Carnegie Mellon University

Buffer Pools

Lecture #05

Database Systems
15-445/15-645
Fall 2018

AP
Andy Pavlo
Computer Science
Carnegie Mellon Univ.

UPCOMING DATABASE EVENTS

Relational AI Talk

→ Wednesday Sep 12th @ 4:00pm
→ GHC 8102

MapD Talk

→ Thursday Sept 20th @ 12pm
→ CIC 4th Floor

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

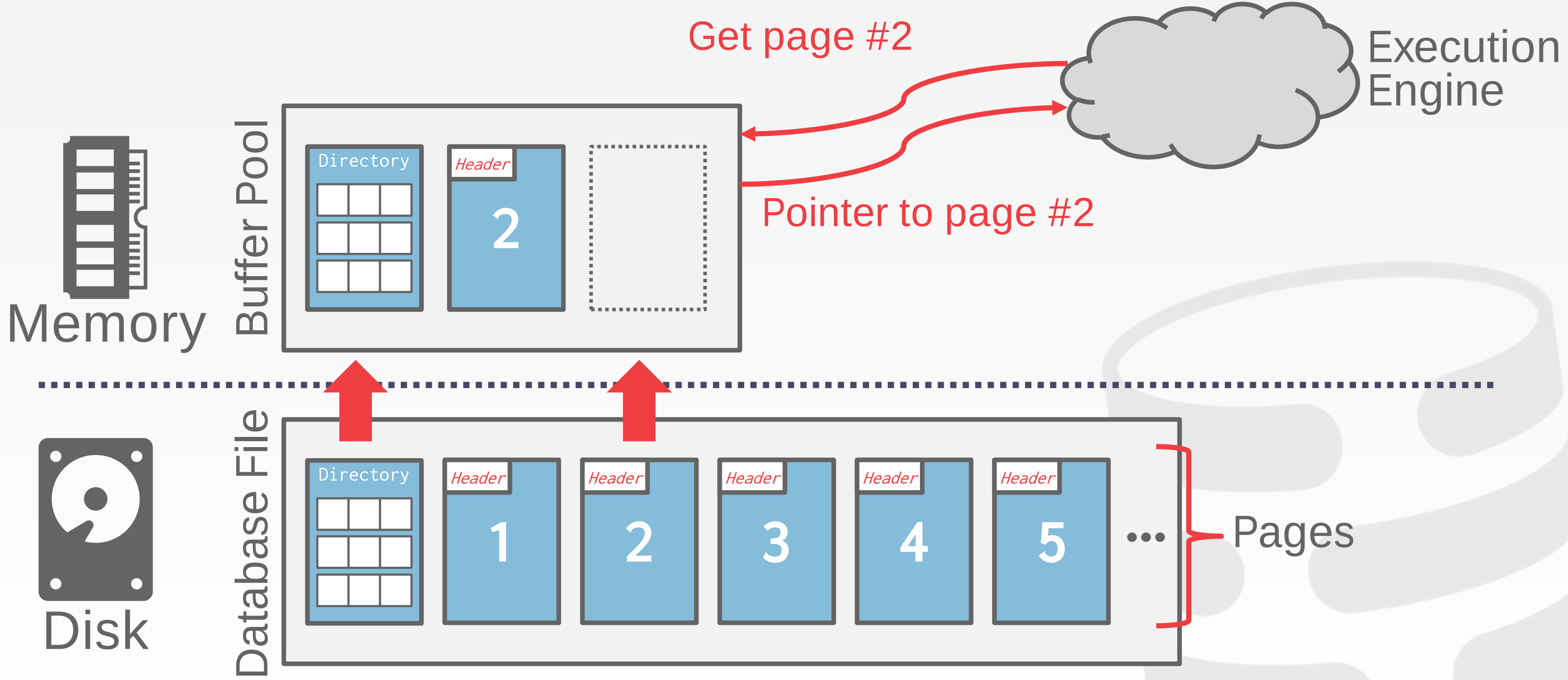
Spatial Control:

→ 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 minimize the number of stalls from having to read data from disk.

DISK-ORIENTED DBMS



TODAY'S AGENDA

Buffer Pool Manager

Replacement Policies

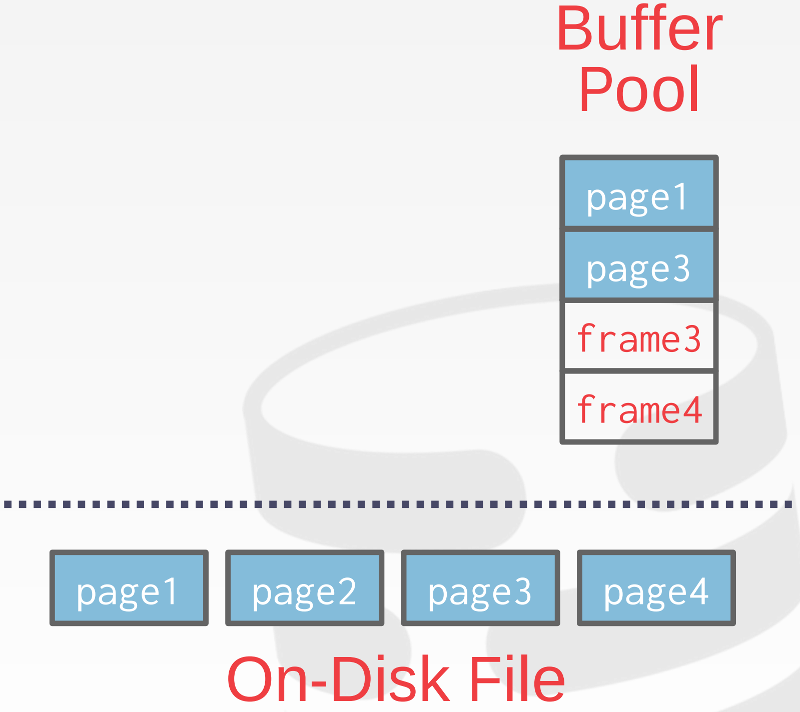
Allocation Policies

Other Memory Pools

BUFFER POOL ORGANIZATION

Memory region organized as an array of fixed-size pages.
An array entry is called a frame.

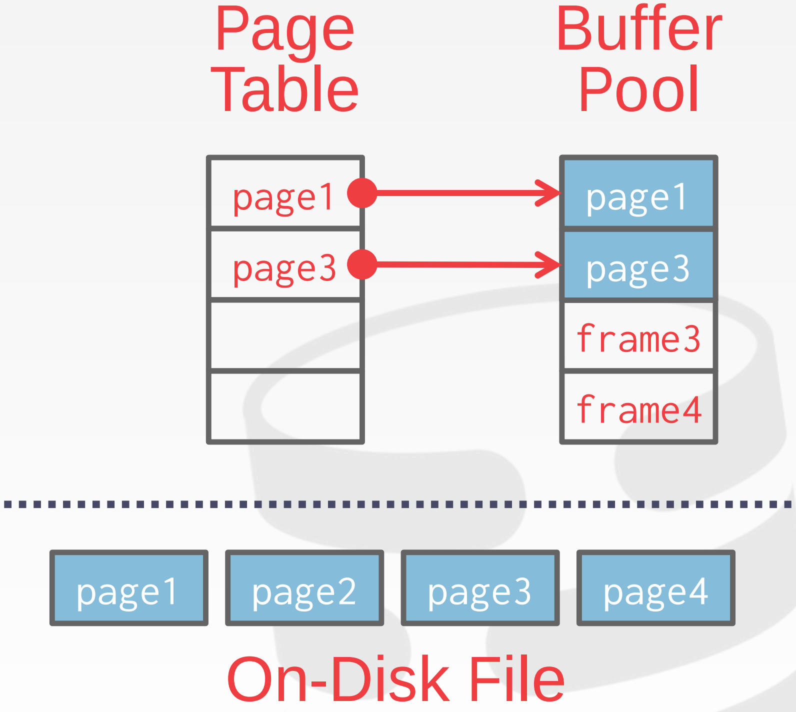
When the DBMS requests a page, an exact copy is placed into one of these frames.



BUFFER POOL META-DATA

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

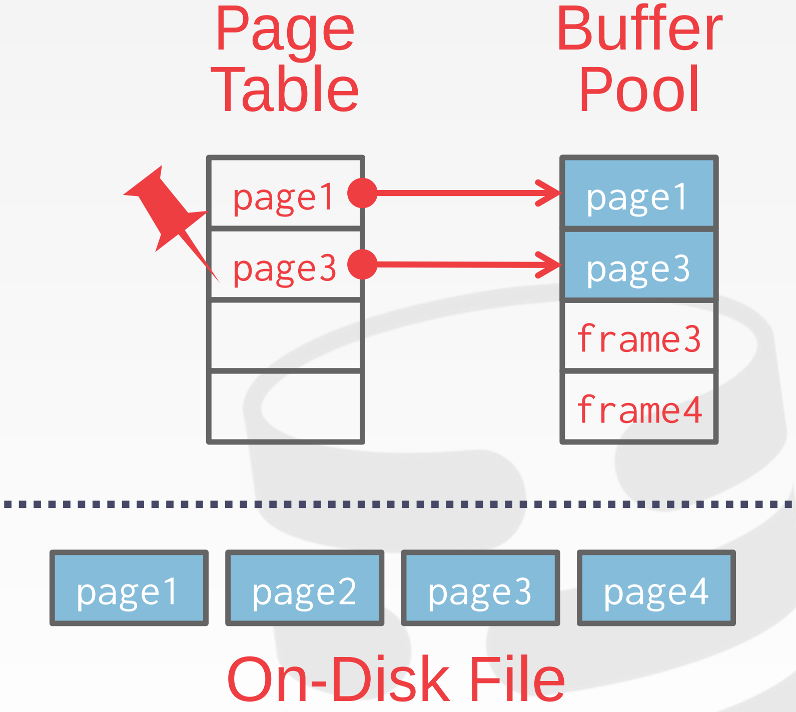
Also maintains additional meta-data per page:
→ **Dirty Flag**
→ **Pin/Reference Counter**



BUFFER POOL META-DATA

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

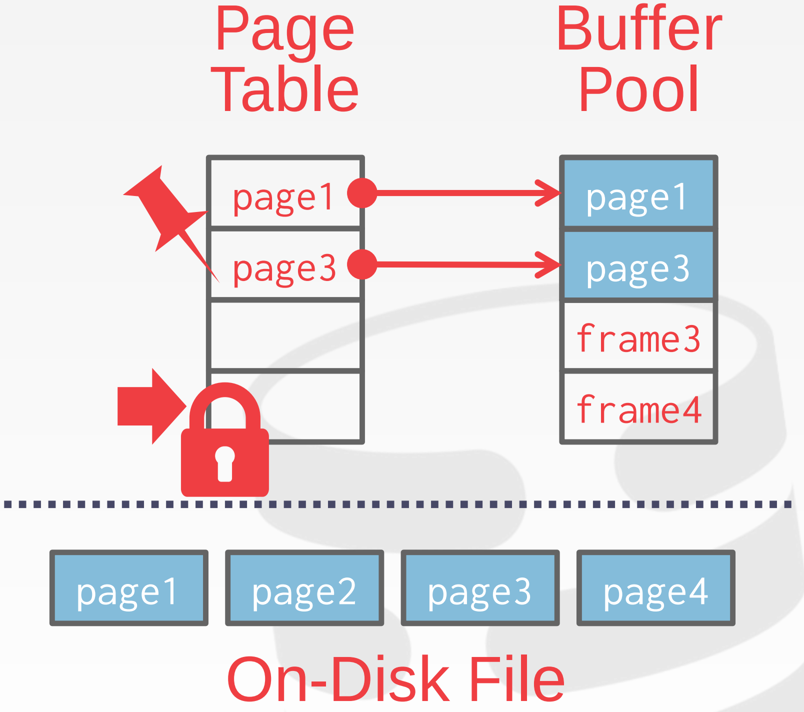
Also maintains additional meta-data per page:
→ **Dirty Flag**
→ **Pin/Reference Counter**



BUFFER POOL META-DATA

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

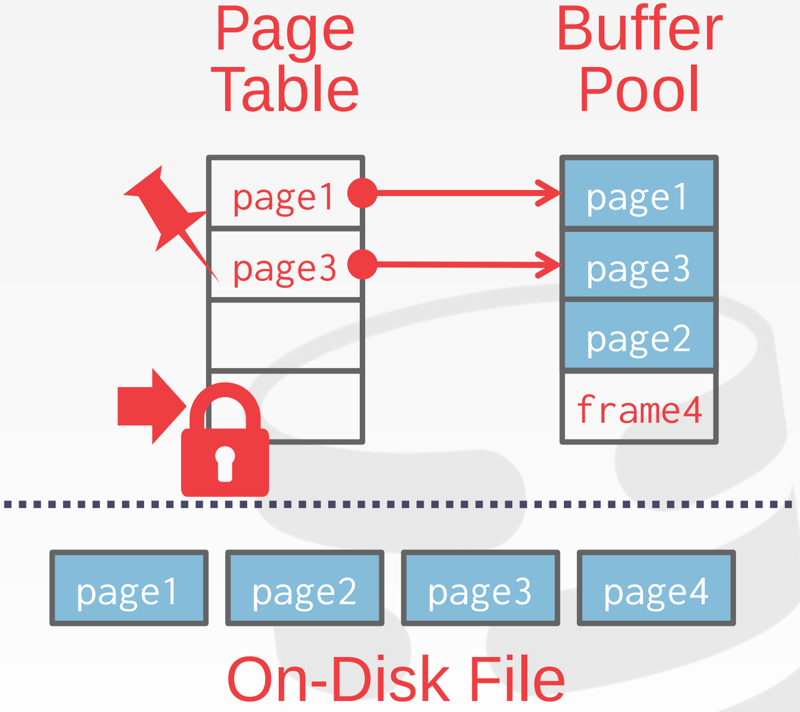
Also maintains additional meta-data per page:
→ **Dirty Flag**
→ **Pin/Reference Counter**



BUFFER POOL META-DATA

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

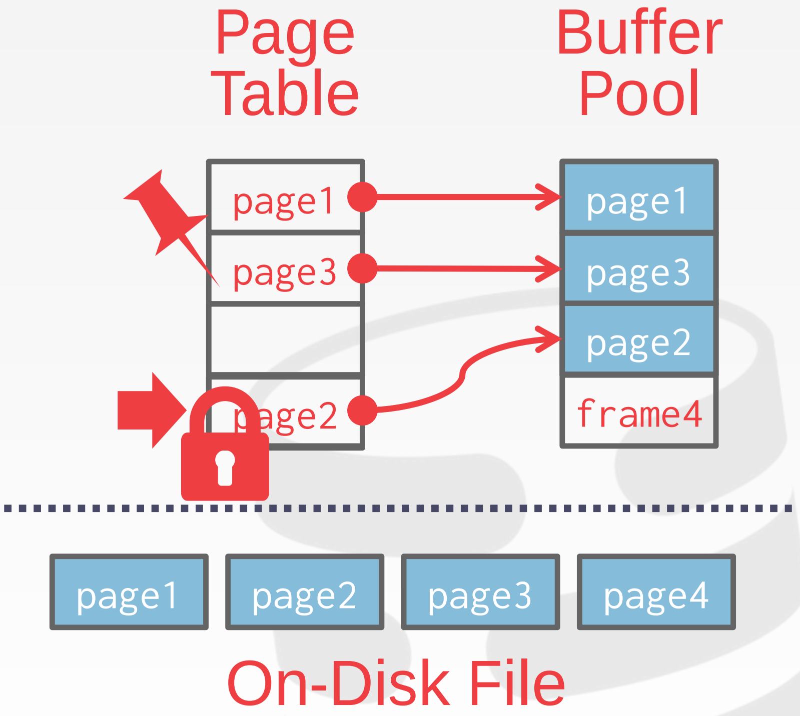
Also maintains additional meta-data per page:
→ **Dirty Flag**
→ **Pin/Reference Counter**



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**

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.

One buffer pool with multi threads accessing would cause bottleneck

MULTIPLE BUFFER POOLS

buffer pool pre warm: put everything in the DBMS into buffer pool to its limit

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

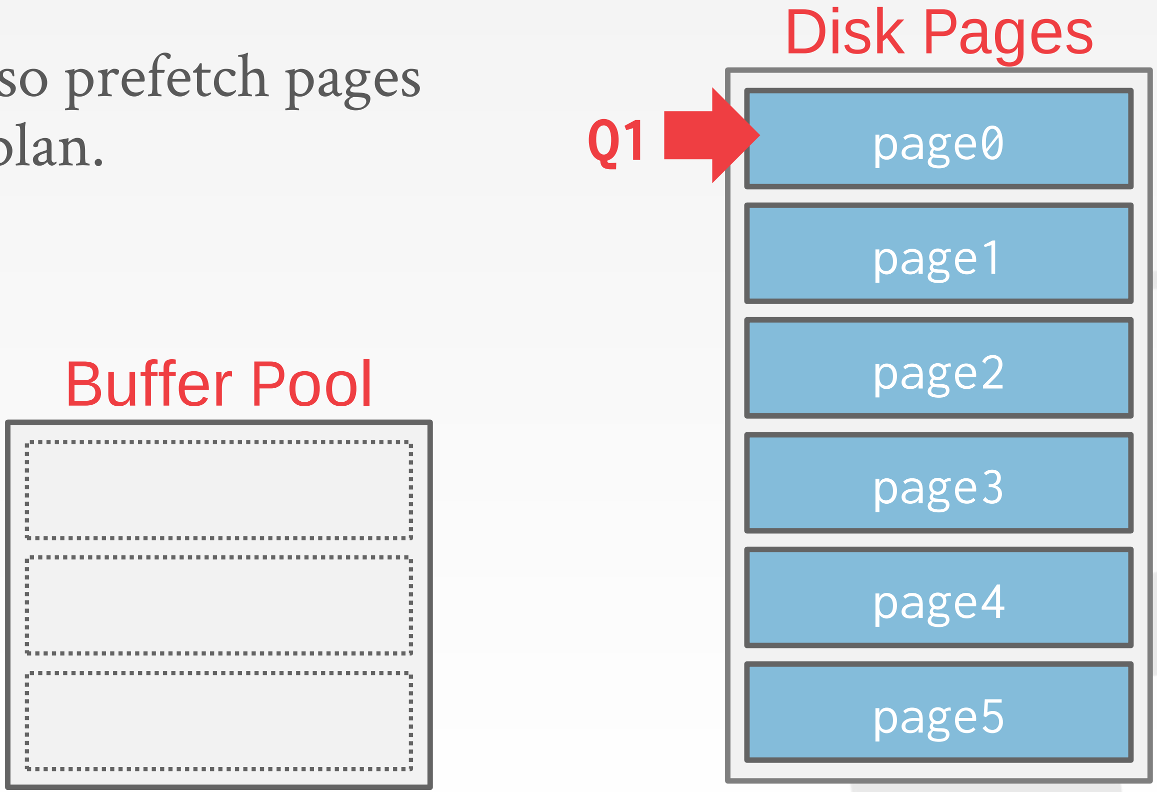
partition memory to several buffer pools and same data use same latch (simply by hash or mod to map page id to buffer pool)

Helps reduce latch contention and improve locality.

Again, OS does not know anything in a file but bytes, but DBMS can tell if this page contains indexes or data and do specific things on that

PRE-FETCHING

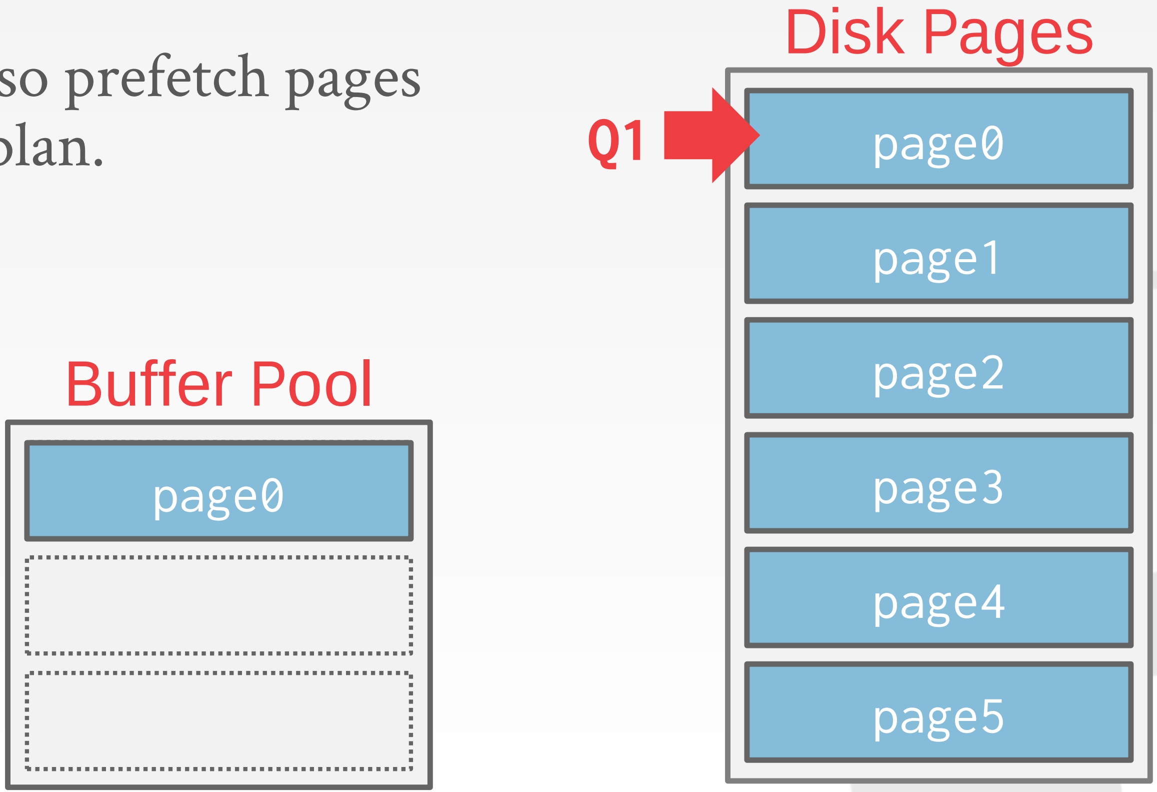
The DBMS can also prefetch pages based on a query plan.
→ Sequential Scans
→ Index Scans



PRE-FETCHING

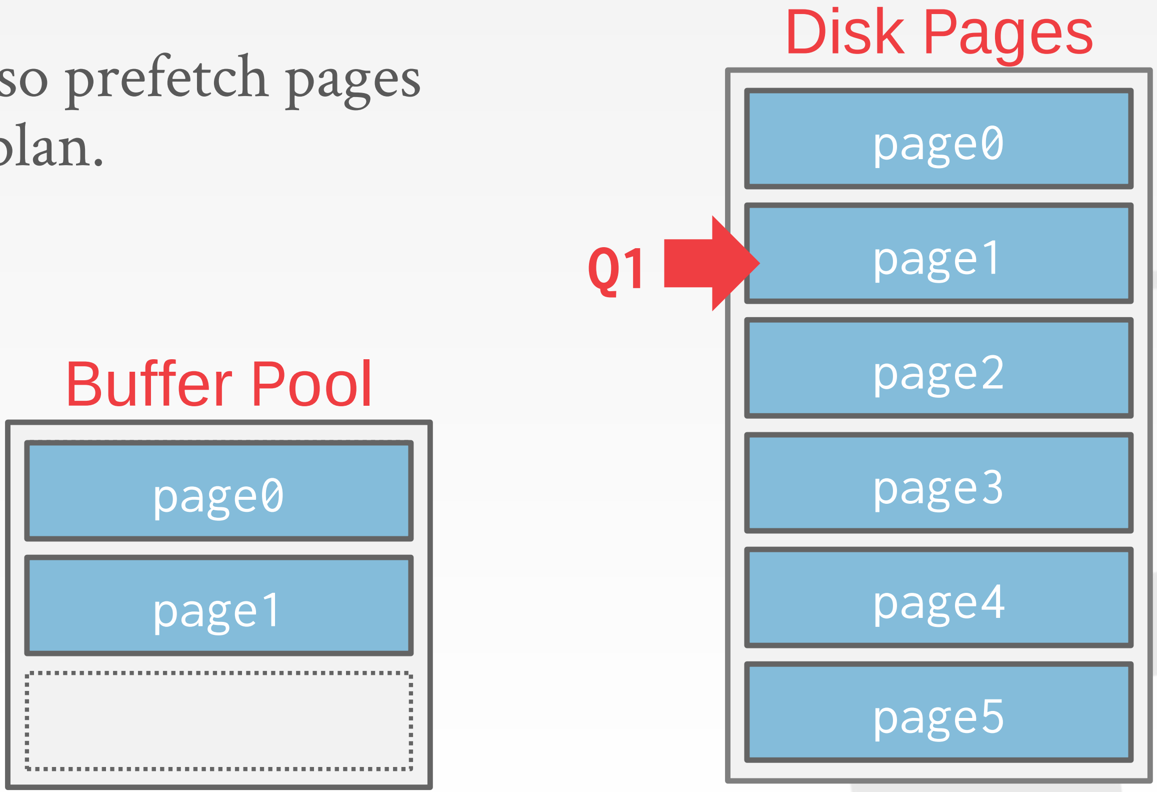
The DBMS can also prefetch pages based on a query plan.

→ Sequential Scans
→ Index Scans



PRE-FETCHING

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

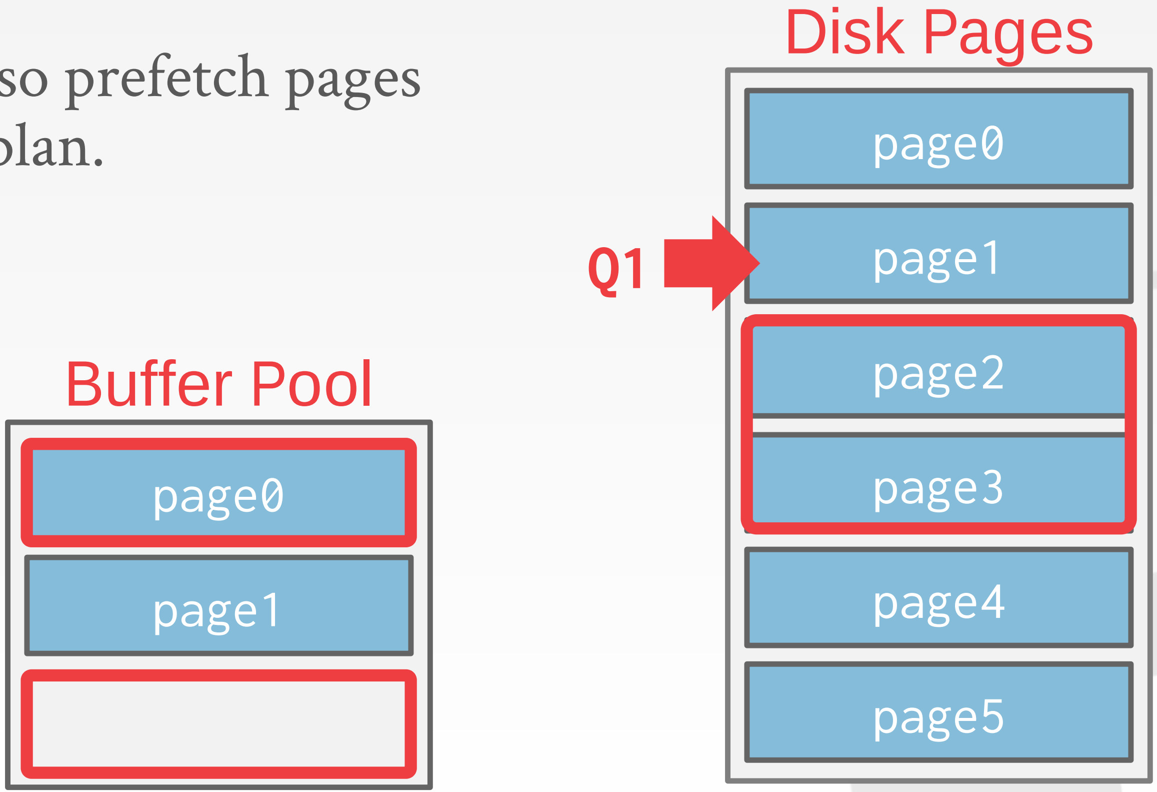


PRE-FETCHING

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

→ Sequential Scans

→ Index Scans

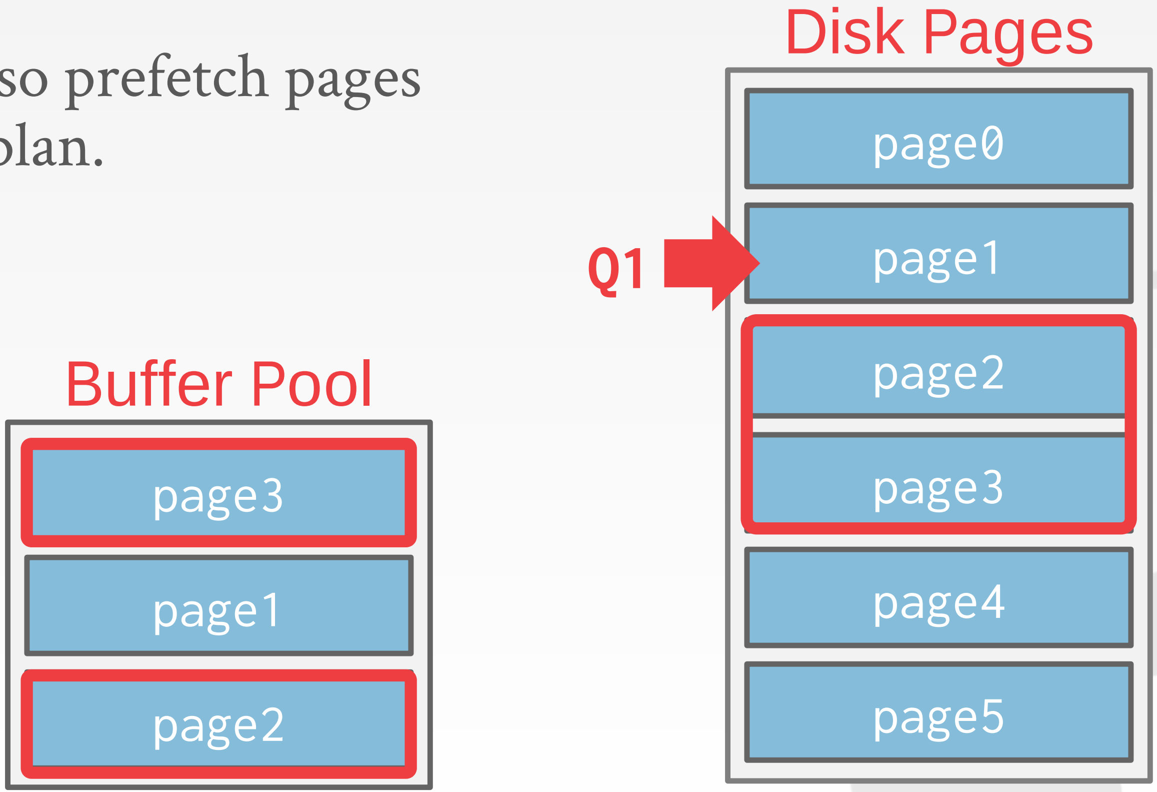


PRE-FETCHING

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

→ Sequential Scans

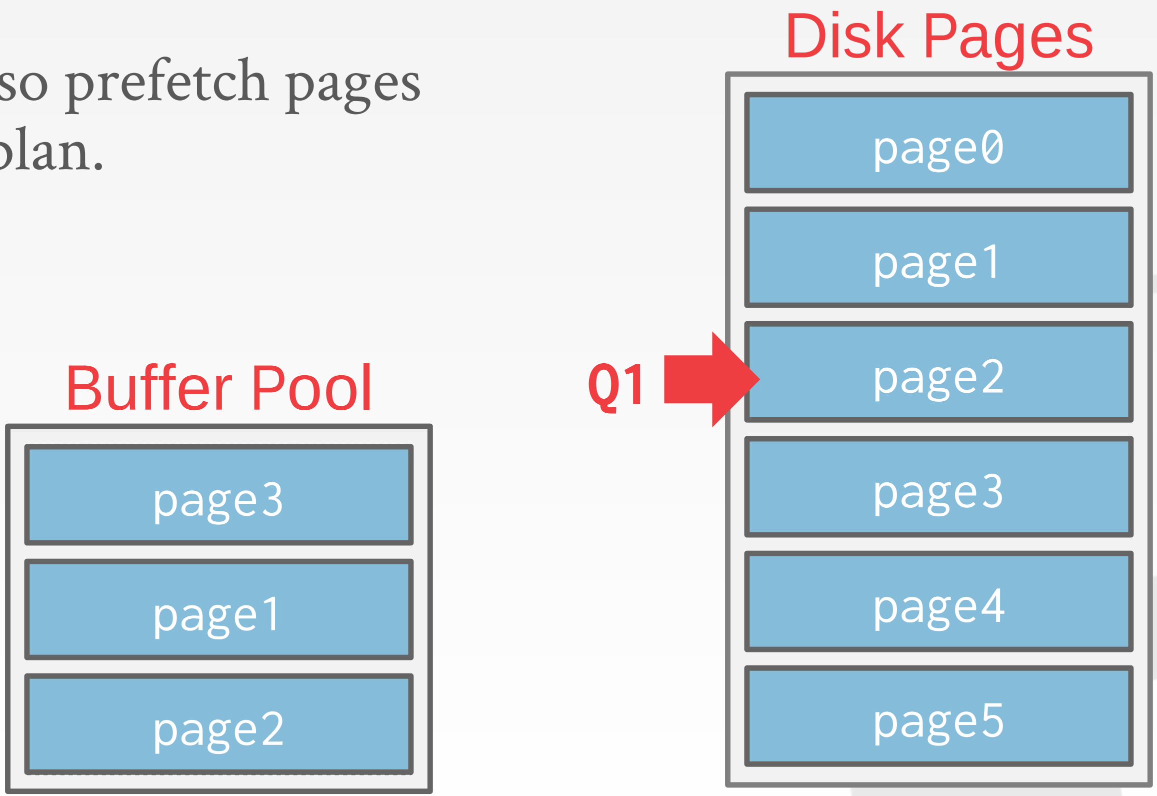
→ Index Scans



PRE-FETCHING

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

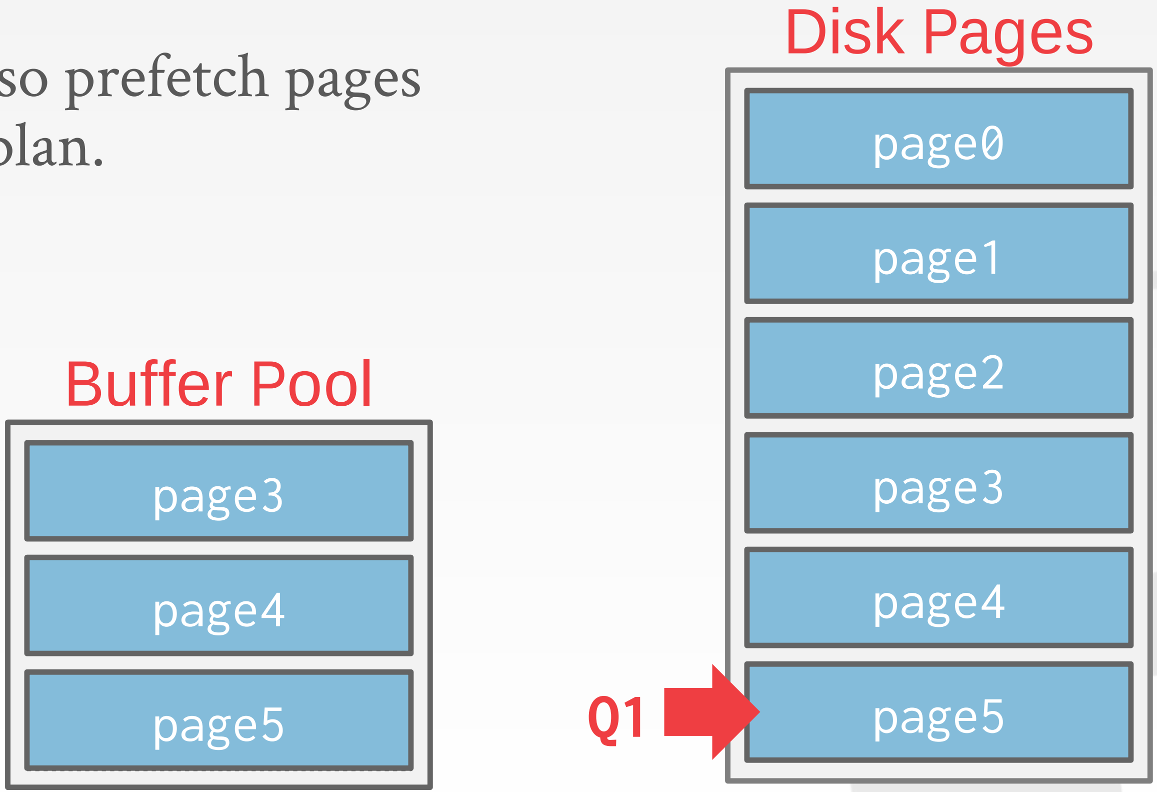
→ Sequential Scans
→ Index Scans



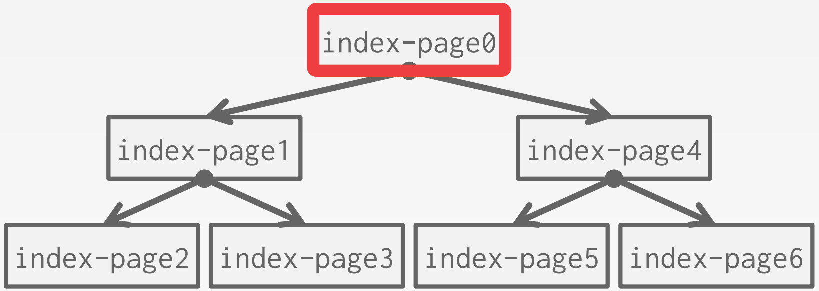
PRE-FETCHING

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

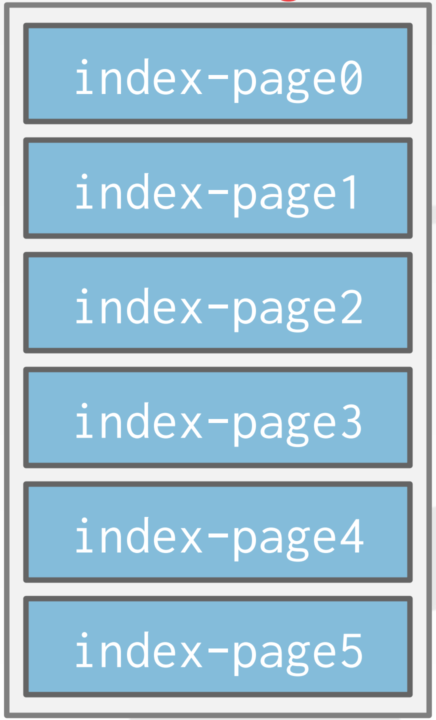
→ Sequential Scans
→ Index Scans



PRE-FETCHING

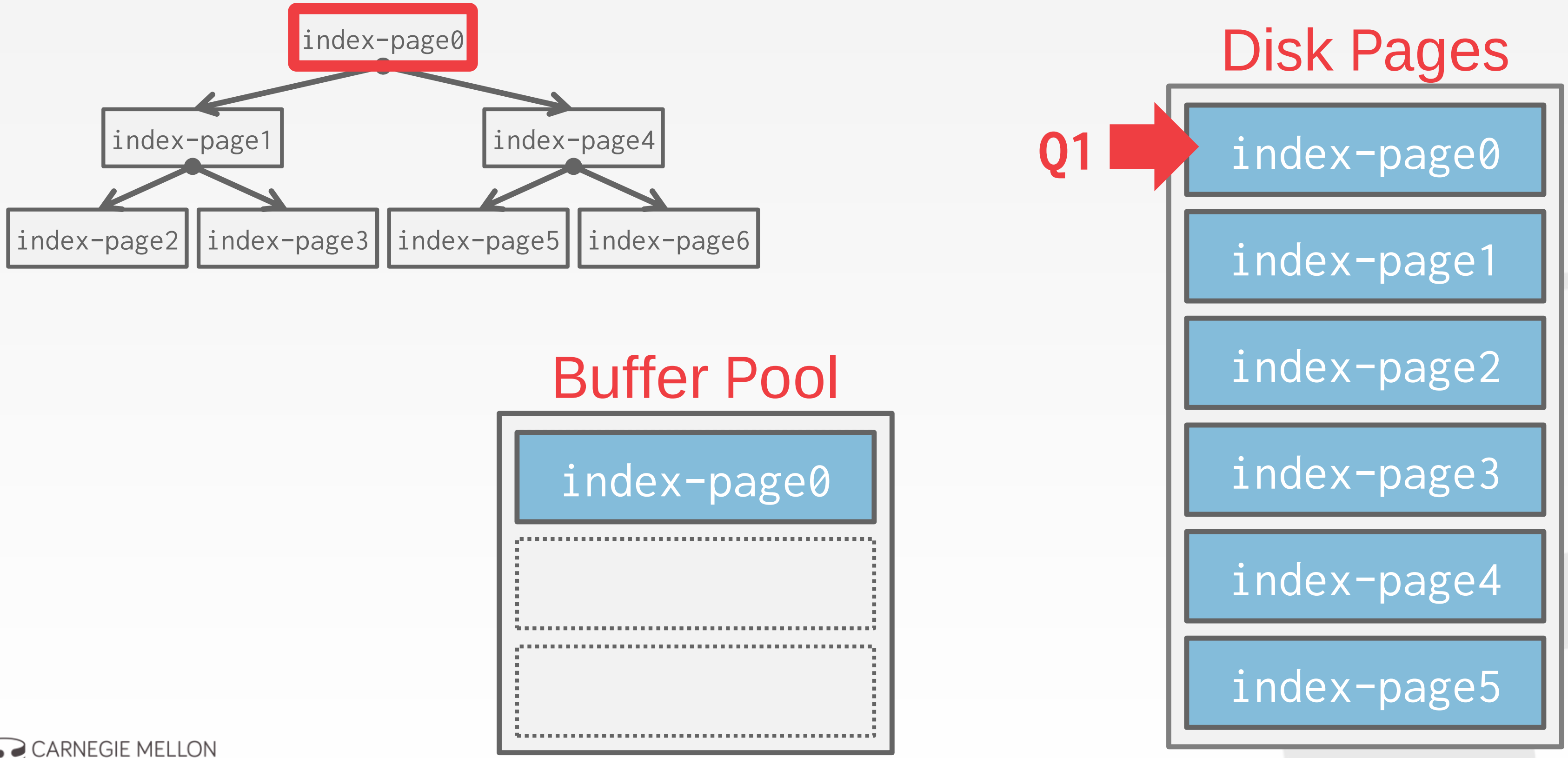


Disk Pages

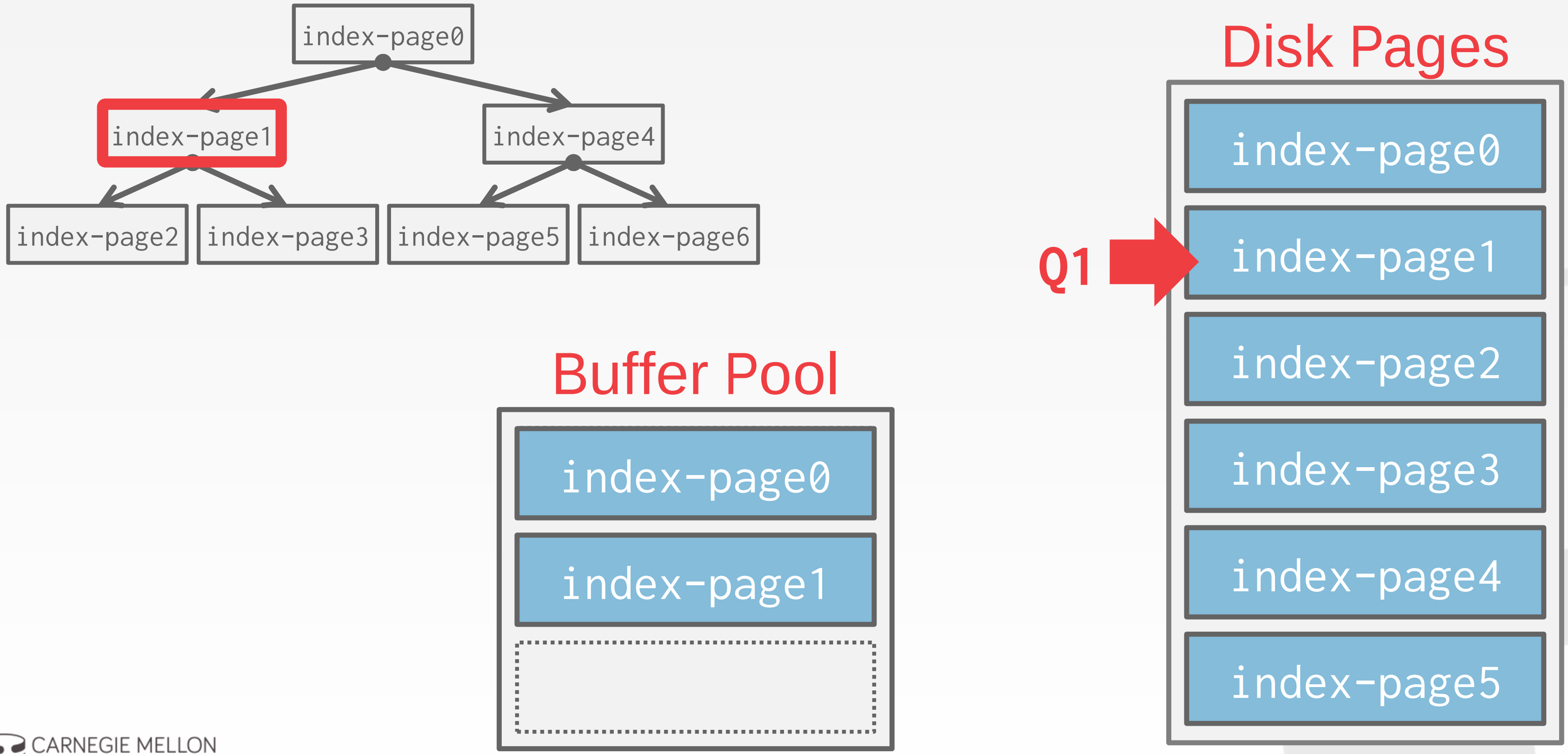


Buffer Pool

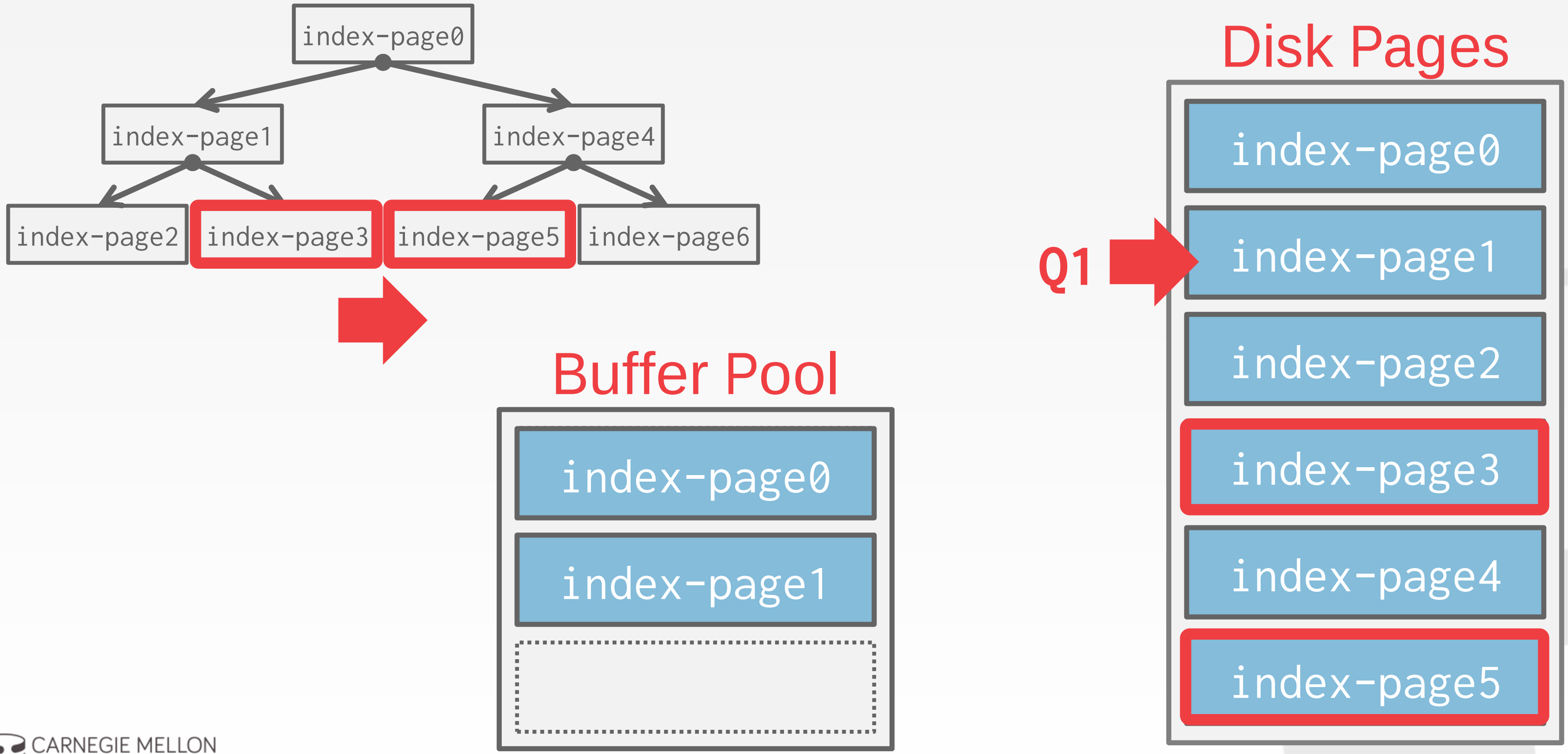
PRE-FETCHING



PRE-FETCHING



PRE-FETCHING



SCAN SHARING

Queries are able to reuse data retrieved from storage or operator computations.
→ 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 exactly the same.
→ Can also share intermediate results.

SCAN SHARING

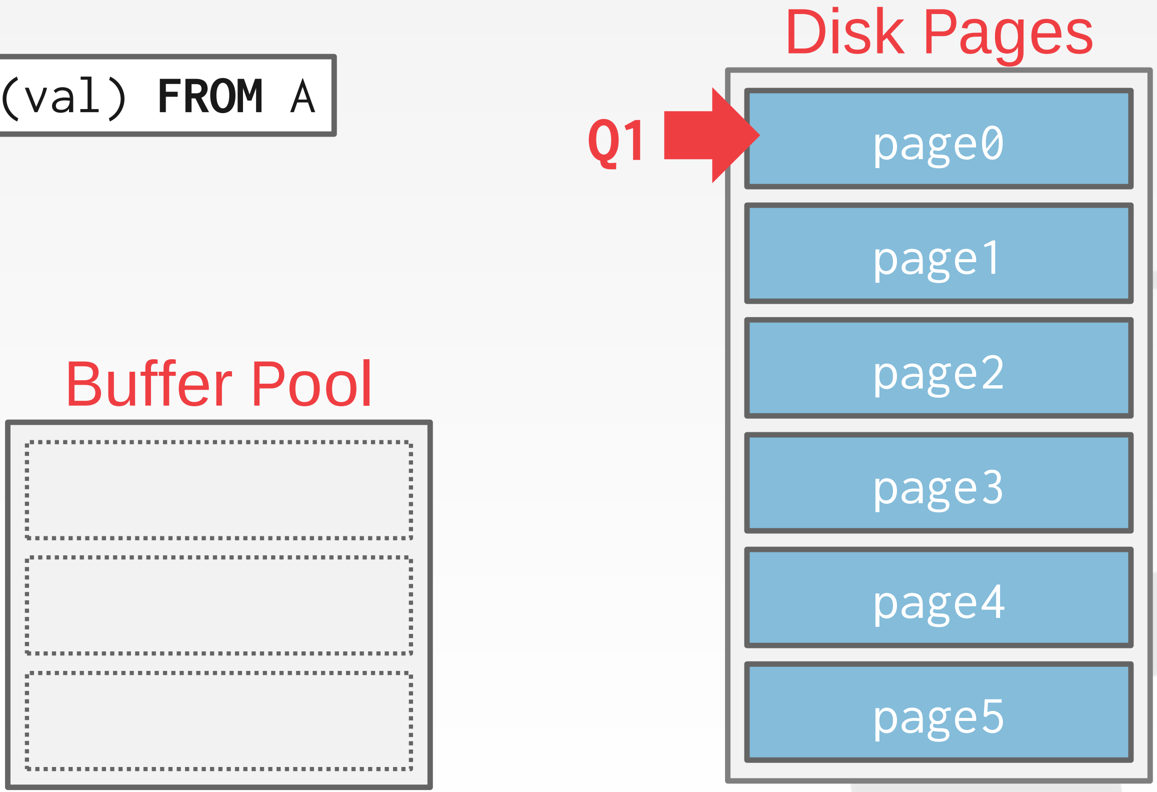
If a query starts a scan and if there one already doing this, then the DBMS will attach to the second query's cursor.

→ The DBMS keeps track of where the second query joined with the first so that it can finish the scan when it reaches the end of the data structure.

Fully supported in IBM DB2 and MSSQL.
Oracle only supports cursor sharing for identical queries.

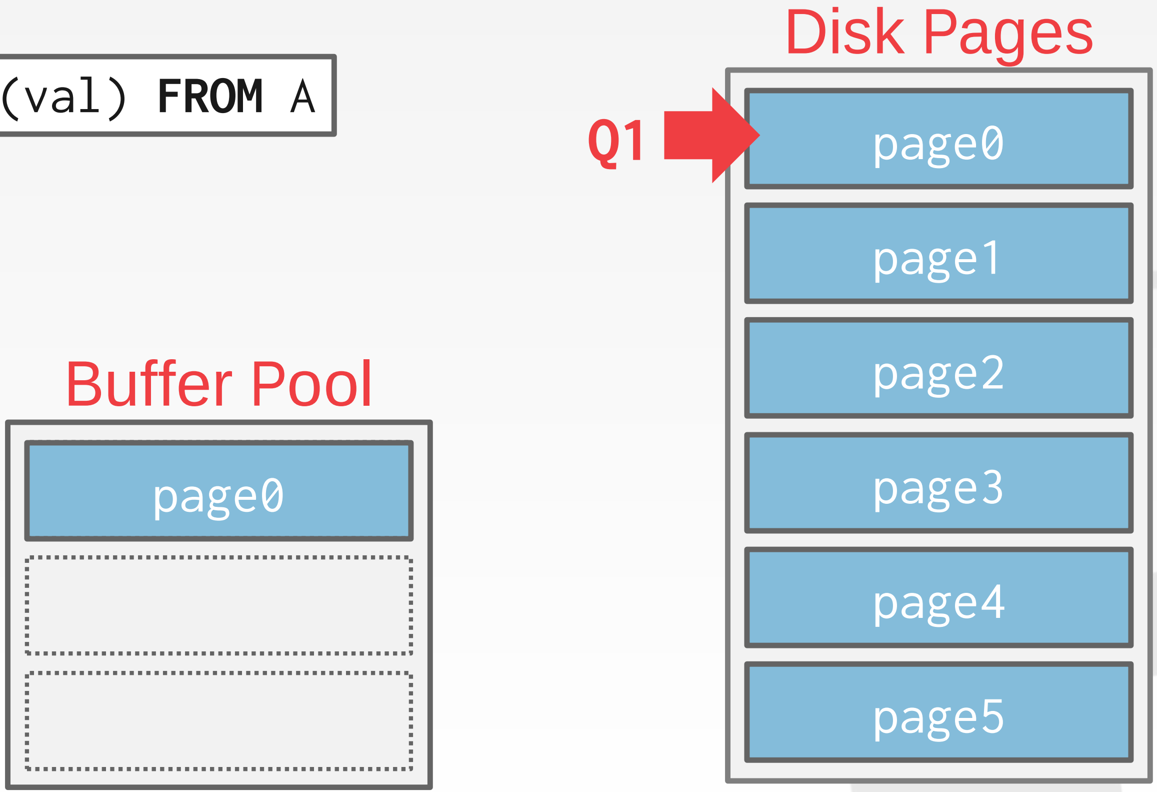
SCAN SHARING

Q1 `SELECT SUM(val) FROM A`



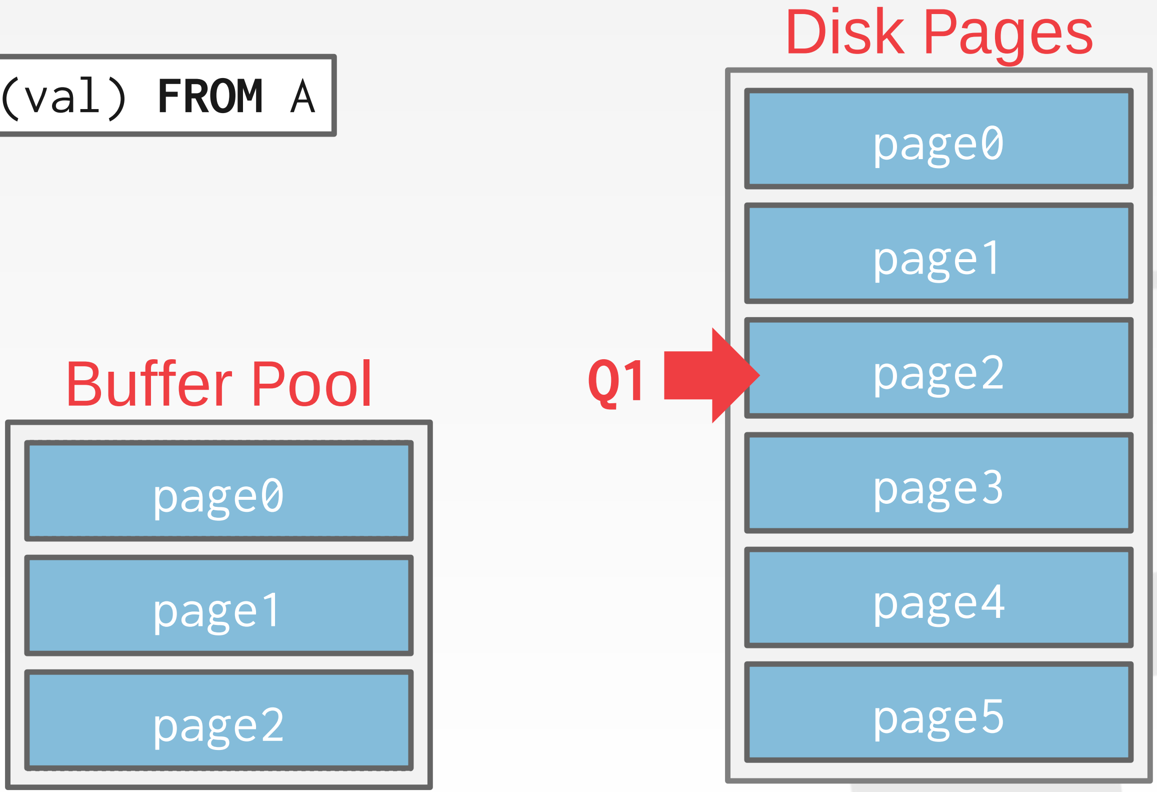
SCAN SHARING

Q1 `SELECT SUM(val) FROM A`



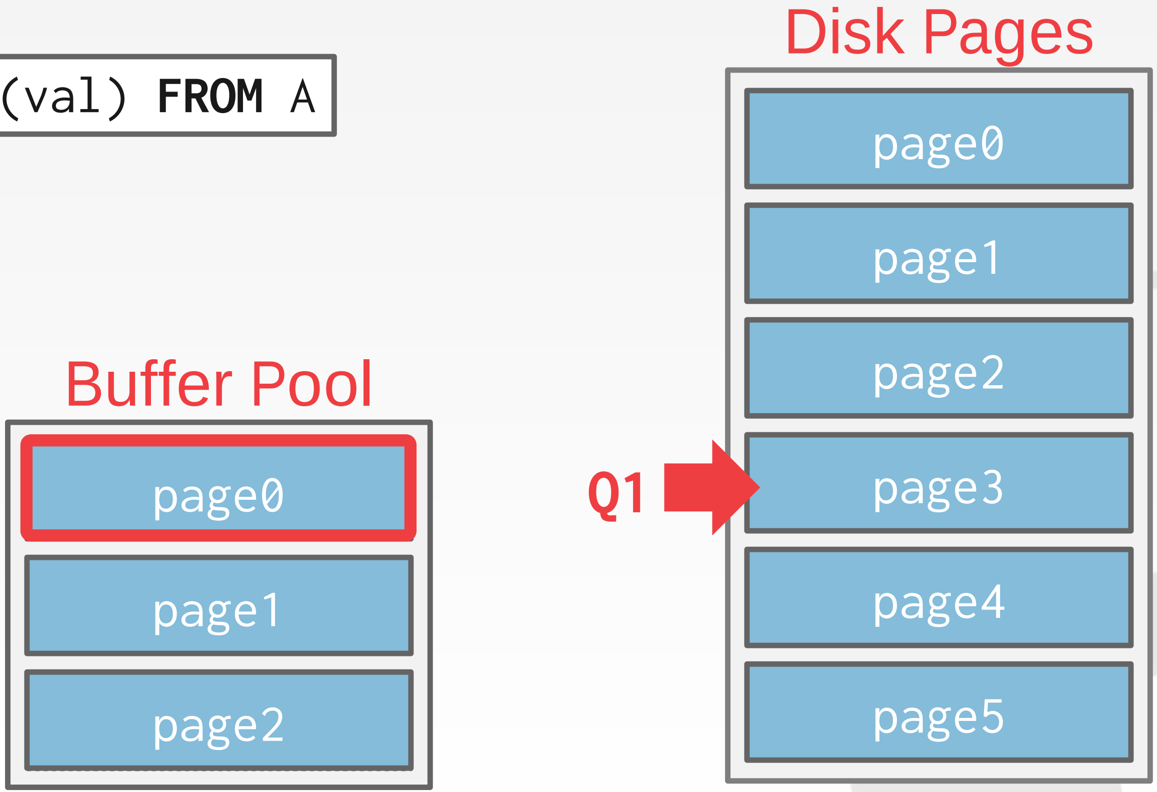
SCAN SHARING

Q1 `SELECT SUM(val) FROM A`



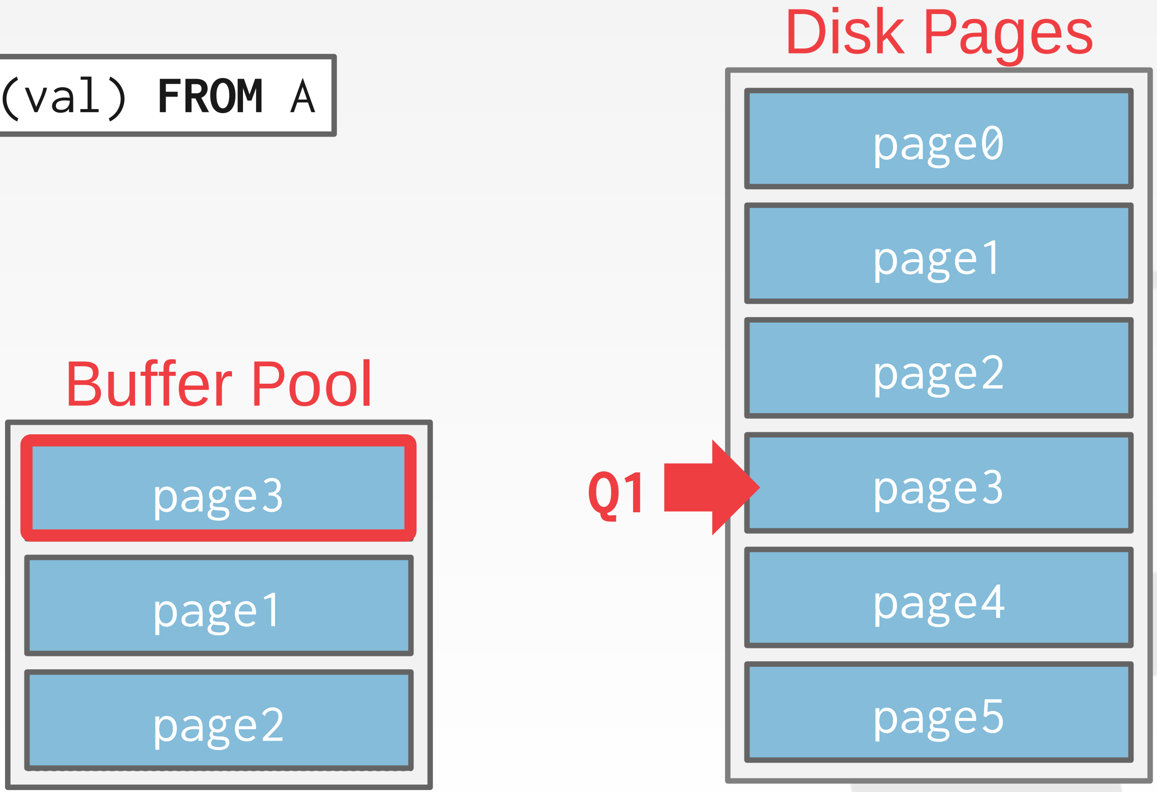
SCAN SHARING

Q1 `SELECT SUM(val) FROM A`



SCAN SHARING

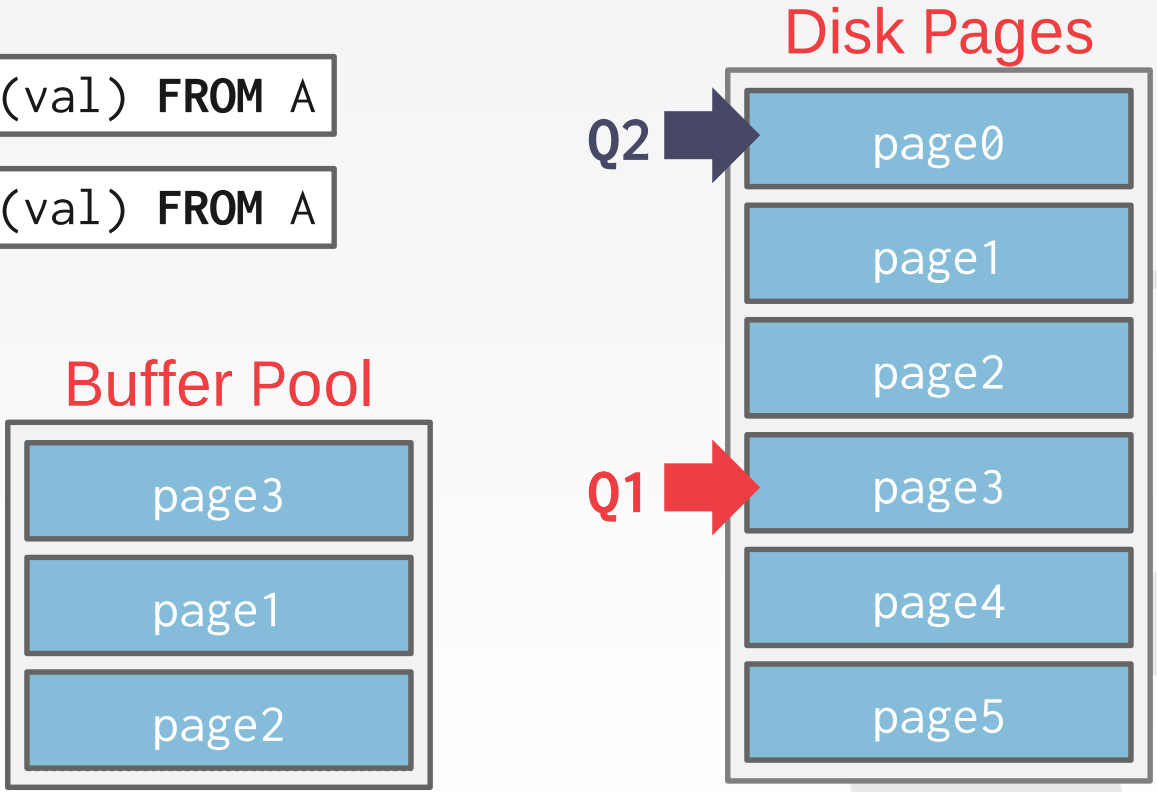
Q1 `SELECT SUM(val) FROM A`



SCAN SHARING

Q1 `SELECT SUM(val) FROM A`

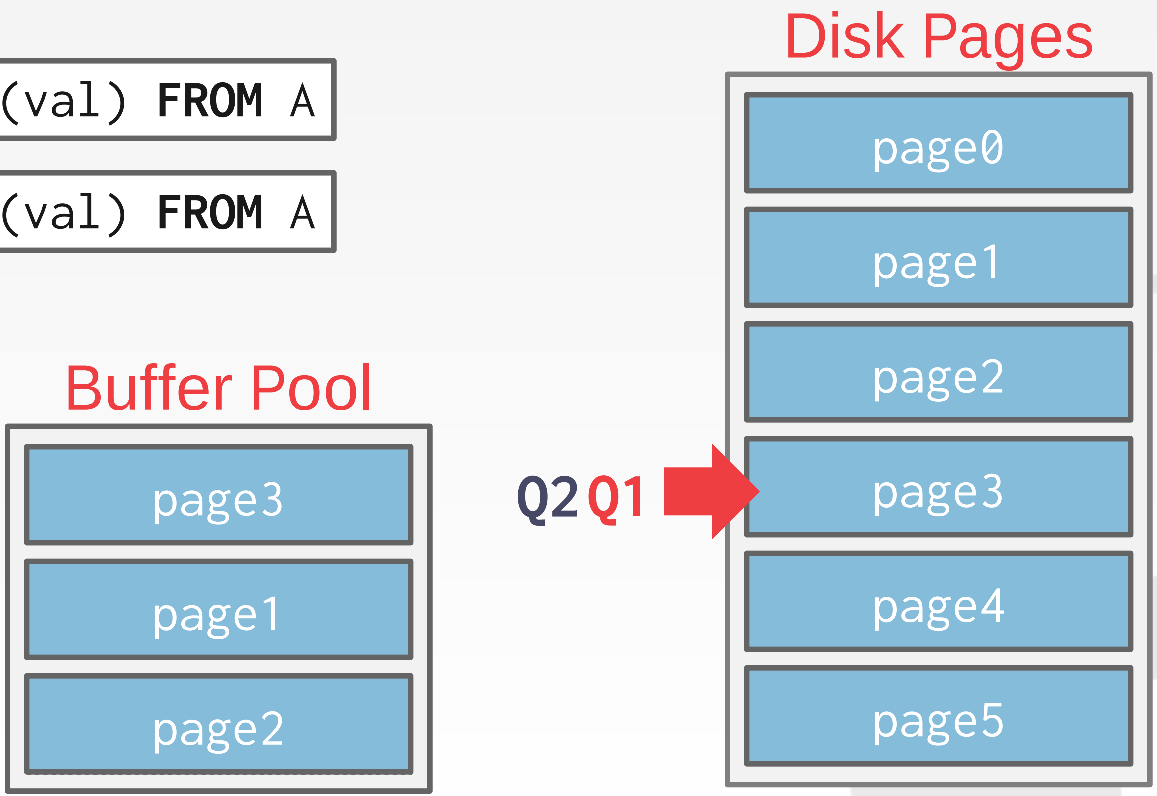
Q2 `SELECT AVG(val) FROM A`



SCAN SHARING

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

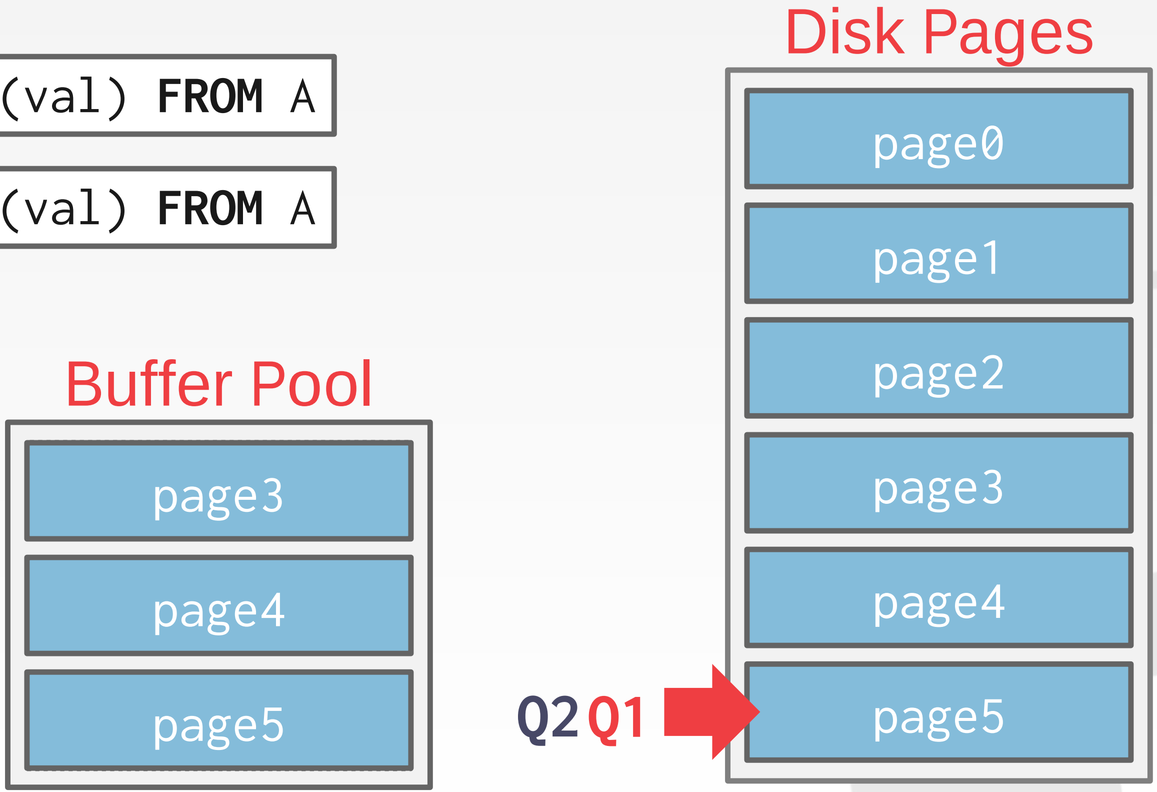
Q2
```
SELECT AVG(val) FROM A
```



SCAN SHARING

Q1 `SELECT SUM(val) FROM A`

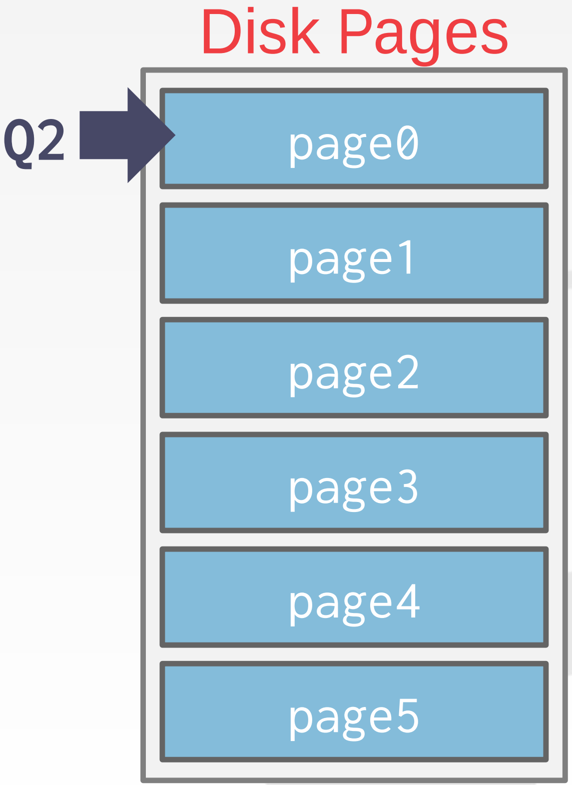
Q2 `SELECT AVG(val) FROM A`

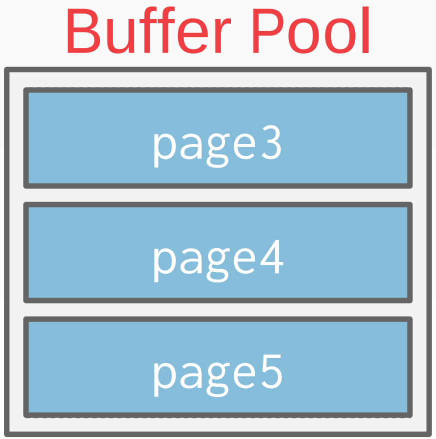


SCAN SHARING

Q1 `SELECT SUM(val) FROM A`

Q2 `SELECT AVG(val) FROM A`

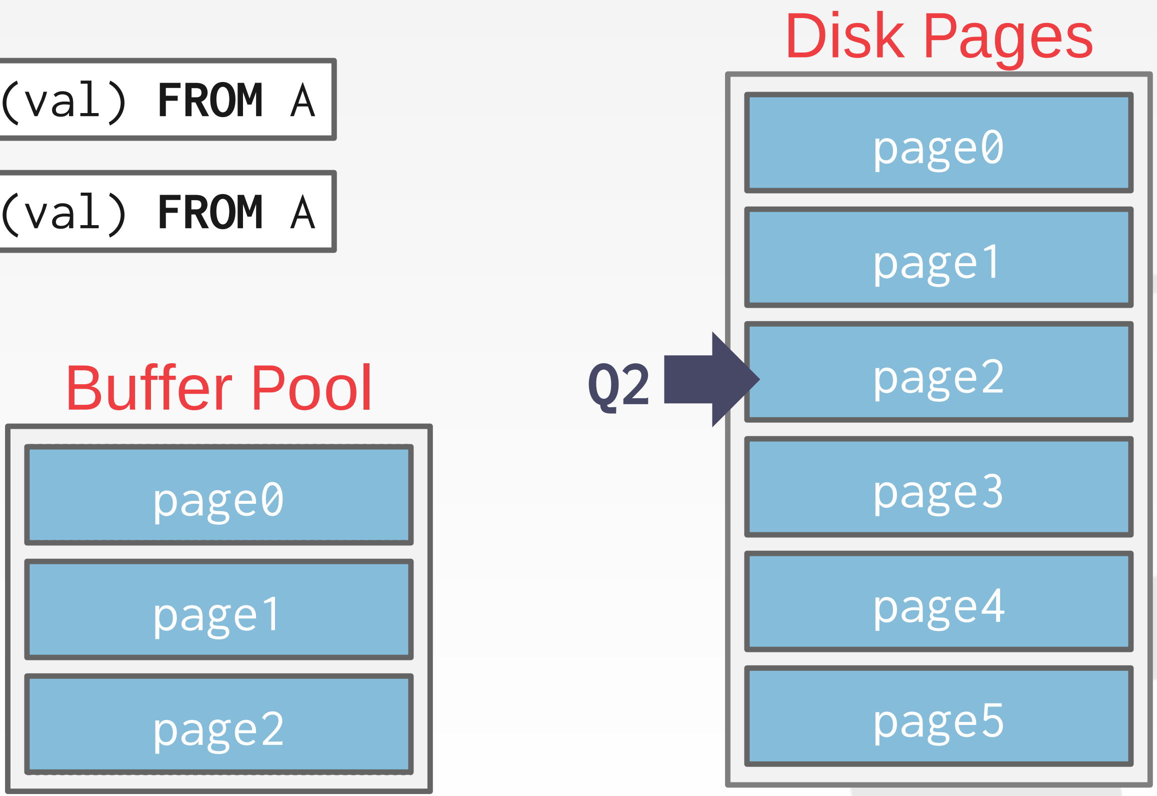




SCAN SHARING

Q1 `SELECT SUM(val) FROM A`

Q2 `SELECT AVG(val) FROM A`



The very last evicted page from buffer pool could be needed page in the future but cannot be maintained in the buffer pool for the current scan

BUFFER POOL BYPASS

So buffer pool bypass is kind of caching the data which could be used for anyone to avoid frequent update on the page table

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.

Called "Light Scans" in Informix.

OS will cache the data your program are using, to avoid caching this data in memory twice, DBMS usually turn os page cache off

OS PAGE CACHE

Most disk operations go through the OS API.

Unless you tell it not to, the OS maintains its own filesystem cache.

Buffer pool is in the residence memory

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

→ Redundant copies of pages.
→ Different eviction policies.

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 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 without needing a separate timestamp per page.

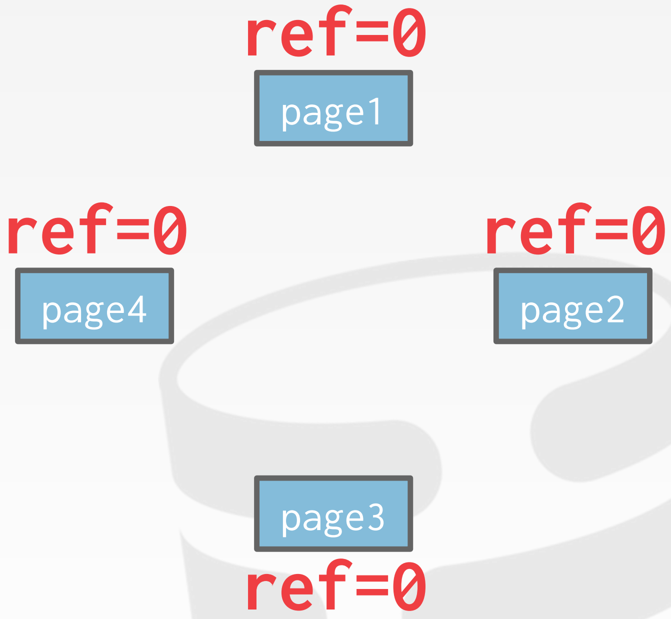
→ Each page has a reference bit.

→ 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.



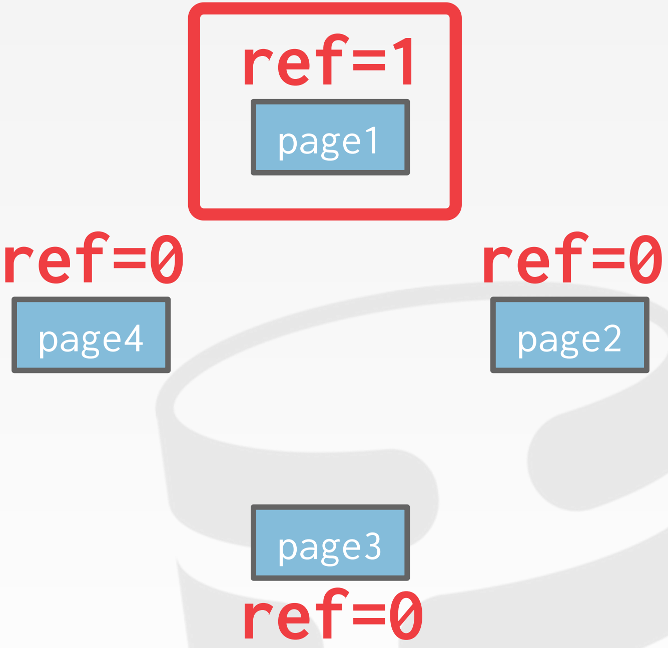
CLOCK

Approximation of LRU without needing 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 without needing a separate timestamp per page.

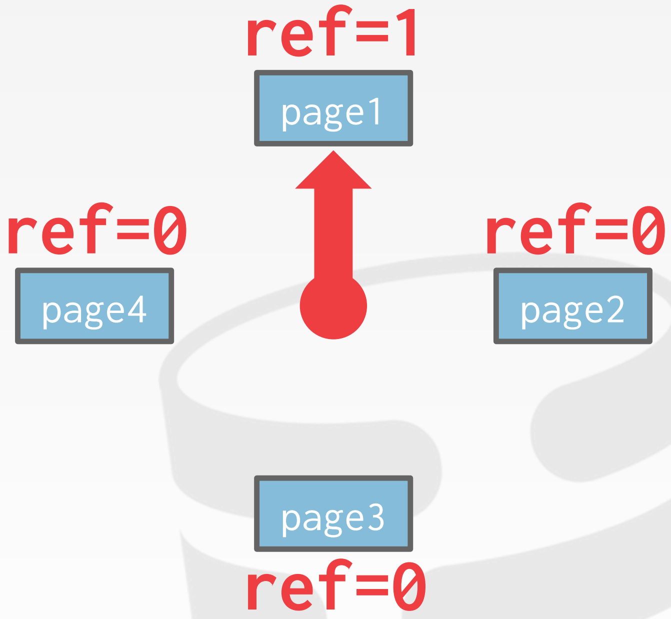
→ Each page has a reference bit.

→ 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 without needing a separate timestamp per page.

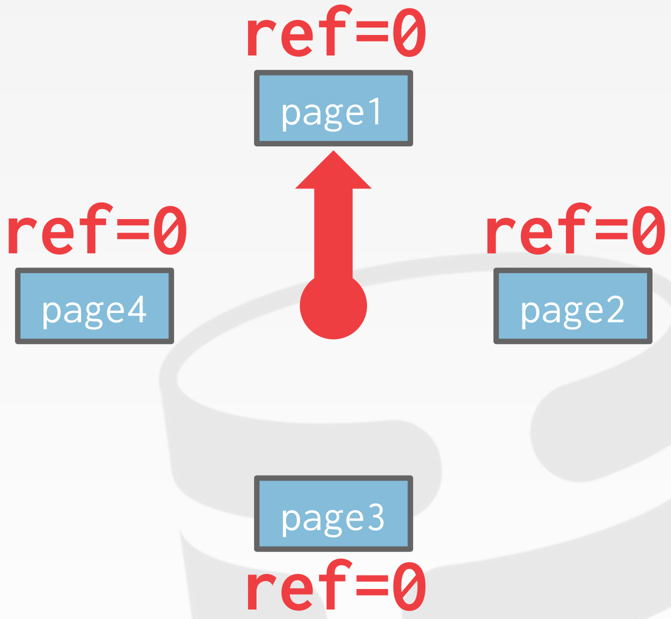
→ Each page has a reference bit.

→ 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 without needing a separate timestamp per page.

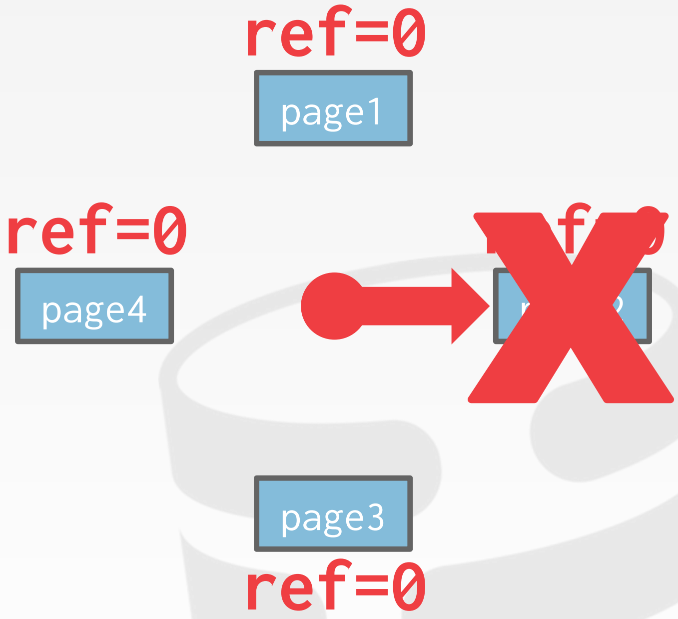
→ Each page has a <u>reference bit</u>.

→ 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 without needing a separate timestamp per page.

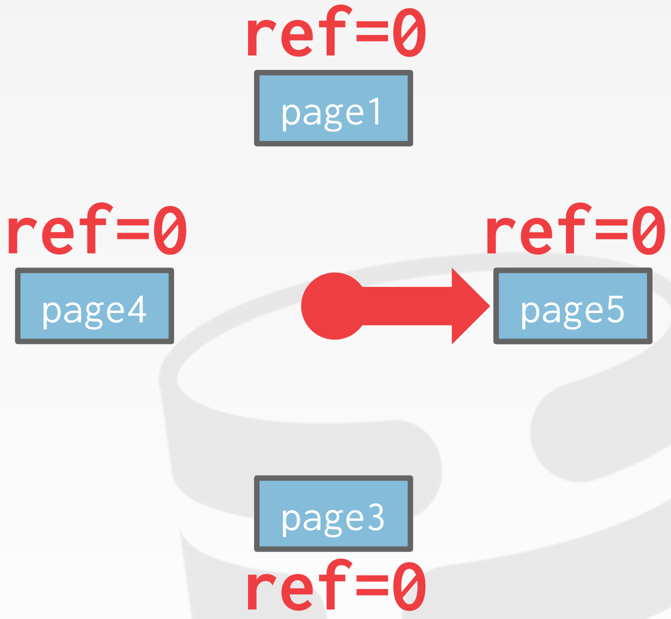
→ Each page has a reference bit.

→ 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 without needing a separate timestamp per page.

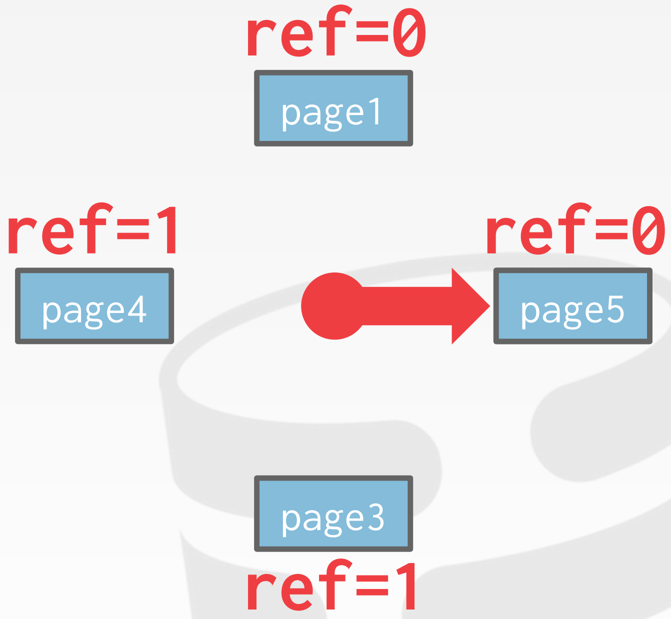
→ Each page has a reference bit.

→ 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 without needing a separate timestamp per page.

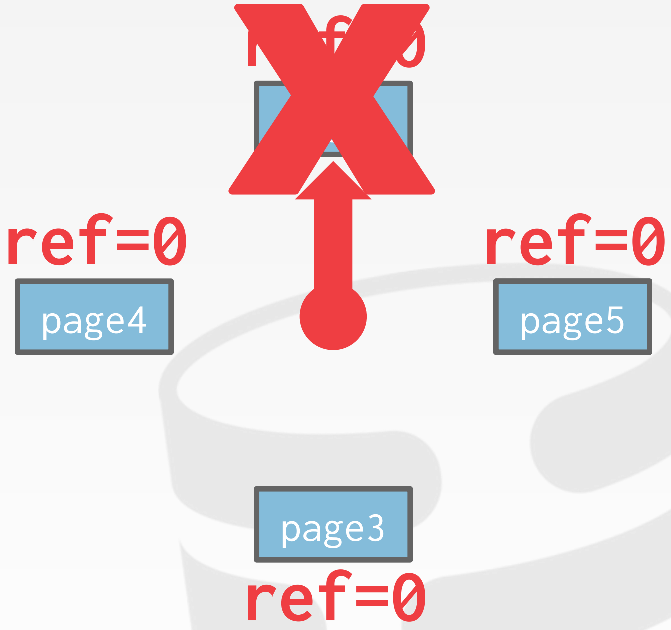
→ Each page has a reference bit.

→ 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

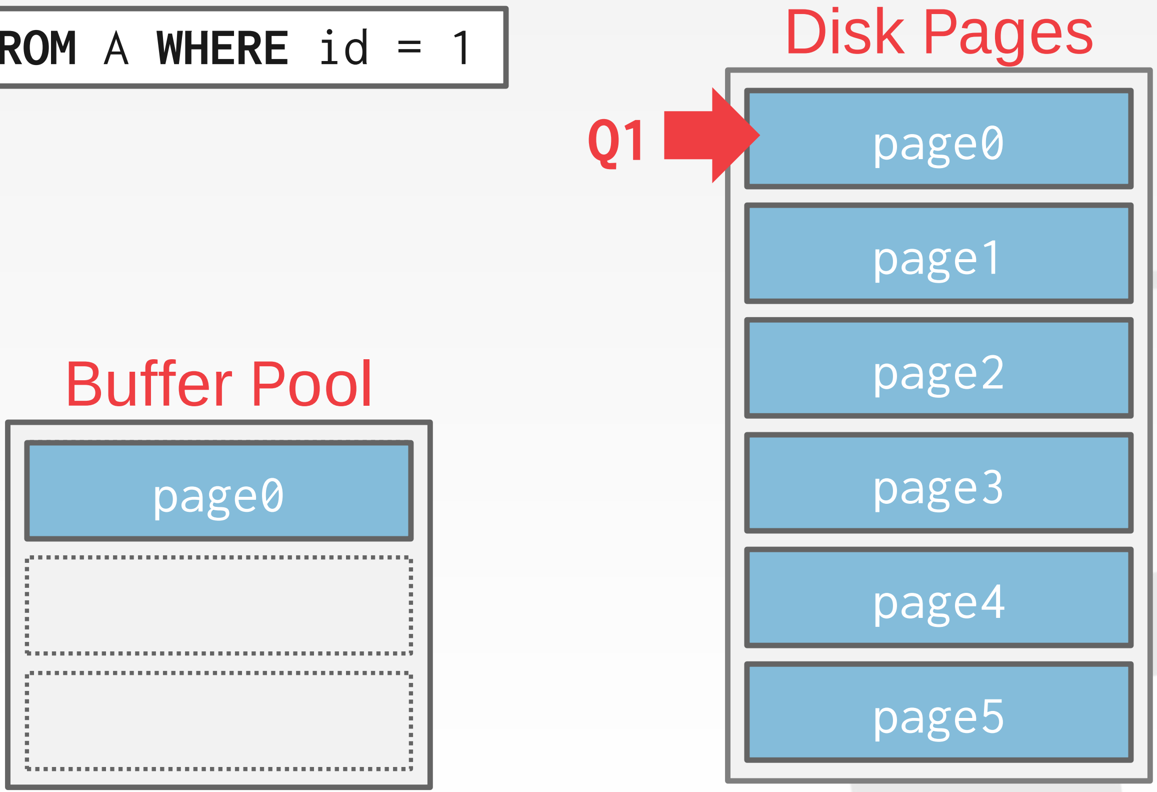
LRU and CLOCK replacement policies are susceptible to <u>sequential flooding</u>.

→ 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.

The most recently used page is actually the most unneeded page.

SEQUENTIAL FLOODING

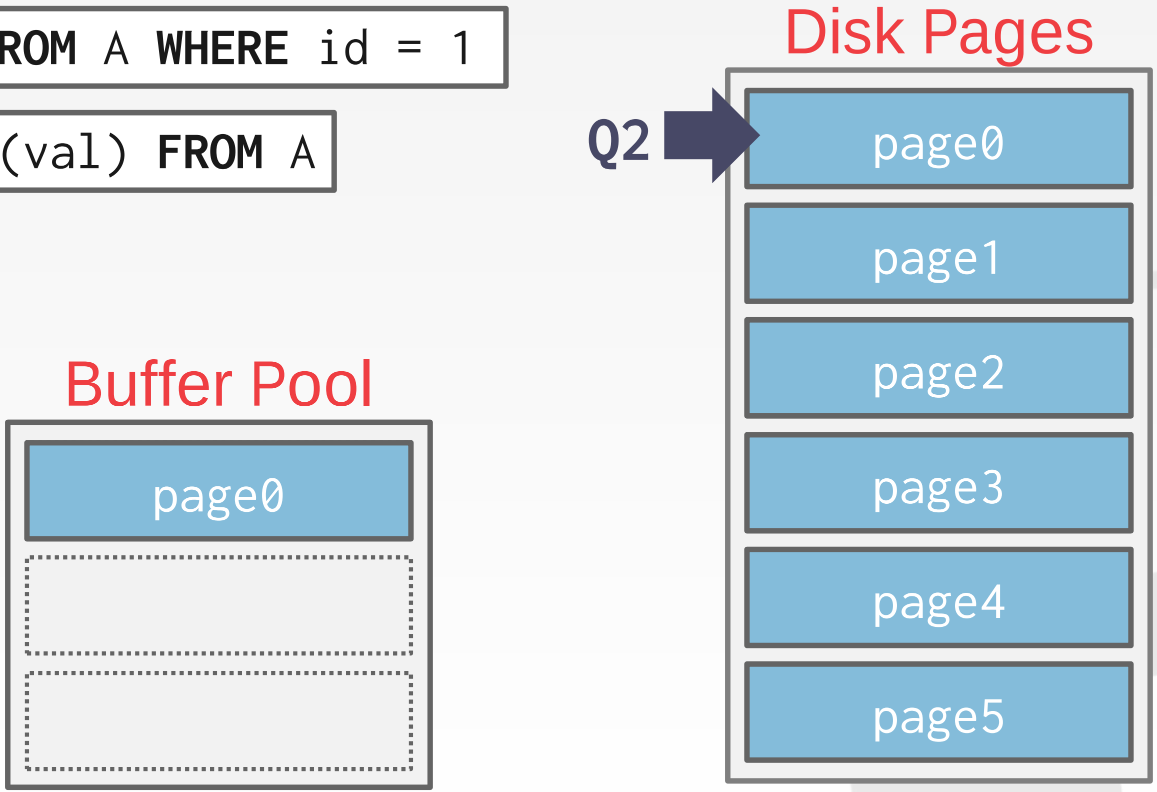
Q1 `SELECT * FROM A WHERE id = 1`



SEQUENTIAL FLOODING

Q1 `SELECT * FROM A WHERE id = 1`

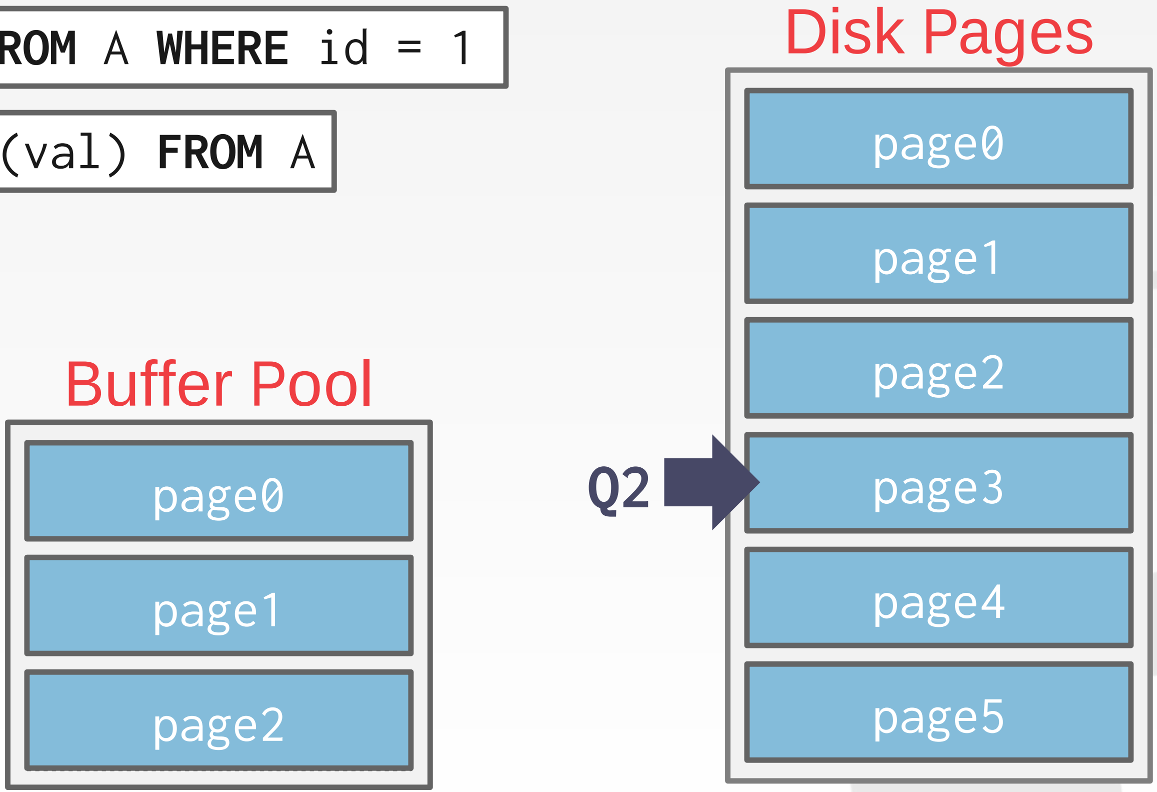
Q2 `SELECT AVG(val) FROM A`



SEQUENTIAL FLOODING

Q1 `SELECT * FROM A WHERE id = 1`

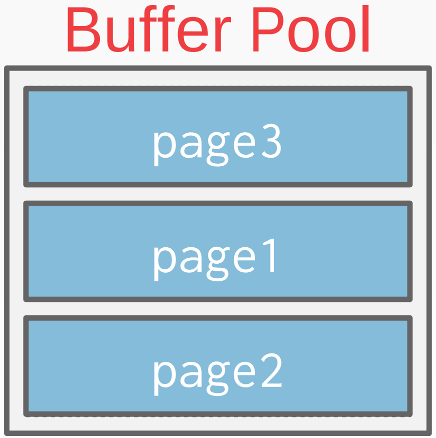
Q2 `SELECT AVG(val) FROM A`

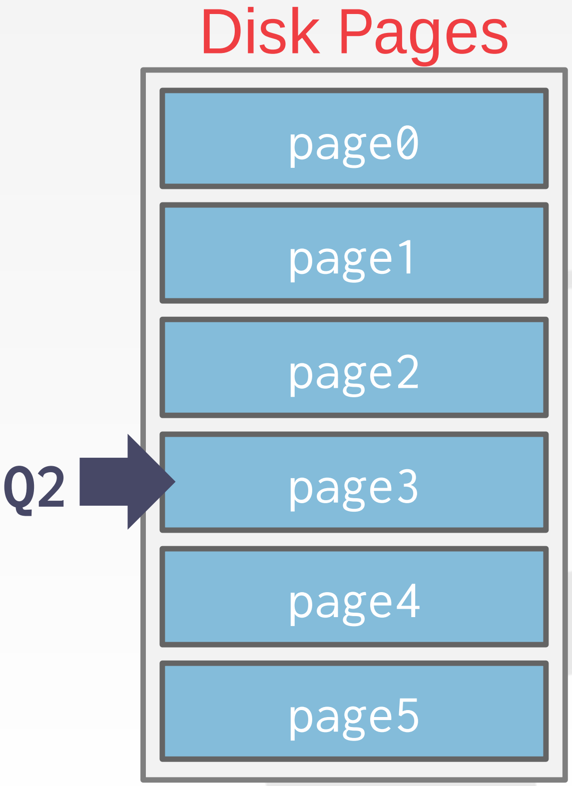


SEQUENTIAL FLOODING

Q1 `SELECT * FROM A WHERE id = 1`

Q2 `SELECT AVG(val) FROM A`



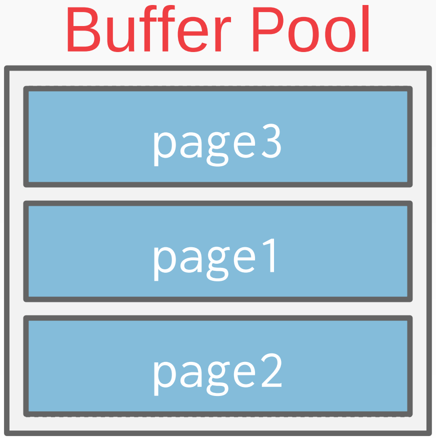


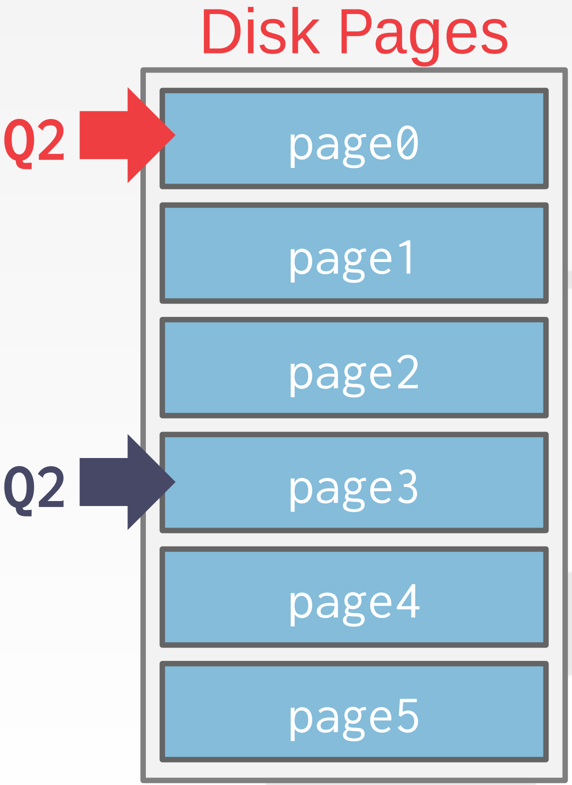
SEQUENTIAL FLOODING

Q1 `SELECT * FROM A WHERE id = 1`

Q2 `SELECT AVG(val) FROM A`

Q3 `SELECT * FROM A WHERE id = 1`



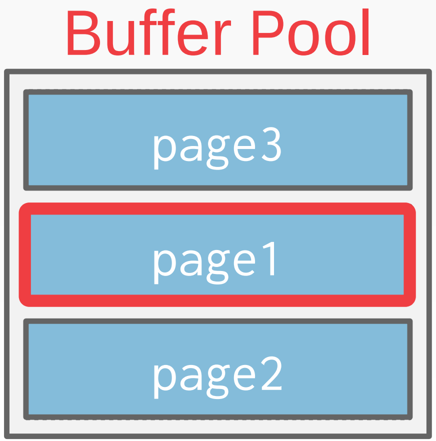


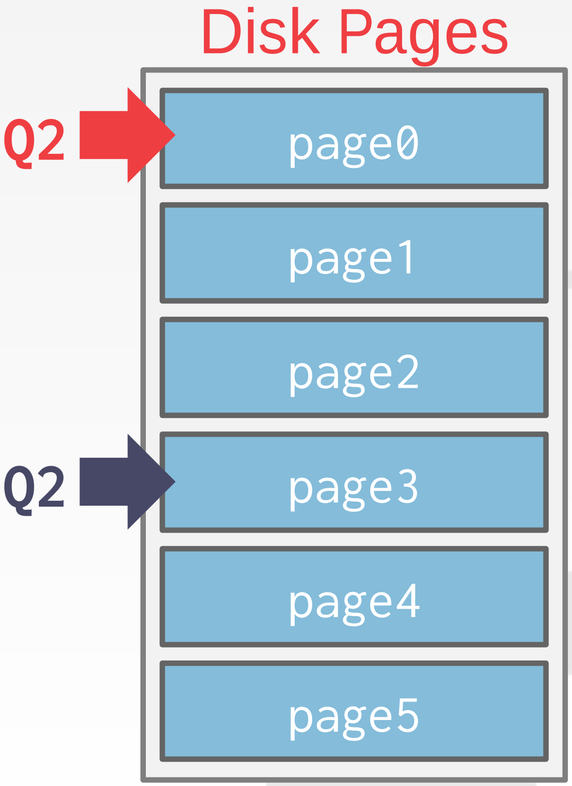
SEQUENTIAL FLOODING

Q1 `SELECT * FROM A WHERE id = 1`

Q2 `SELECT AVG(val) FROM A`

Q3 `SELECT * FROM A WHERE id = 1`





BETTER POLICIES: LRU-K

Take into account history of the last *K* references 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 what the context of each page during query execution.

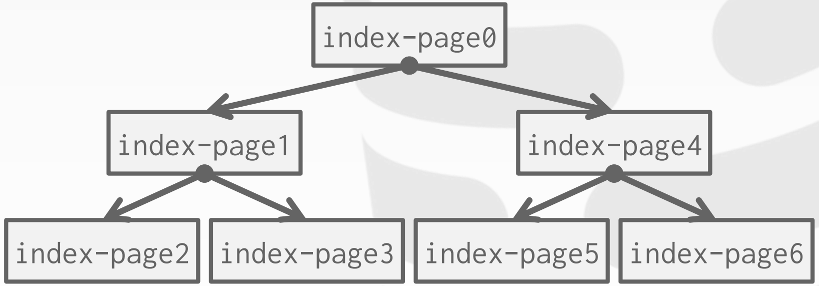
It can provide hints to the buffer pool on whether a page is important or not.

BETTER POLICIES: PRIORITY HINTS

The DBMS knows what 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++)`

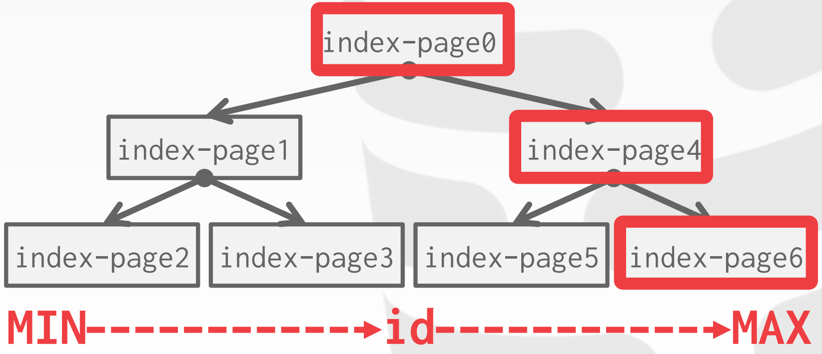


BETTER POLICIES: PRIORITY HINTS

The DBMS knows what 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++)
```



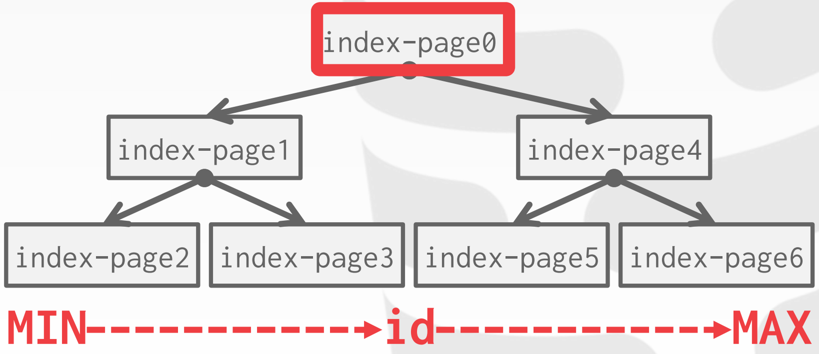
BETTER POLICIES: PRIORITY HINTS

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

SLOW: 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 we don't write dirty pages before their log records have been written…

ALLOCATION POLICIES

Global Policies:

→ Make decisions for all active txns.

Local Policies:

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

→ Still need to support sharing pages.

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 manage that sweet, sweet memory better than the OS.

Leverage the semantics about the query plan to make better decisions:
- → Evictions
- → Allocations
- → Pre-fetching

PROJECT #1

You will build the first component of your storage manager.

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

All of the projects are based on SQLite, but you will not be able to use your storage manger just yet after this first project.

SQLite

Due Date:
Wed Sept 26th @ 11:59pm

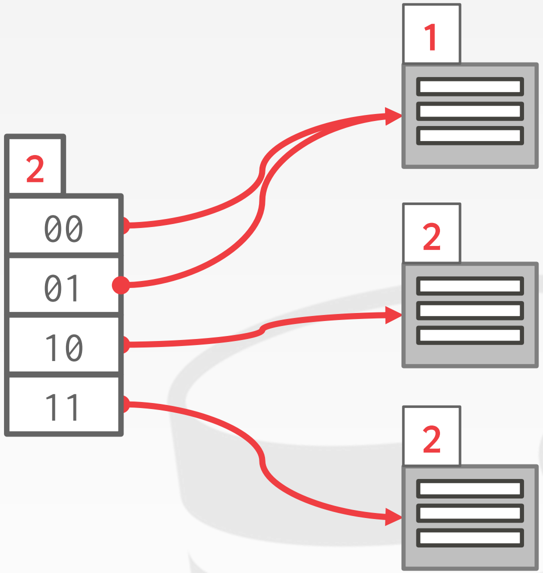
TASK #1 – EXTENDIBLE HASH TABLE

Build a thread-safe extendible hash 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 REPLACEMENT POLICY

Build a data structure that tracks the usage of **`Page`** objects in the buffer pool using the least-recently used policy.

General Hints:

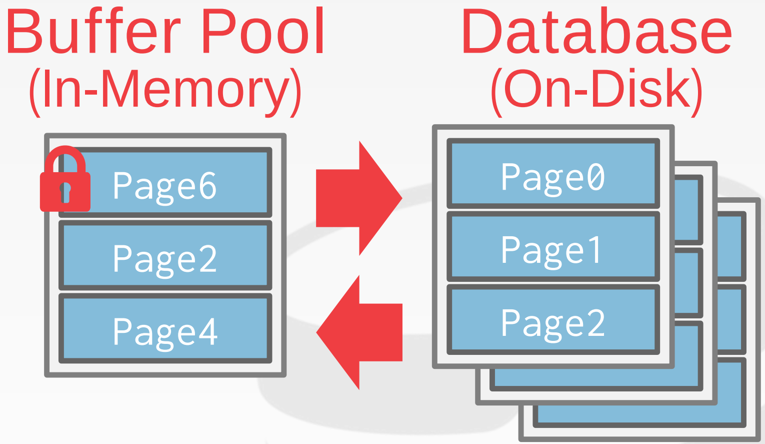
→ Your **`LRUReplacer`** does not need to worry about the "pinned" status of a **`Page`**.

TASK #3 – BUFFER POOL MANAGER

Combine your hash table and LRU replacer together to manage the allocation of pages.

→ Need to maintain an internal data structures of allocated + free pages.

→ We will provide you components to read/write data from disk.



General Hints:

→ Make sure you get the order of operations correct when pinning.

GETTING STARTED

Download the source code from the project webpage.

Make sure you can build it on your machine.

→ We've test it on Andrew machines, OSX, and Linux.

→ It should compile on Windows 10 w/ Ubuntu, but we haven't tried it.

THINGS TO NOTE

Do **not** change any file other than the six that you have to hand in.

The projects are cumulative.

We will **not** be providing solutions.

Post your questions on Piazza or come to our office hours. We will **not** help you debug.

PLAGIARISM WARNING

Your project implementation must be your own work.

→ You may **not** copy source code from other groups or the web.

→ Do **not** publish your implementation on Github.

Plagiarism will **not** be tolerated.
See CMU's Policy on Academic Integrity for additional information.

NEXT CLASS

HASH TABLES!