in the sun for too long.

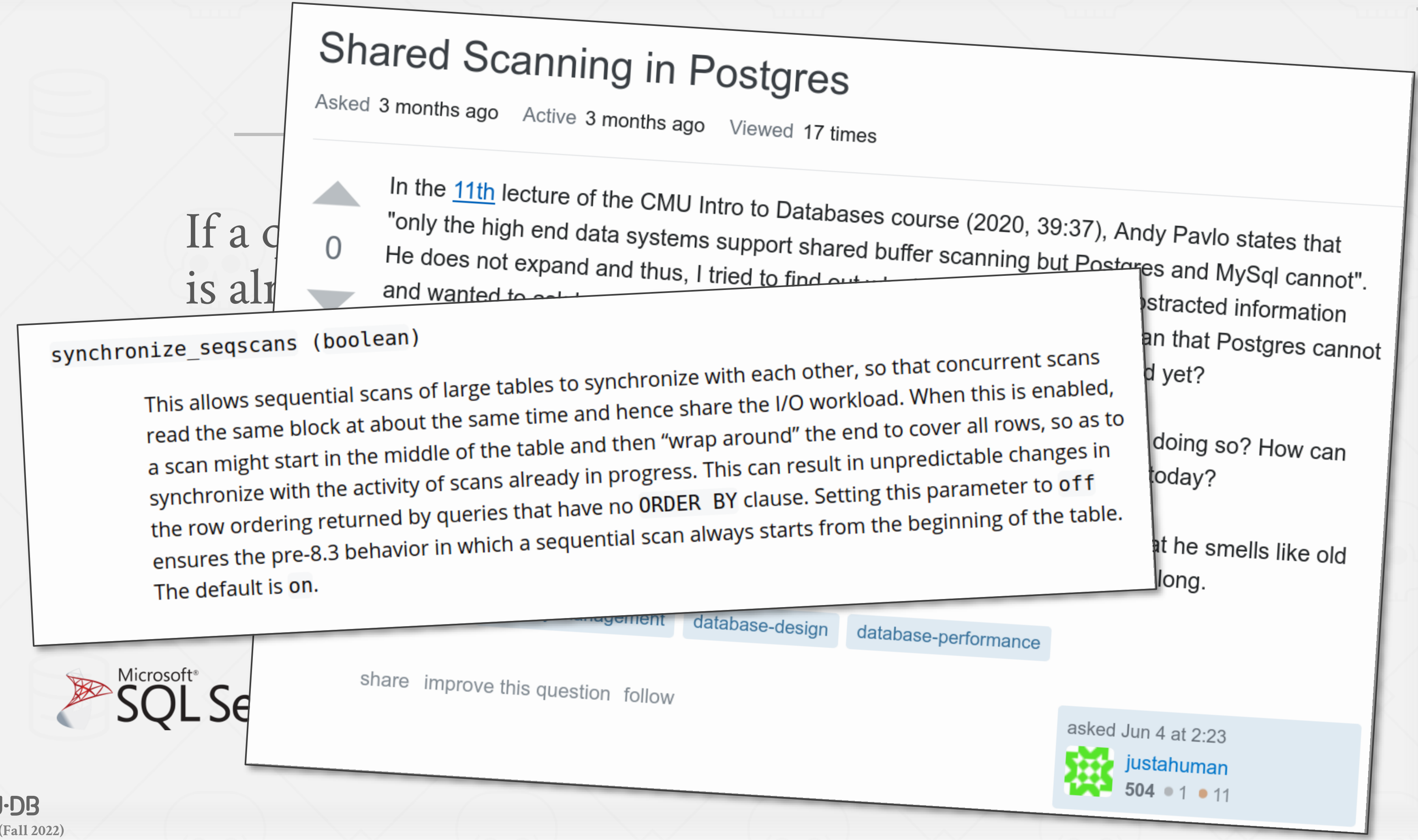
postgresql memory-management database-design database-performance

share improve this question follow

asked Jun 4 at 2:23
justahuman
504 ● 1 ● 11

Microsoft SQL Se

If a q
is alr


Shared Scanning in Postgres

Asked 3 months ago Active 3 months ago Viewed 17 times

0

In the 11th lecture of the CMU Intro to Databases course (2020, 39:37), Andy Pavlo states that "only the high end data systems support shared buffer scanning but Postgres and MySql cannot". He does not expand and thus, I tried to find out ... and wanted to ...

...stracted information
...an that Postgres cannot
...d yet?

...doing so? How can
...today?

...at he smells like old
...long.

...anagement database-design database-performance

share improve this question follow

asked Jun 4 at 2:23
justahuman
504 ● 1 ● 11

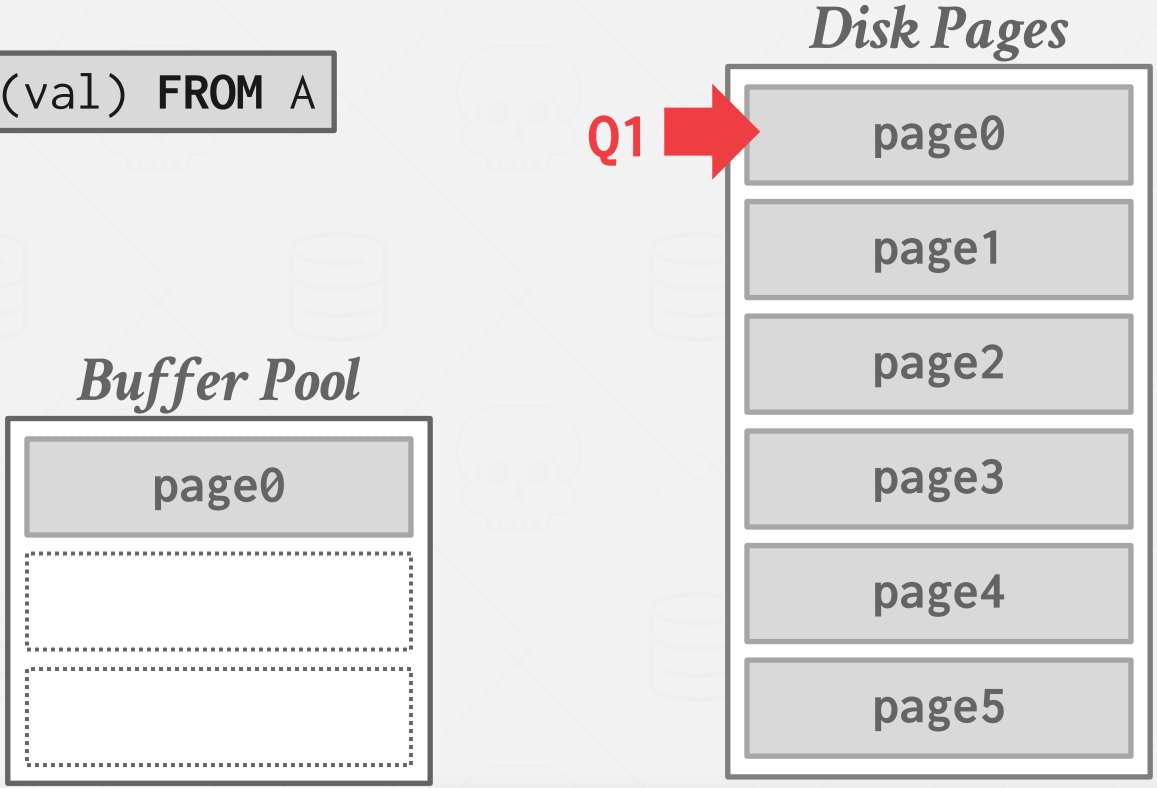
`synchronize_seqscans (boolean)`

This allows sequential scans of large tables to synchronize with each other, so that concurrent scans read the same block at about the same time and hence share the I/O workload. When this is enabled, a scan might start in the middle of the table and then "wrap around" the end to cover all rows, so as to synchronize with the activity of scans already in progress. This can result in unpredictable changes in the row ordering returned by queries that have no `ORDER BY` clause. Setting this parameter to `off` ensures the pre-8.3 behavior in which a sequential scan always starts from the beginning of the table. The default is `on`.

Microsoft SQL Se

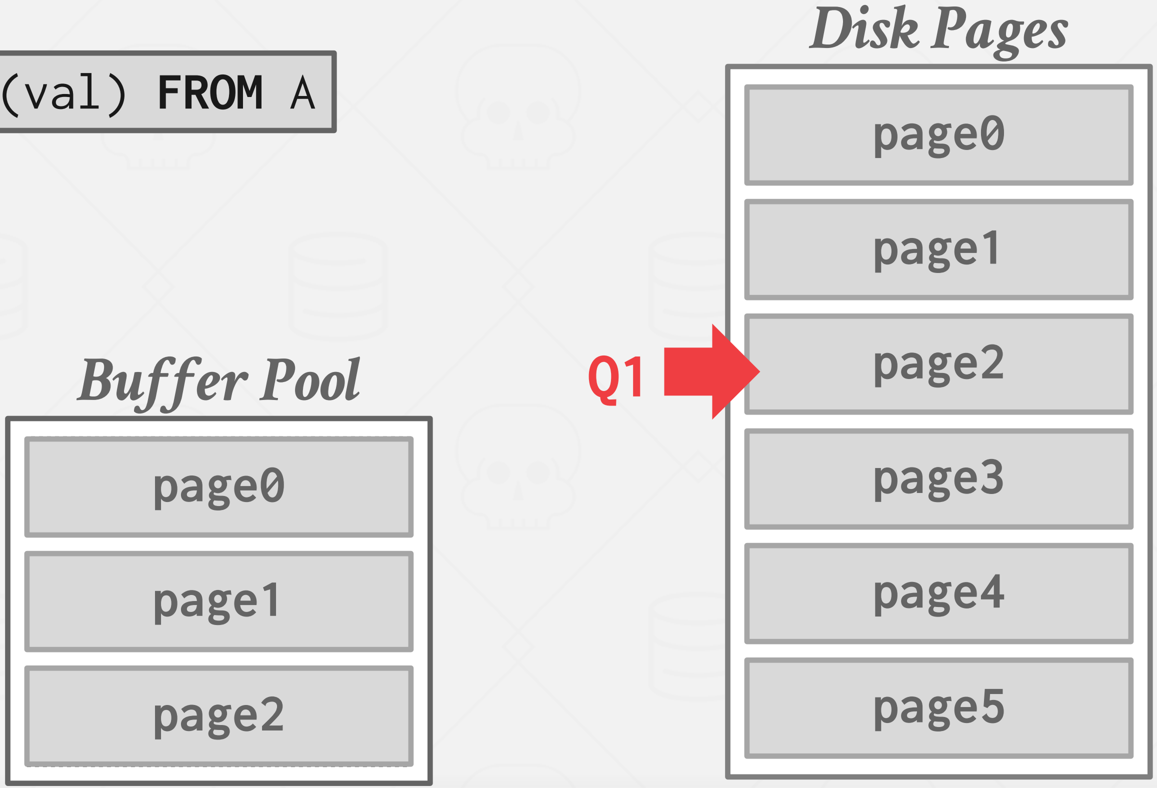
SCAN SHARING

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



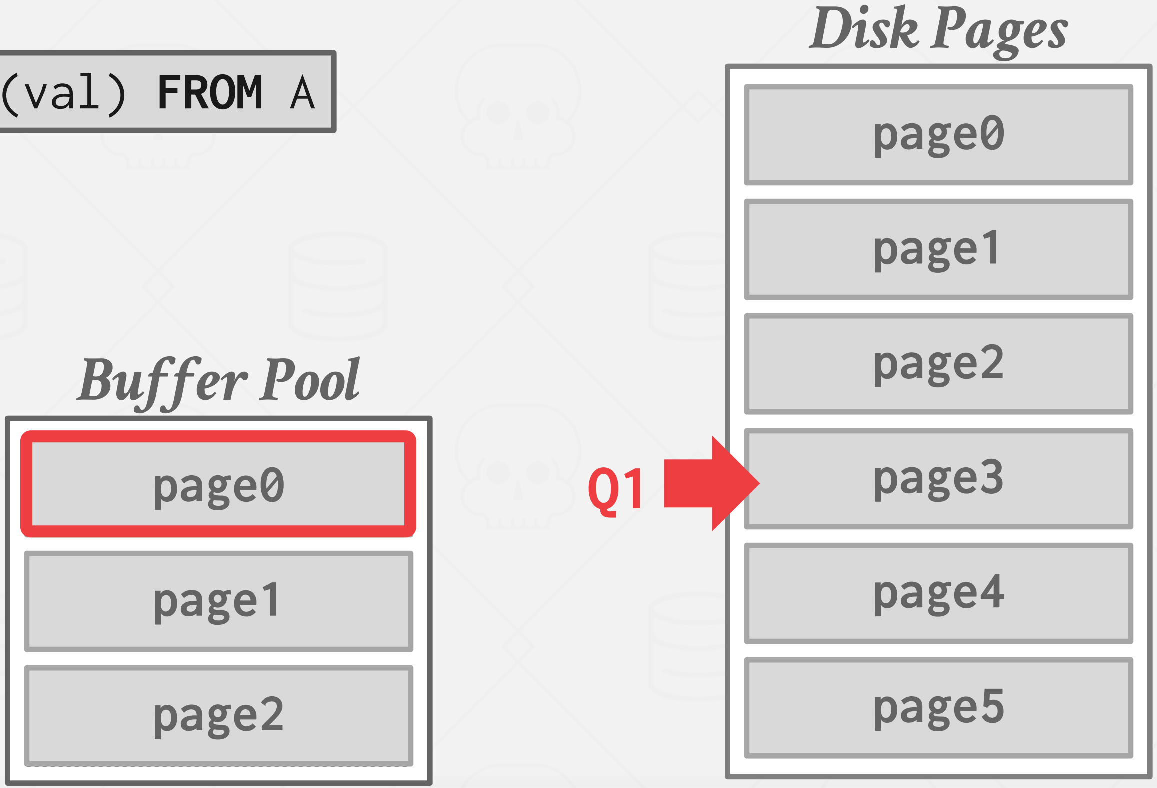
SCAN SHARING

Q1 `SELECT SUM(val) FROM A`



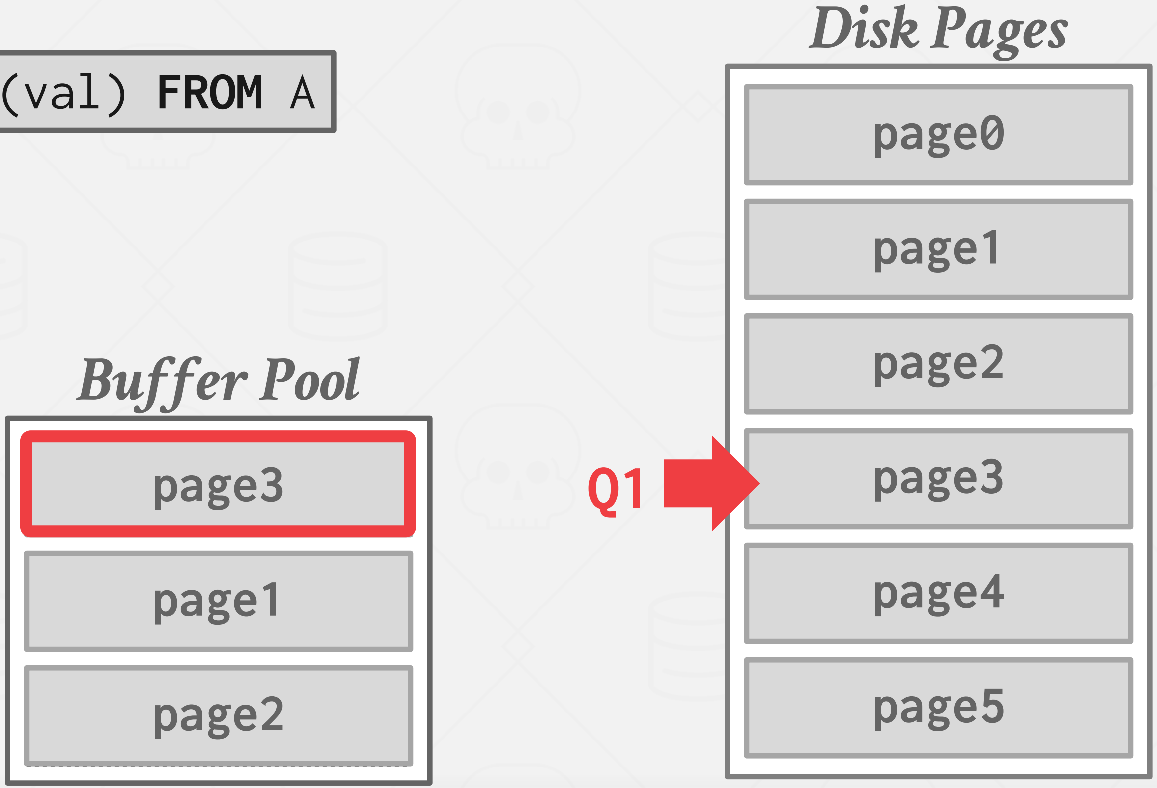
SCAN SHARING

Q1 `SELECT SUM(val) FROM A`



SCAN SHARING

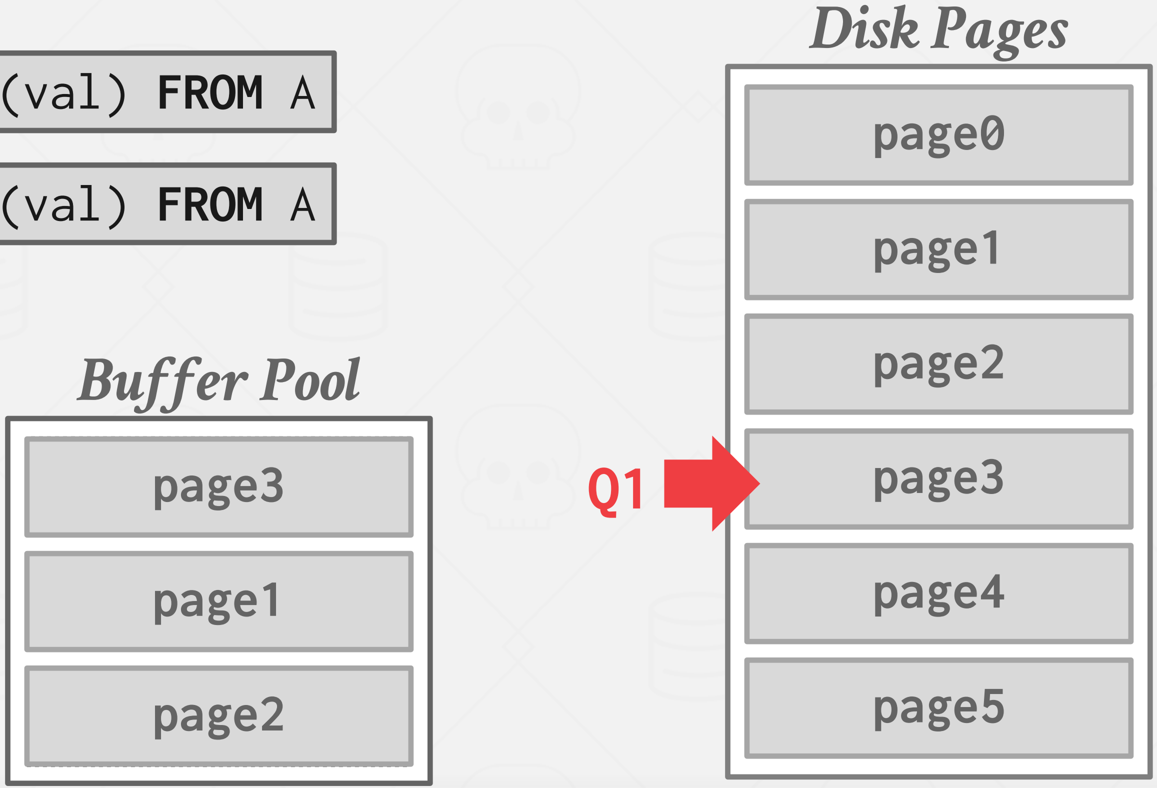
Q1 `SELECT SUM(val) FROM A`



SCAN SHARING

Q1 `SELECT SUM(val) FROM A`

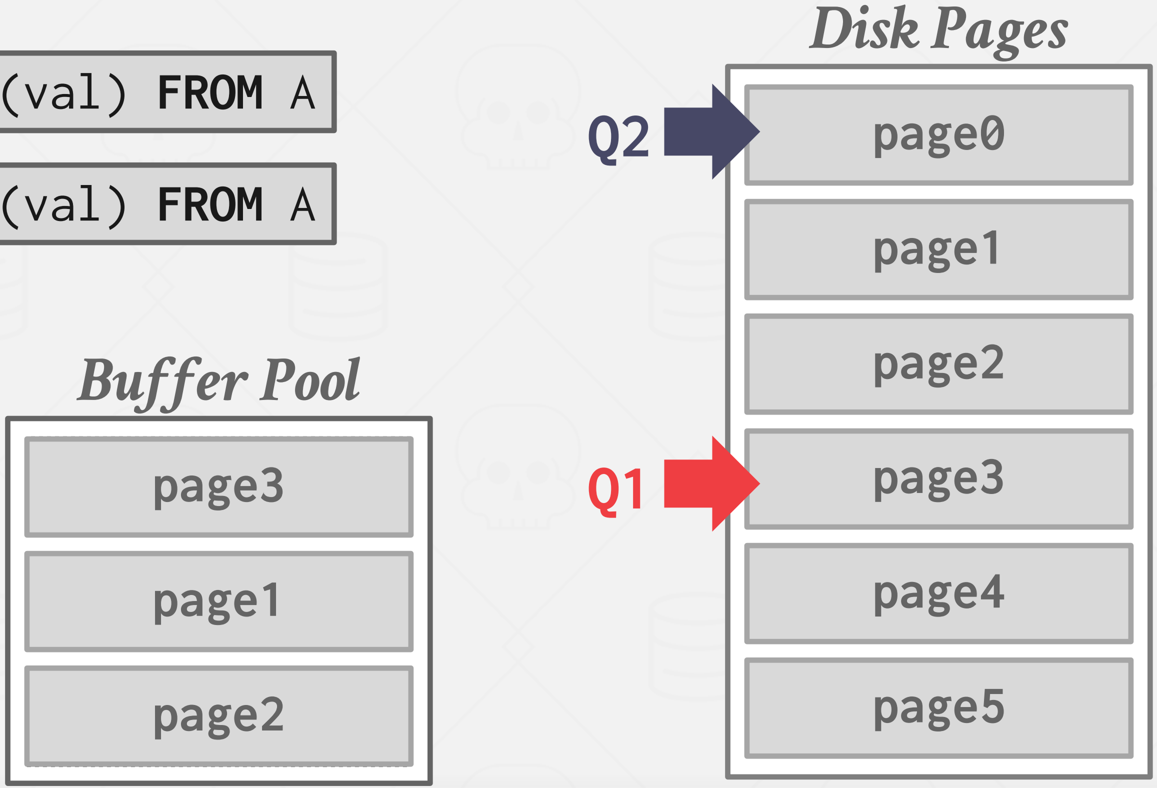
Q2 `SELECT AVG(val) FROM A`



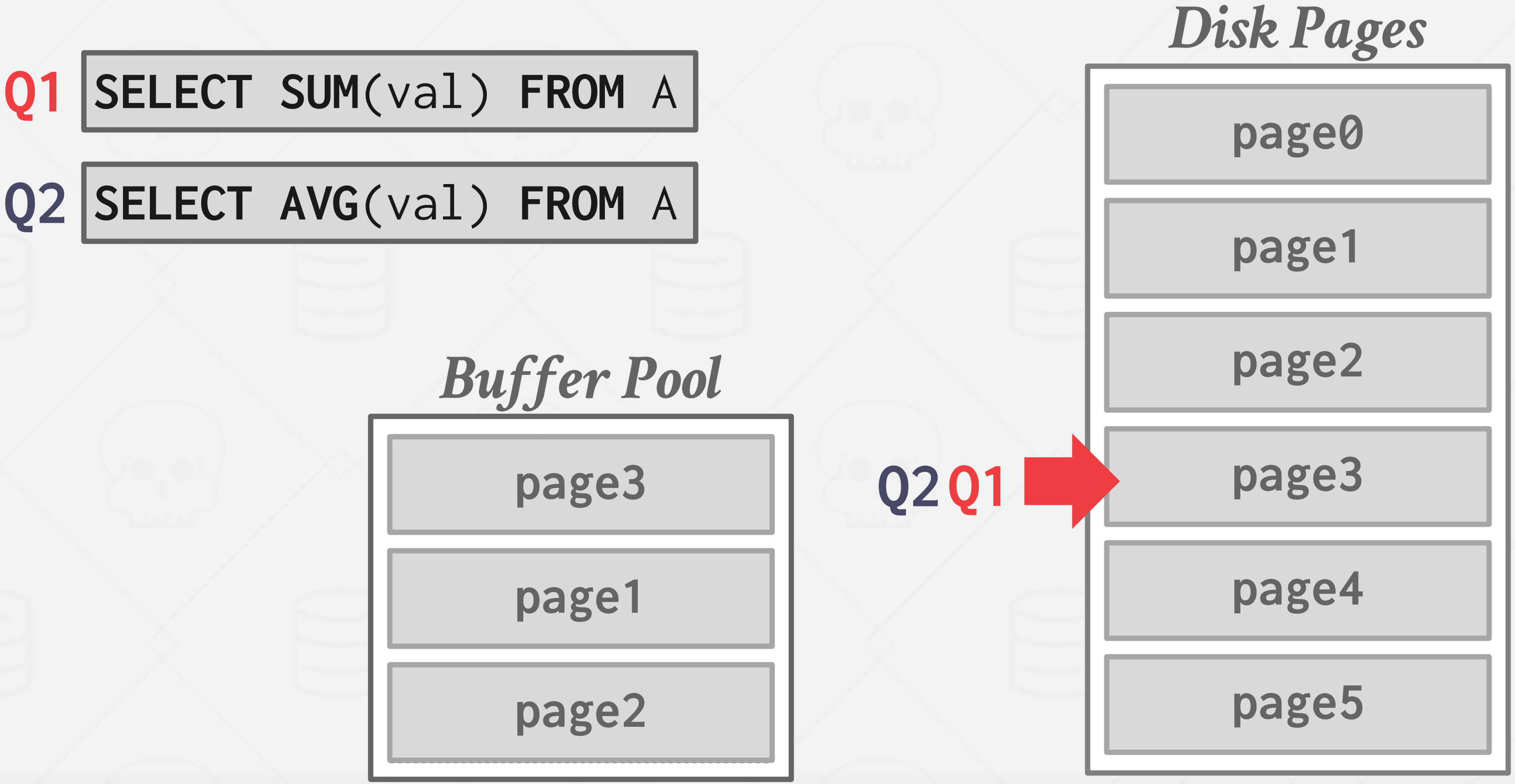
SCAN SHARING

Q1 `SELECT SUM(val) FROM A`

Q2 `SELECT AVG(val) FROM A`



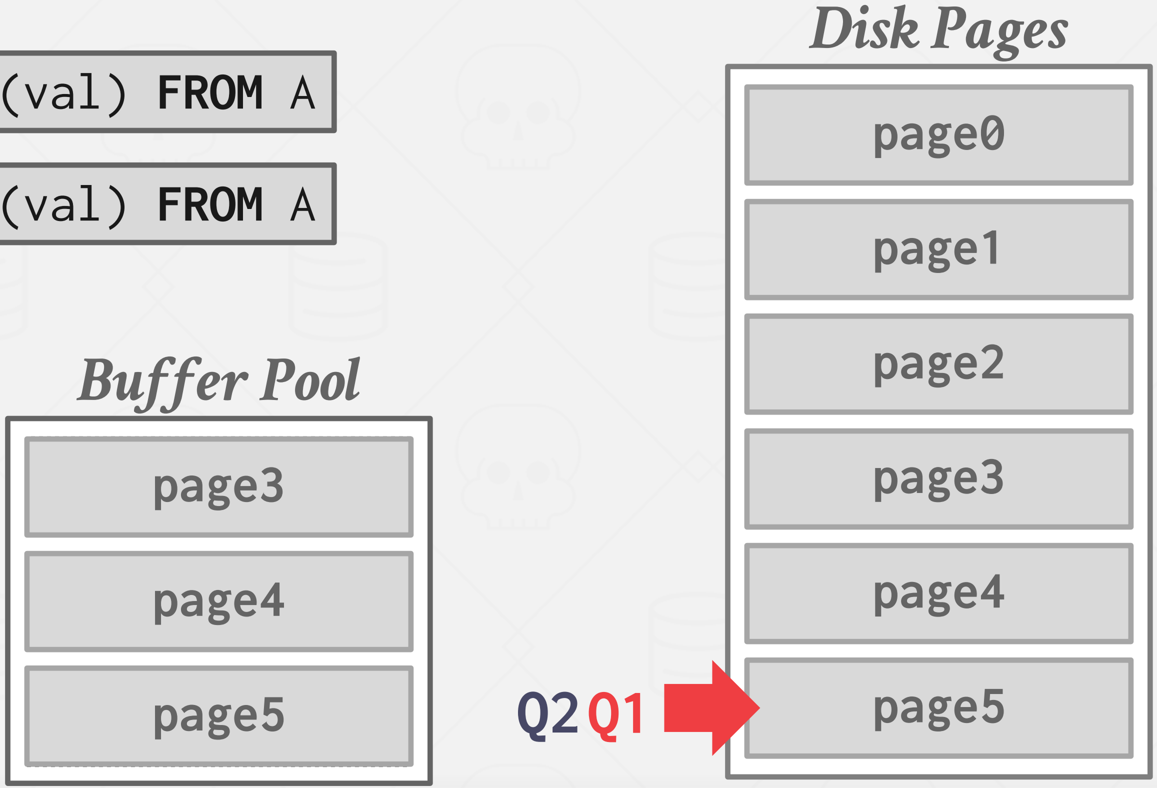
SCAN SHARING



SCAN SHARING

Q1 `SELECT SUM(val) FROM A`

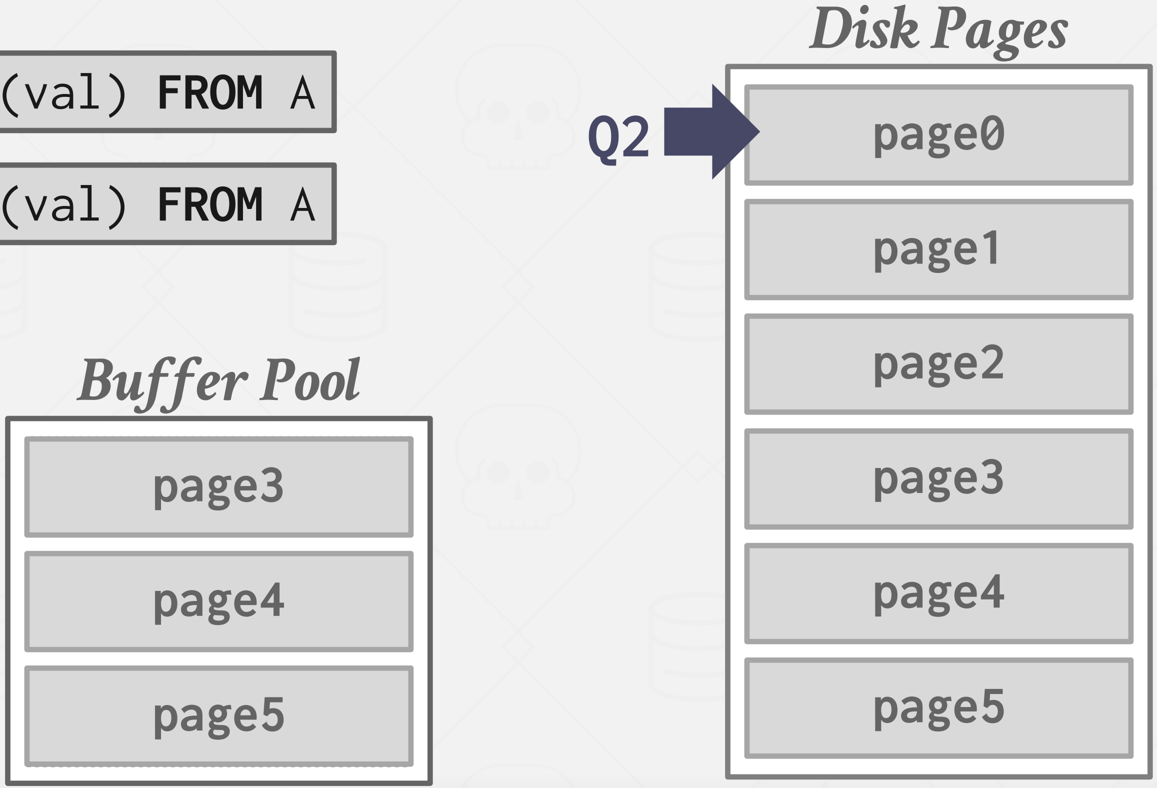
Q2 `SELECT AVG(val) FROM A`



SCAN SHARING

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

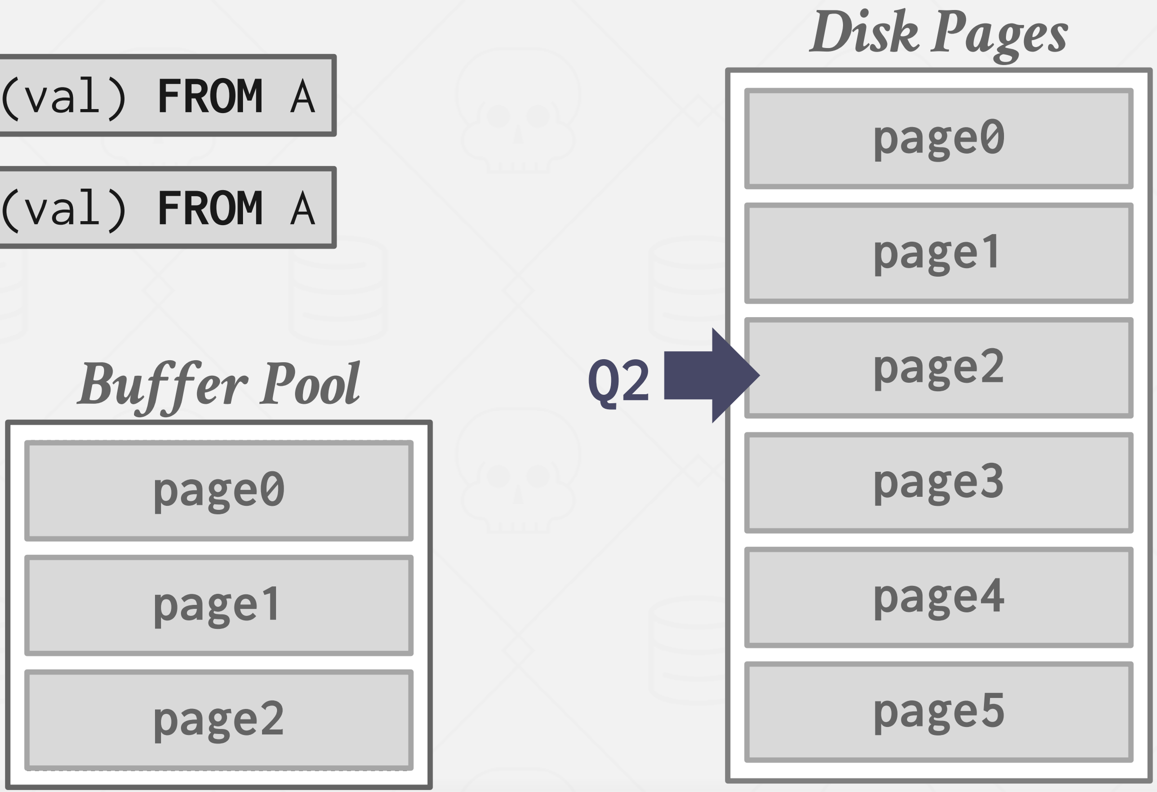
Q2
```
SELECT AVG(val) FROM A
```



SCAN SHARING

Q1 `SELECT SUM(val) FROM A`

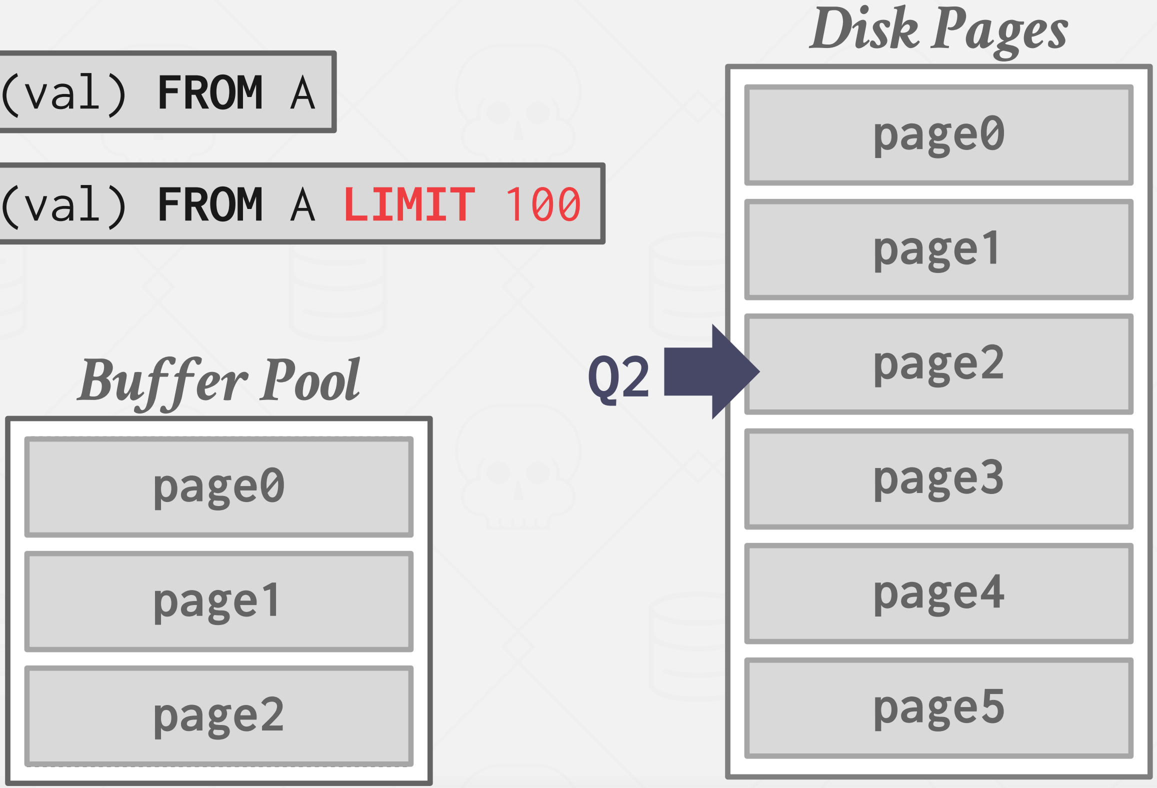
Q2 `SELECT AVG(val) FROM A`



SCAN SHARING

Q1 `SELECT SUM(val) FROM A`

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



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.

OS PAGE CACHE

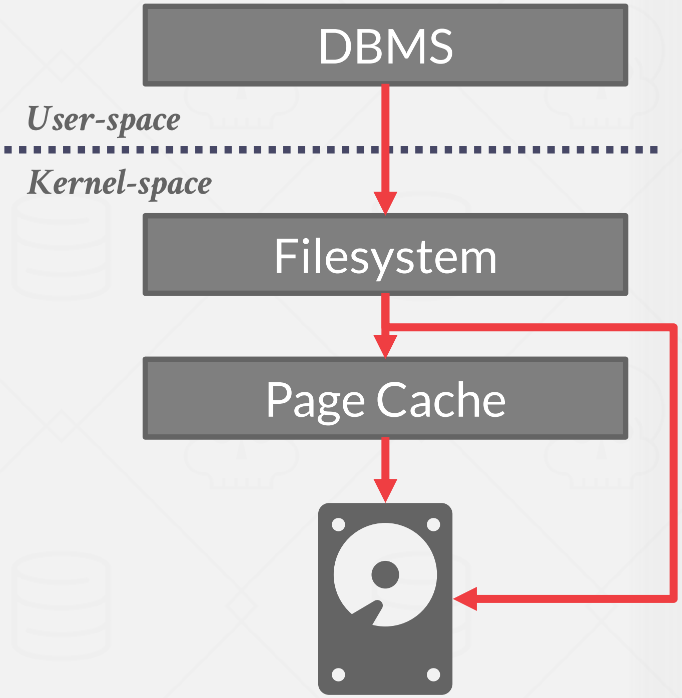
Most disk operations go through the OS API. Unless the DBMS tells it not to, the OS maintains its own filesystem cache (aka page cache, buffer cache).

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 you tell it not to, the OS maintains its own filesystem cache (i.e., the page cache).

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

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

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.

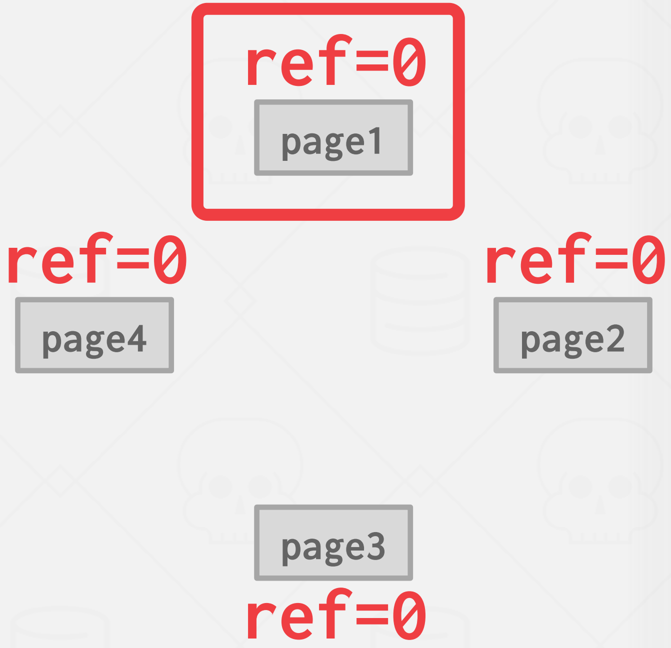
When the DBMS needs to evict a page, select the one with the oldest timestamp.

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

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.

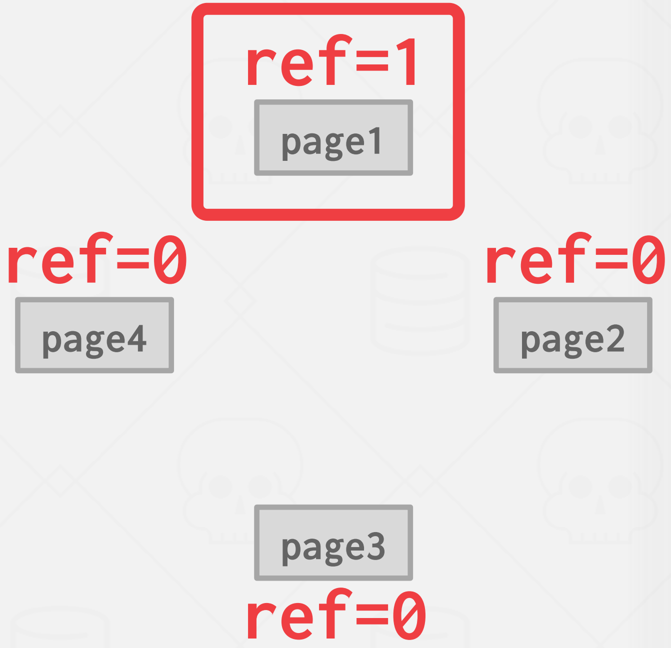
Organize the pages in a circular buffer with a "clock hand":
→ Upon sweeping, check if a page's bit is set to 1.
→ If yes, set to zero. If no, then evict.



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.

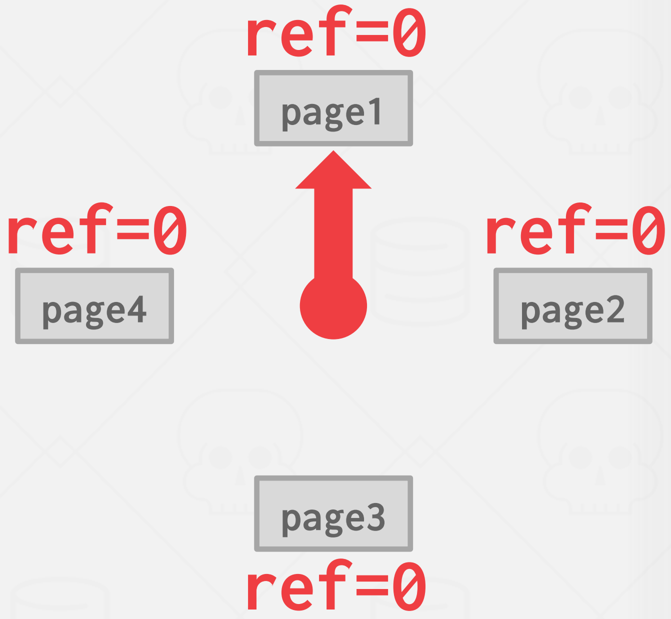
Organize the pages in a circular buffer with a "clock hand":
→ Upon sweeping, check if a page's bit is set to 1.
→ If yes, set to zero. If no, then evict.



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.

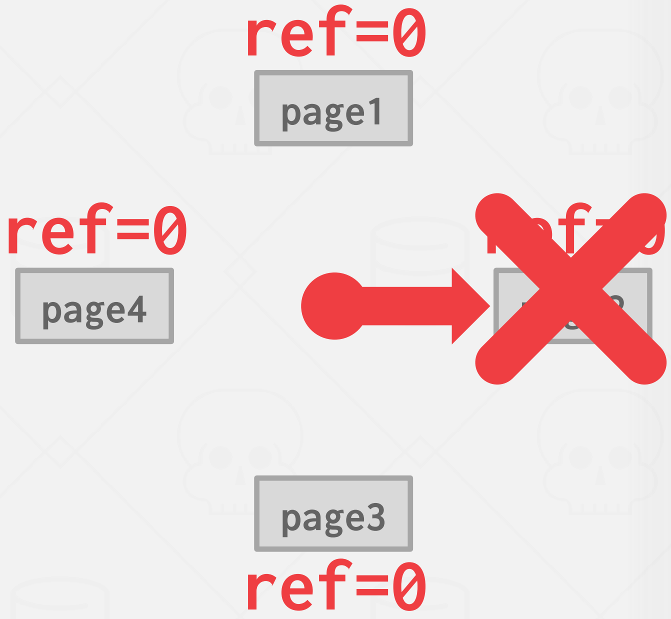
→ Each page has a reference bit.

→ 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.

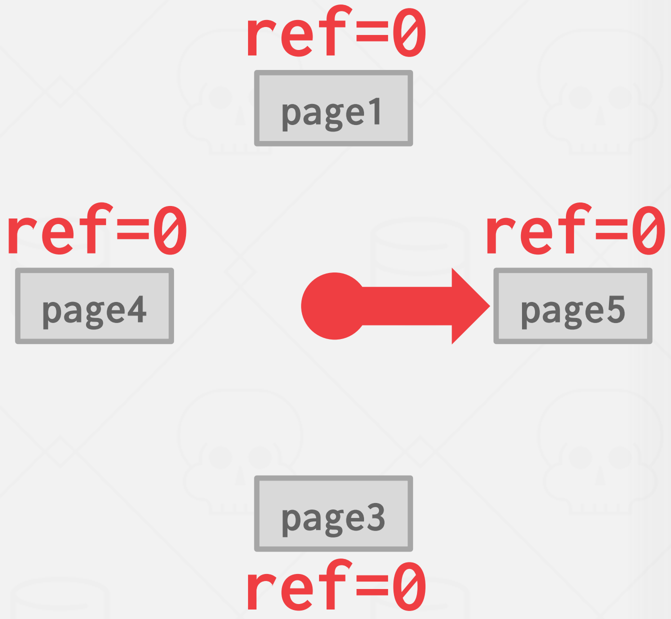
→ If yes, set to zero. If no, then evict.



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.

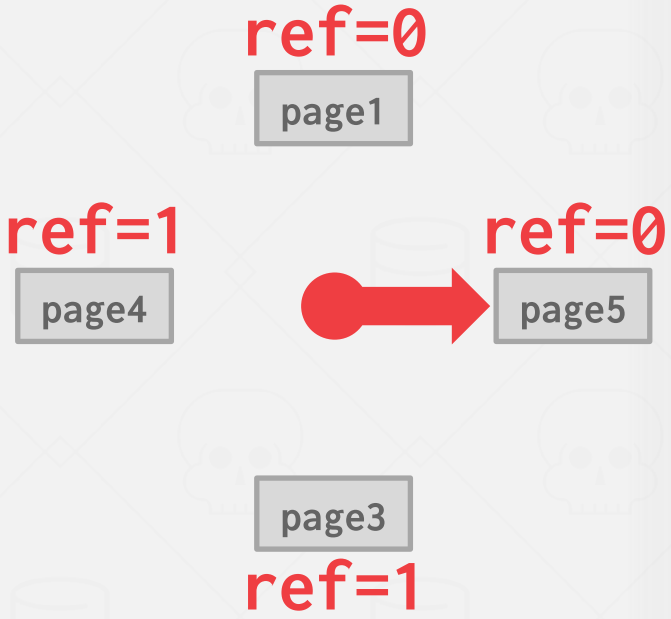
→ Each page has a reference bit.

→ 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.

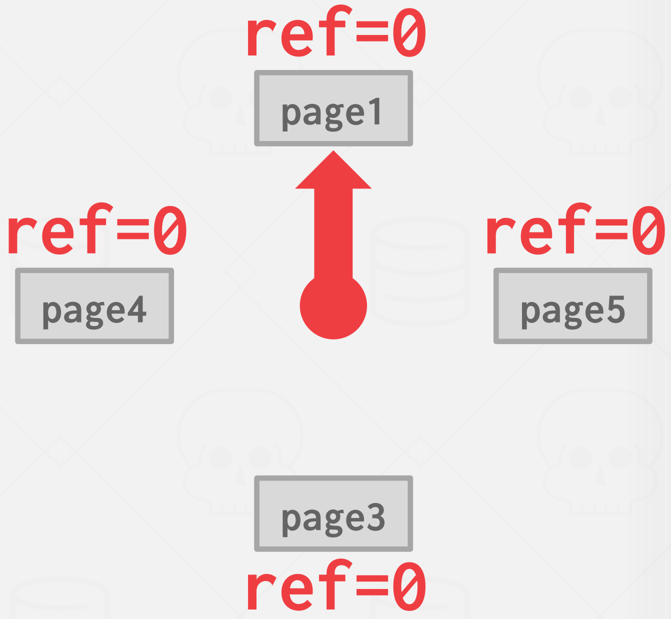
→ If yes, set to zero. If no, then evict.



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.

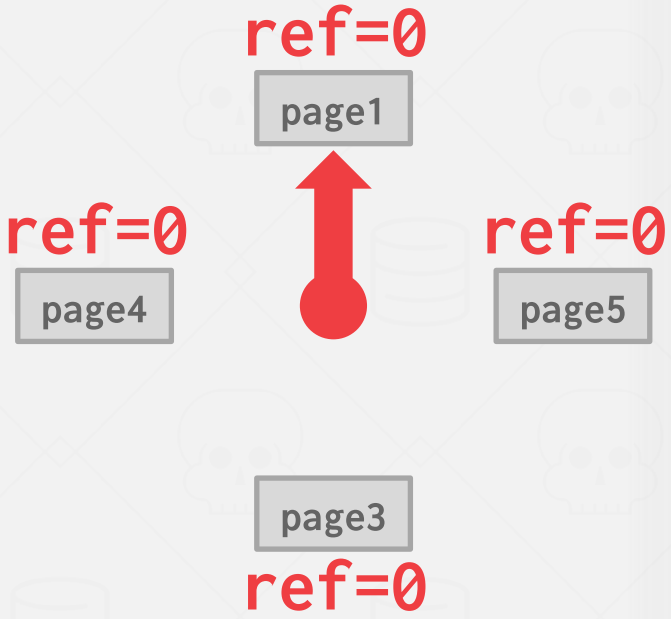
Organize the pages in a circular buffer with a "clock hand":
→ Upon sweeping, check if a page's bit is set to 1.
→ If yes, set to zero. If no, then evict.



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.



PROBLEMS

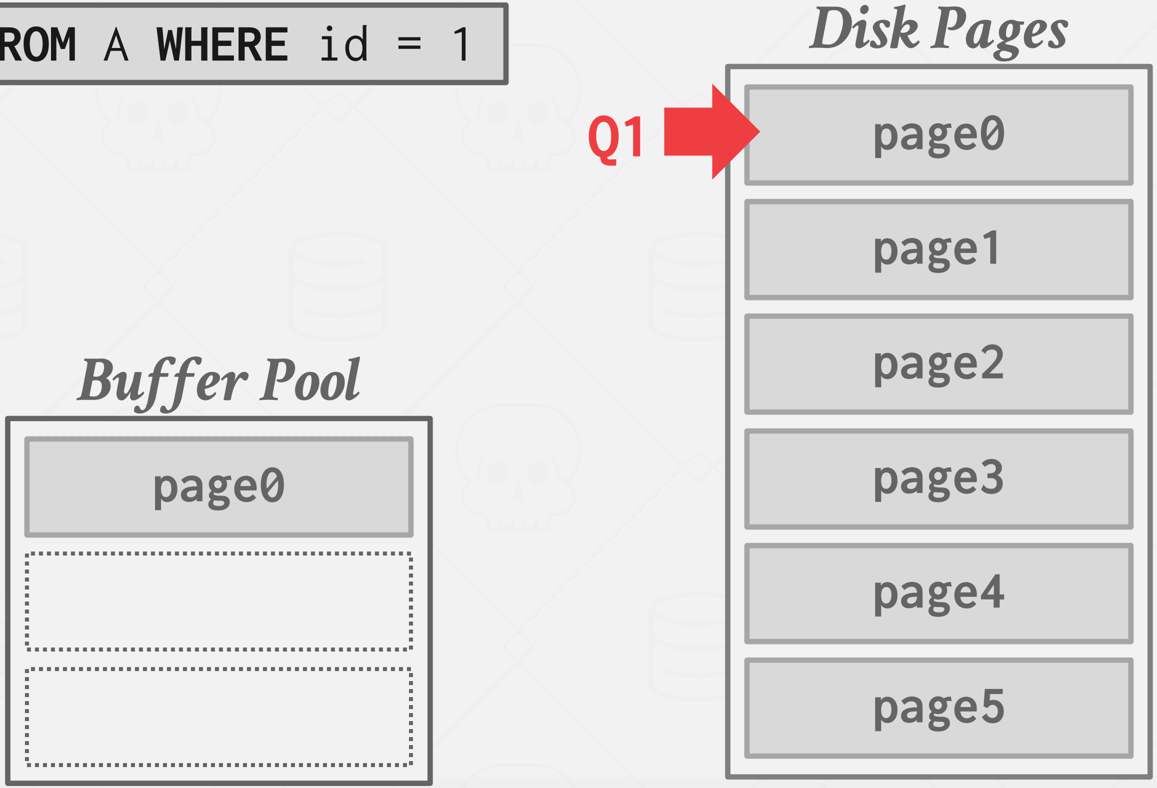
LRU and CLOCK replacement policies are susceptible to sequential flooding.

→ 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 some workloads the most recently used page is the most unneeded page.

SEQUENTIAL FLOODING

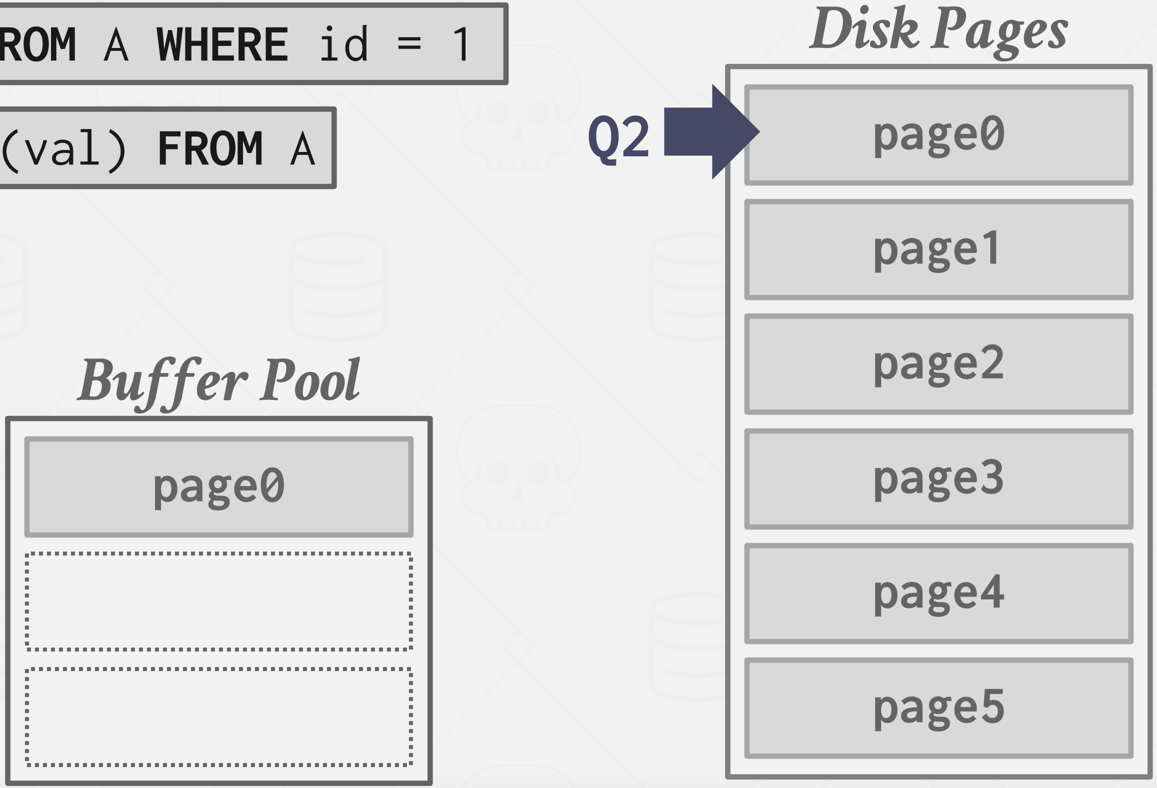
Q1 `SELECT * FROM A WHERE id = 1`



SEQUENTIAL FLOODING

Q1 `SELECT * FROM A WHERE id = 1`

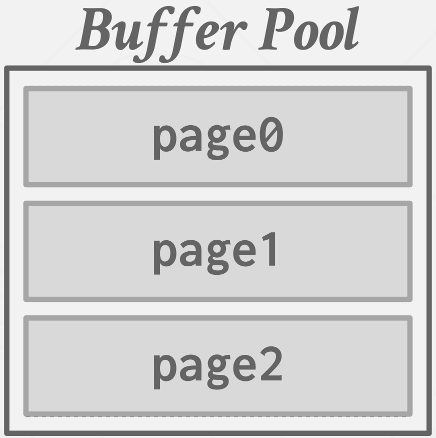
Q2 `SELECT AVG(val) FROM A`

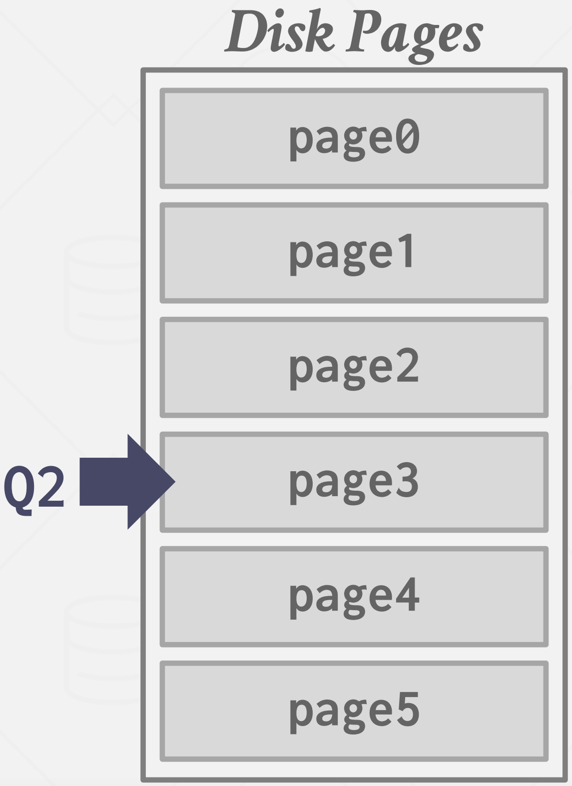


SEQUENTIAL FLOODING

Q1 `SELECT * FROM A WHERE id = 1`

Q2 `SELECT AVG(val) FROM A`

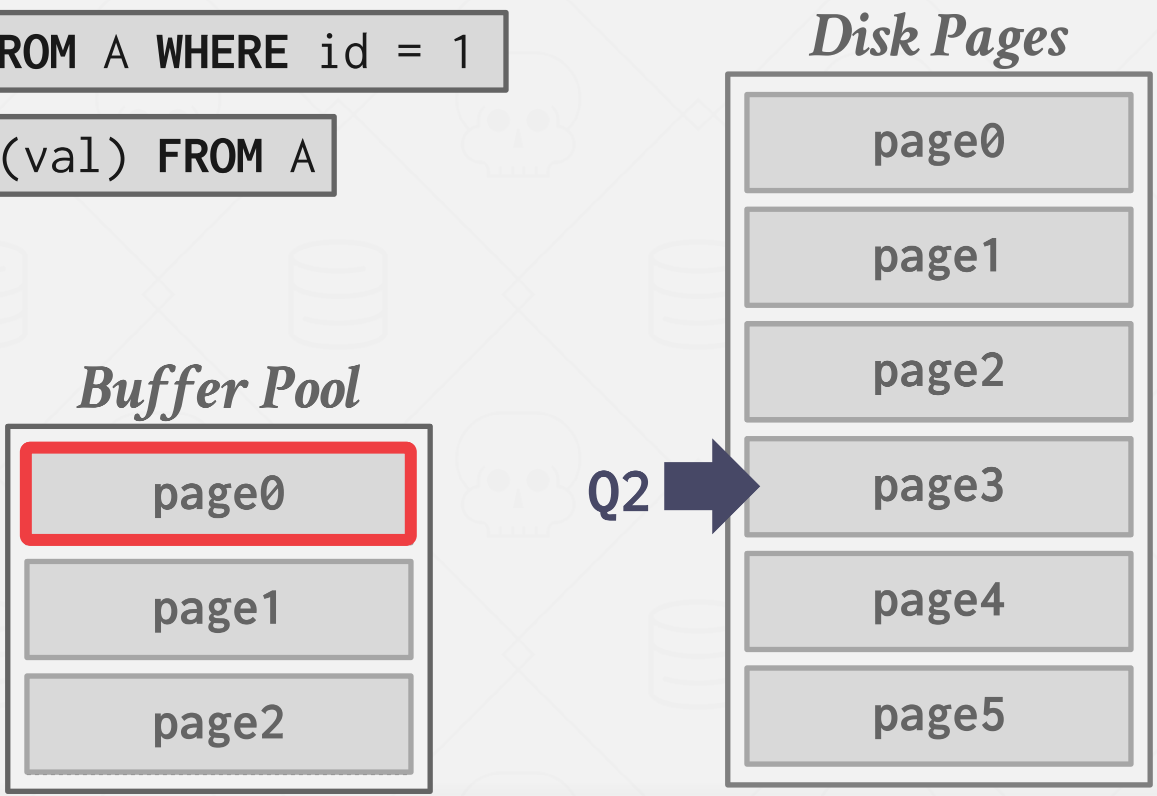




SEQUENTIAL FLOODING

Q1 `SELECT * FROM A WHERE id = 1`

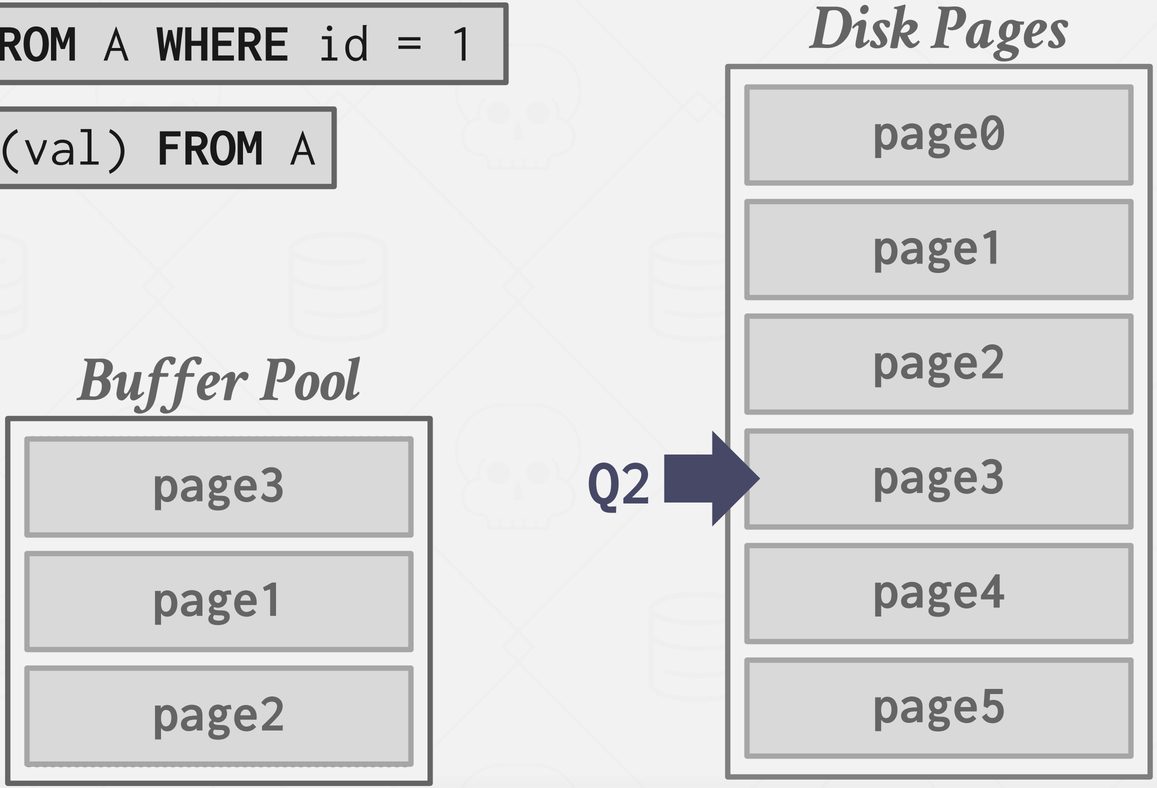
Q2 `SELECT AVG(val) FROM A`



SEQUENTIAL FLOODING

Q1 `SELECT * FROM A WHERE id = 1`

Q2 `SELECT AVG(val) FROM A`

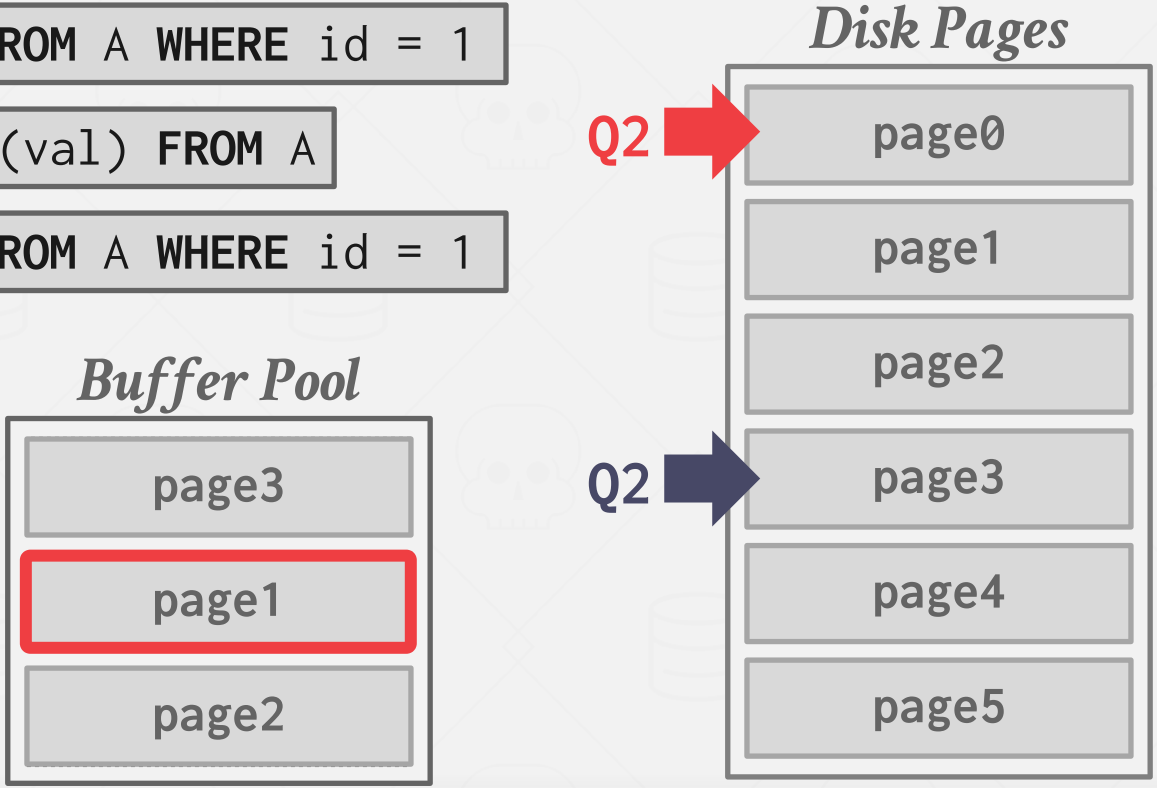


SEQUENTIAL FLOODING

Q1 `SELECT * FROM A WHERE id = 1`

Q2 `SELECT AVG(val) FROM A`

Q3 `SELECT * FROM A WHERE id = 1`



BETTER POLICIES: LRU-K

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

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

BETTER POLICIES: LOCALIZATION

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

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

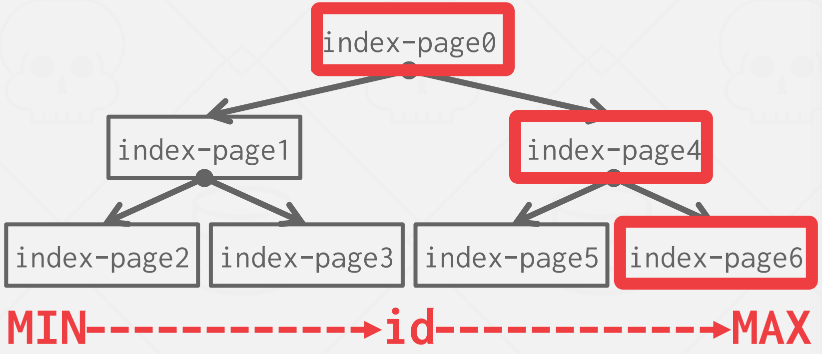
Example: Postgres maintains a small ring buffer that is private to the query.

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++)`



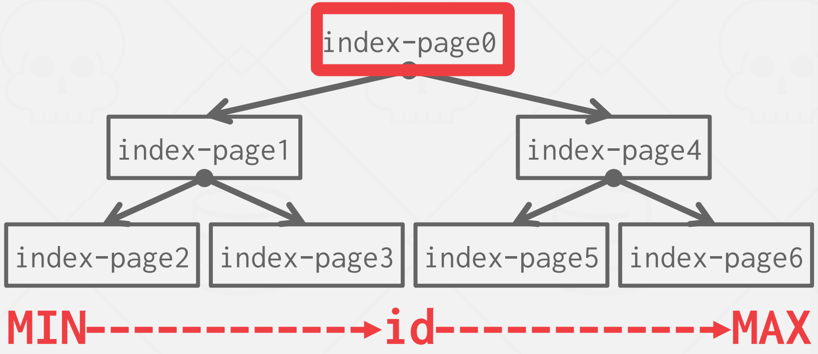
BETTER POLICIES: PRIORITY HINTS

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++)`

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…

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. 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.
→ Extendible Hash Table
→ LRU Replacement Policy
→ Buffer Pool Manager Instance

We will provide you with the disk manager and page layouts.

BusTub

Due Date:
Sunday Oct 2nd @ 11:59pm

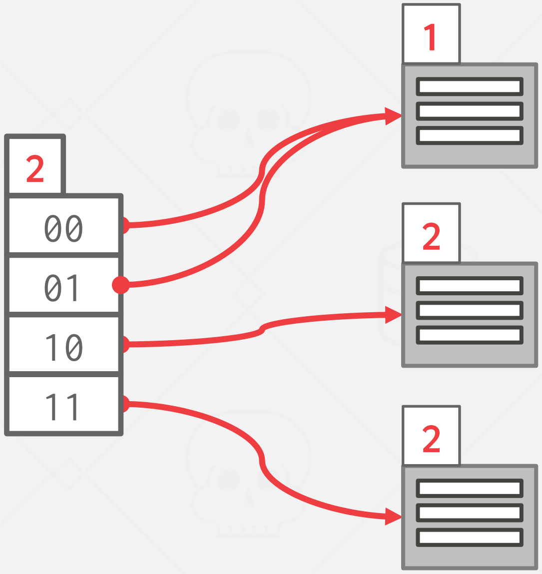
TASK #1 – EXTENDIBLE HASH TABLE

Build a thread-safe extendible hash table to manage the DBMS's buffer pool page table.

→ Use unordered buckets to store key/value pairs.

→ You must support growing table size.

→ You do not need to support shrinking.



General Hints:

→ You can use **std::hash** and **std::mutex**.

TASK #2 – LRU-K REPLACEMENT POLICY

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

General Hints:

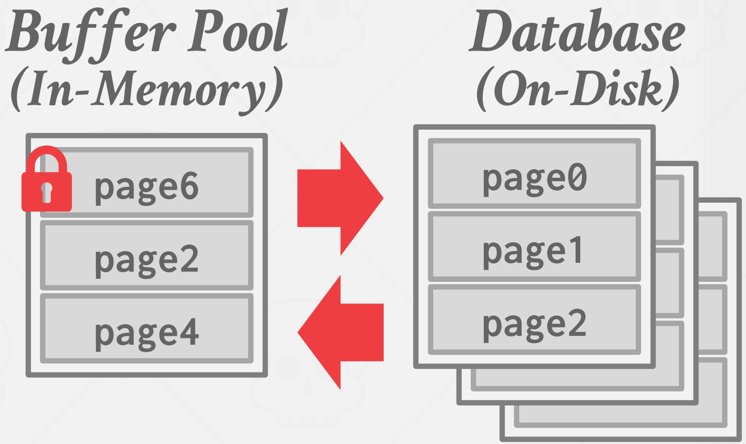
→ Your `LRUKReplacer` needs to check the "pinned" status of a `Page`.

→ If there are no pages touched since last sweep, then return the lowest page id.

TASK #2 – 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.
- → We will provide you components to read/write data from disk.
- → Use whatever data structure you want for the page table.

General Hints:
- → 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-lint`
→ `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 shirt!

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.