CS525: Advanced Database Organization

Notes 4: Buffer Pools

Yousef M. Elmehdwi

Department of Computer Science

Illinois Institute of Technology

yelmehdwi@iit.edu

September 13th 2023

Slides: adapted from a course taught by Andy Pavlo, Carnegie Mellon University

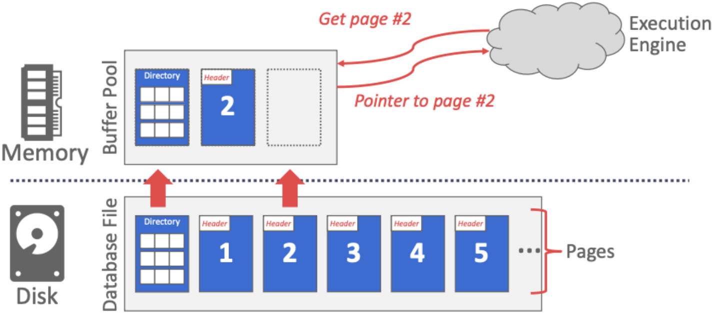
Reading

- Database Systems: The Complete Book, 2nd Edition,
  - *Chapter 2: Data Storage*
- Database System Concepts (6th/7th Edition)
  - *Chapter 10 (6th)/ Chapter 13 (7th)*

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.

Disk Oriented DBMS



Disk Oriented DBMS: Some fundamental Challenges & Goals

- DBMS Memory Management
  - The DBMS is responsible for efficiently managing memory, which includes bringing data from disk into memory and handling data operations.
  - Data stored on disk cannot be directly operated on, so the DBMS must have mechanisms to load and manage data in memory.
- Efficient Data Movement
  - One of the challenges DBMSs face is minimizing the slowdown caused by moving data between disk and memory.
  - Efficient data movement is crucial for maintaining system performance, especially when dealing with large datasets.
- Data Accessibility in Memory
  - Ideally, a well-designed DBMS should make it seem as though all data is already in memory.
  - This illusion of data residing in memory ensures that data operations are performed at the highest possible speed.

Database Storage

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

- Spatial Control (Where to Write Pages on Disk)
    - Involves deciding where to physically store pages (blocks of data) on the disk.
    - The primary objective is to optimize data layout on disk to keep pages that are accessed together often closely located as possible.
    - *By storing related pages in proximity to each other, you reduce seek times and improve data retrieval performance.*
- Temporal Control (When to Read and Write Pages)
    - Focuses on managing when to bring pages into memory (read) and when to flush them back to disk (write).
    - The goal is to minimize the number of stalls or delays that occur when data must be read from disk because it's not available in memory.
    - *Efficient caching, prefetching, and write policies are used to achieve this goal.*

Database Storage

Spatial control optimizes the physical layout of data on disk to enhance data retrieval efficiency, while temporal control optimizes data movement between memory and disk to minimize delays in accessing data. Both aspects are crucial for improving the overall performance of a DBMS, especially when dealing with large datasets where efficient memory management becomes critical.

Today's Agenda

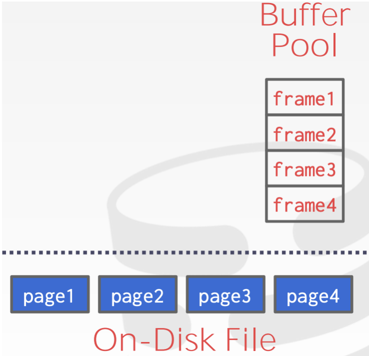
- Buffer Pool Manager
- Replacement Policies
- Other Memory Pools

Buffer Pool

- The DBMS always knows better so we want to manage memory and pages ourselves.
- Buffer pool
  - Integral part of a DBMS and serves as an in-memory cache for pages retrieved from the disk.
  - It is essentially a large memory region allocated inside of the DBMS to store pages from the database that have been fetched (read) from disk.
  - The primary purpose of a buffer pool is to improve the efficiency of data access by reducing the need for frequent and redundant disk I/O operations.

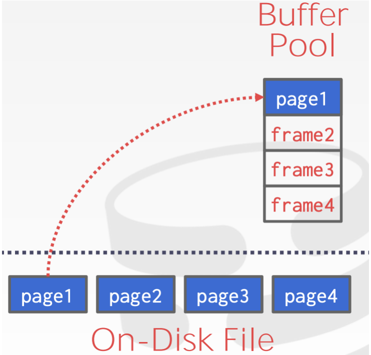
Buffer Pool Organization

- The buffer pool's region of memory region organized as an array of fixed-size pages.
- Each array entry is called a `frame`.



Buffer Pool Organization

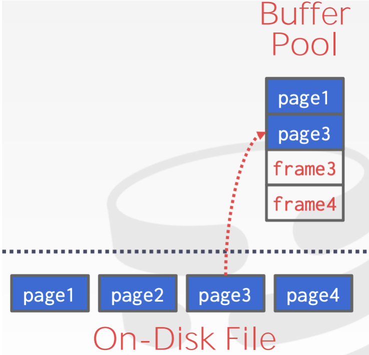
- When the DBMS requests a page, an exact copy is placed into one of these frames.
- Then, the database system can search the buffer pool first when a page is requested.



Buffer Pool Organization

- When the DBMS requests a page, an exact copy is placed into one of these frames.
- Then, the database system can search the buffer pool first when a page is requested.
- If the page is not found, then the system fetches a copy of the page from the disk.

- Pages on buffer pool can be in any order not as the same order on disk.

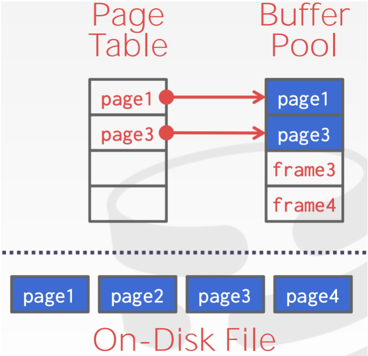


Buffer Pool Meta-Data

- The buffer pool must maintain certain meta-data in order to be used efficiently and correctly.
- Metadata maintained by the buffer pool:
  - Page Table
    - In-memory hash table that keeps track of pages that are currently in memory.
    - Its purpose is to keep track of which pages are currently resident in memory (in the buffer pool).
    - It maps page ids to frame locations in the buffer pool
    - Since the order of pages in the buffer pool does not necessarily reflect the order on the disk, this extra indirection layer allows for the identification of page locations in the pool.
  - Dirty-flag
    - A bit that identify if a page in the memory has been modified or not.
    - Gets set when a thread modifies a page (will need to be written back).
    - This indicates to storage manager that the page must be written back to disk.
  - Pin/Reference Counter
    - Track the number of threads that are currently accessing that page (either reading or modifying it).
    - A thread has to increment the counter before they access the page.
    - If a page's count is greater than zero, then the storage manager is **not** allowed to evict that page from memory

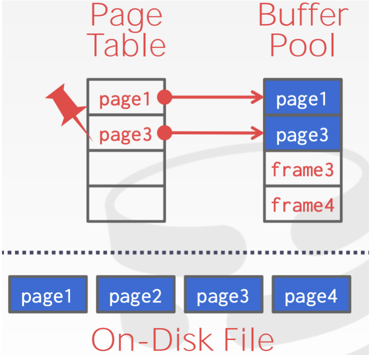
Buffer Pool Meta-Data

- The page table keeps track of pages that are currently in memory.
    - *what pages are currently in memory and the location of these pages*
- Also maintains additional meta-data per page:
    - Dirty Flag
    - Pin/Reference Counter



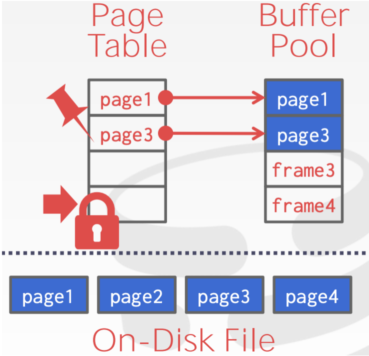
Buffer Pool Meta-Data

- 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

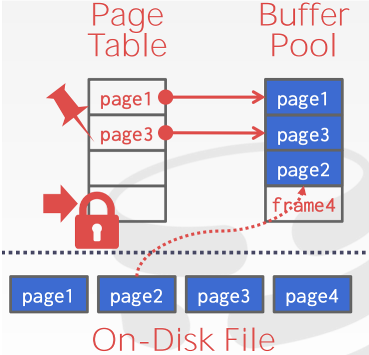
- When a thread tries to read a page not currently in memory (i.e., the page is a cache miss)
    - put a latch on this entry in the hash table to indicate that the entry is occupied or in use.



- *The latch serves as a synchronization mechanism to ensure that only one thread at a time can attempt to load the missing page into memory. To prevent multiple threads from attempting to load the same page into memory concurrently, the thread puts a latch or lock on the entry in the hash table corresponding to the missing page.*

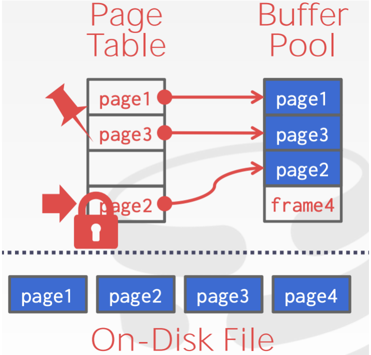
Buffer Pool Meta-Data

- When a thread tries to read a page not currently in memory
    - put a latch on this entry in the hash table to indicate that the entry is occupied
    - fetch the page to the buffer pool



Buffer Pool Meta-Data

- When a thread tries to read a page not currently in memory
    - put a latch on this entry in the hash table to indicate that the entry is occupied
    - fetch the page to the buffer pool
    - update the entry in the hash table to point to the fetched page in pool, and release the latch



Locks vs. Latches

We need to make a distinction between locks and latches when discussing how the DBMS protects its internal elements.

- Locks
    - A higher-level, logical primitive that protects the database's contents (e.g., tuples, tables, indexes) from other transactions.
        - *when a transaction is being made to manipulate data, a lock is responsible to protect the logical content of the DB that are currently being manipulated from other transactions*
    - Transactions will hold a lock for its entire duration
    - Need to be able to rollback changes.
- Latches
    - A low-level primitive that the DBMS uses to protects the critical sections of the DBMS's internal data structure (e.g., page table, region of memory) from other threads.
    - Typically held for the duration of a specific operation or critical section within the DBMS.
    - Used to synchronize access to internal data structures.
    - Do not need to be able to rollback changes

Page Table vs. Page Directory

- Note that the `page table` is not to be confused with the `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

Memory Allocation Policies

- Memory allocation is a critical aspect of database systems, and these policies help determine how memory is allocated to optimize performance.
- Memory in the database is allocated for the buffer pool according to two policies:
    - Global (memory allocation)Policies
    - Local (memory allocation) Policies
    - Combination of Policies

Memory Allocation Policies

- Global (memory allocation)Policies
  - Focus on making decisions that benefit the entire workload being executed by the DBMS.
  - Consider all active transactions, queries, and processes for optimal memory allocation.
  - This often involves decisions like how much memory to allocate to buffer pool, resource distribution among transactions, and addressing memory contention.
  - The aim is to efficiently use memory for improved overall system performance.

Memory Allocation Policies

- Local (memory allocation) Policies
    - Make decisions that prioritize the performance of individual queries or transactions, even if those decisions may not be ideal for the entire workload.
    - Allocate memory to specific transactions or queries without considering the concurrent transactions.
    - Aim to enhance response time and execution speed of individual tasks, crucial for interactive applications, while maintaining data consistency.
    - Systems employing local policies must still support mechanisms for sharing pages among transactions, ensuring data consistency.

Memory Allocation Policies

- Combination of Policies
  - Many modern DBMSs use a combination of both global and local views.
  - This hybrid approach aims to balance overall system performance and individual transactions or queries.
  - Global policies ensure that resources are distributed efficiently, while local policies improve the responsiveness of critical tasks.

Buffer Pool Optimizations

- There are a number of ways to optimize a buffer pool to tailor it to the application's workload.
- Multiple Buffer Pools
  - The DBMS can also have multiple buffer pools for different purposes. This helps reduce latch contention and improves locality
- Pre-Fetching
  - The DBMS can also optimize by pre fetching pages based on the query plan. Commonly done when accessing pages sequentially.
- Scan Sharing
  - Query cursors can attach to other cursors and scan pages together
- Buffer Pool Bypass
  - A sequential scan of large data tables that bypasses the overhead of the buffer pool.

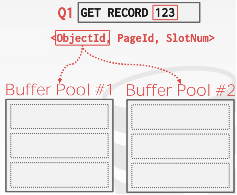
Multiple Buffer Pools

- The DBMS does not always have a single buffer pool for the entire system.
    - Multiple buffer pool instances
        - Each page that is stored in or read from the buffer pool is assigned to one of the buffer pools randomly using a hashing function/Object ID.
    - Per-database buffer pool
    - Per-page type buffer pool
- Each buffer pool can adopt local policies tailored for the data stored inside of it
- Helps reduce latch contention and improve locality.
- e.g., `IBM DB2, ORACLE, SQL Server, MySQL`

Multiple Buffer Pools

- Two approaches to mapping desired pages to a buffer pool are object IDs[1] and hashing.

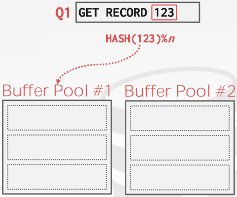
- Approach#1: Object ID
  - Embed an object identifier in record ids and then maintain a mapping from objects to specific buffer pools
  - Example: `<ObjectId, PageId, SlotNum>`
  - ObjectId → Buffer Pool Number



[1] *involve extending the record IDs to include meta-data about what database objects each buffer pool is managing. Then through the object identifier, a mapping from objects to specific buffer pools can be maintained.*

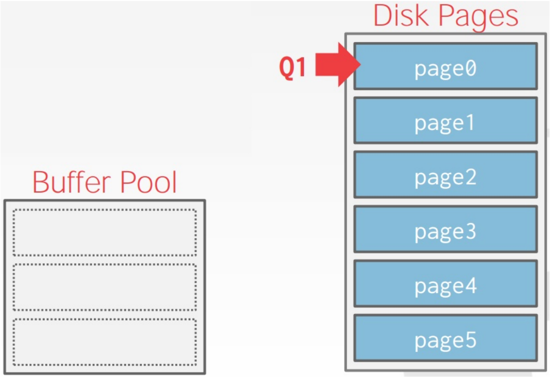
Multiple Buffer Pools

- Approach#2: Hashing
  - Hash the page id to select which buffer pool to access.
  - Example: `HASH(PageId) % (Number of Buffer Pools)`



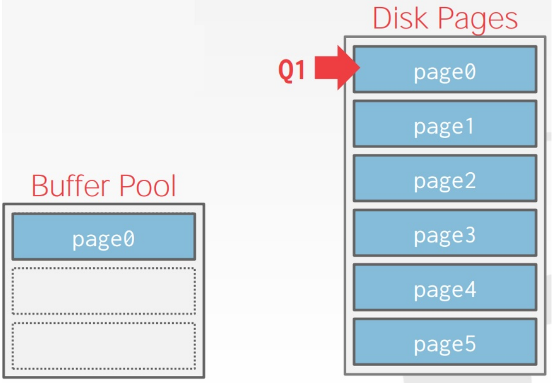
Pre-Fetching

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



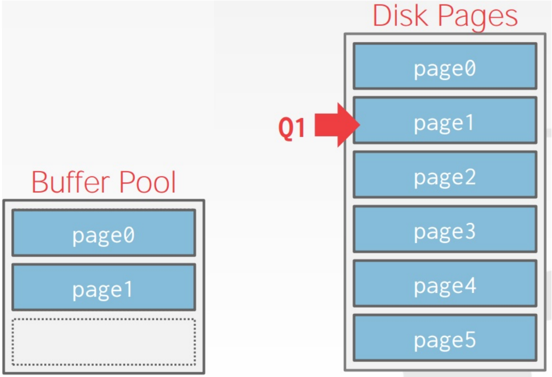
Pre-Fetching

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



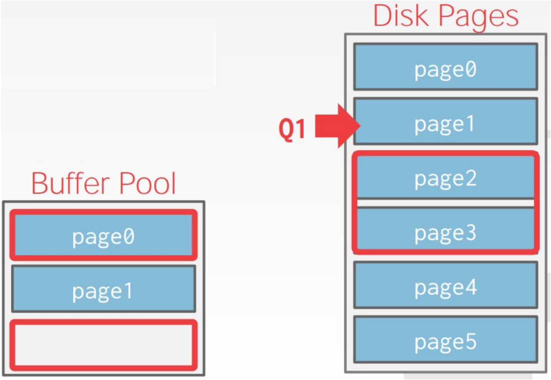
Pre-Fetching

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



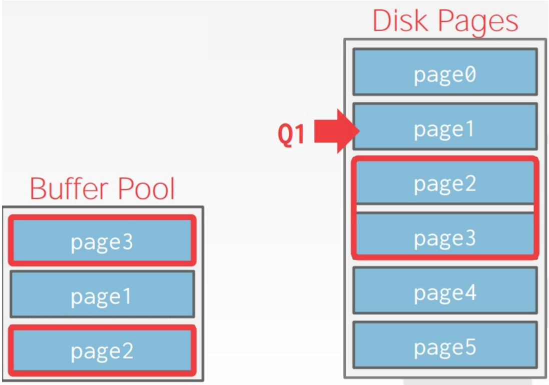
Pre-Fetching

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



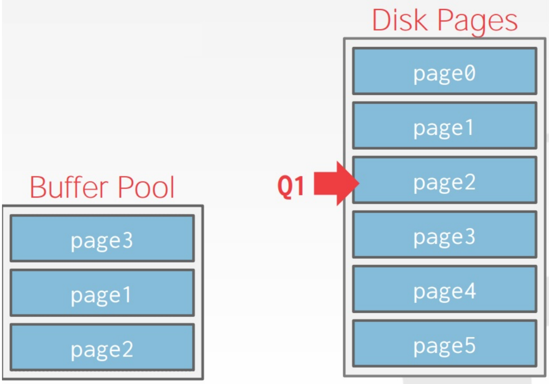
Pre-Fetching

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



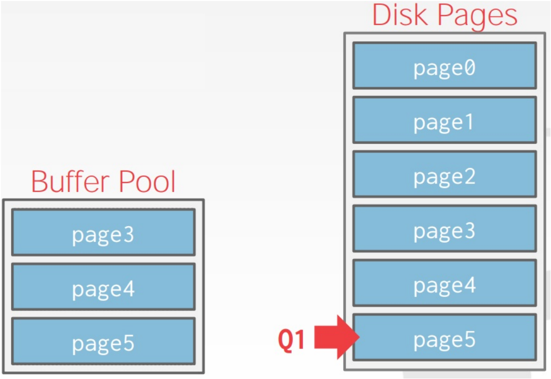
Pre-Fetching

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



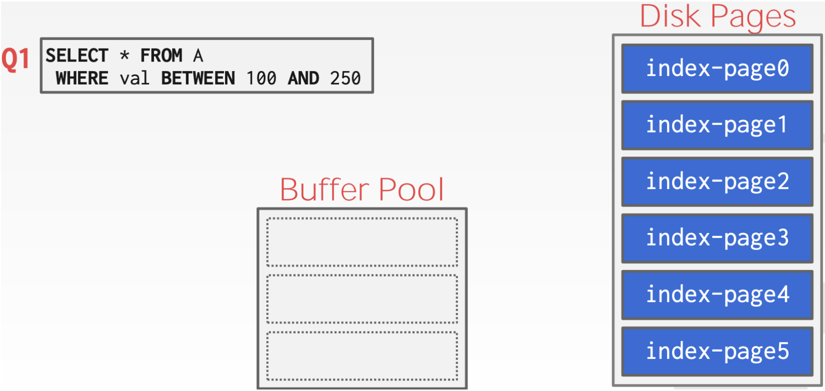
Pre-Fetching

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



Pre-Fetching

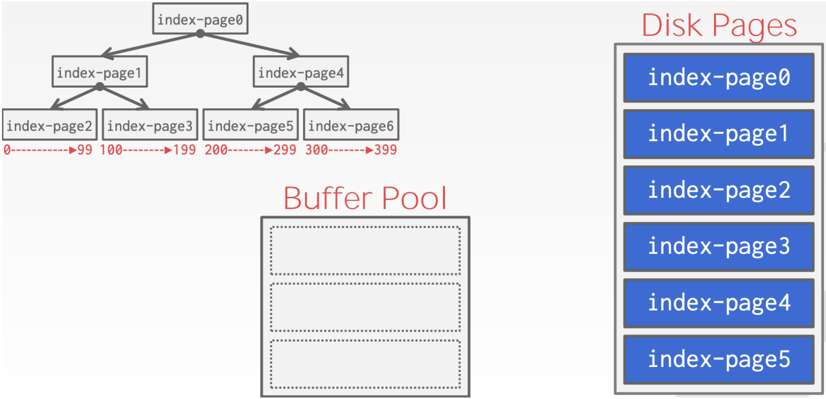
Index Scans: Run range scan



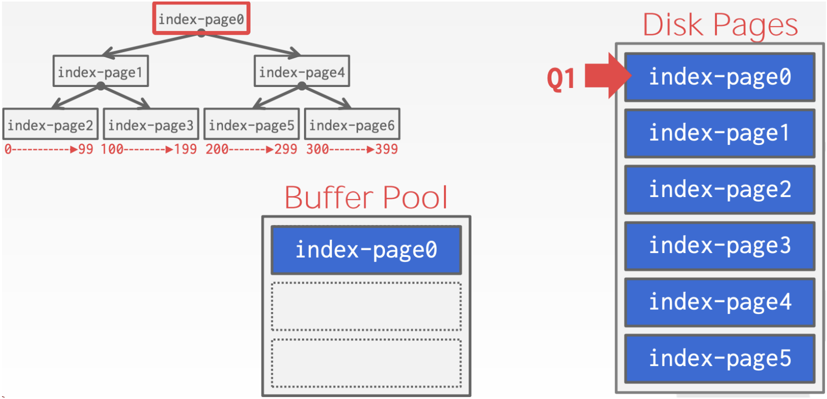
Q1

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

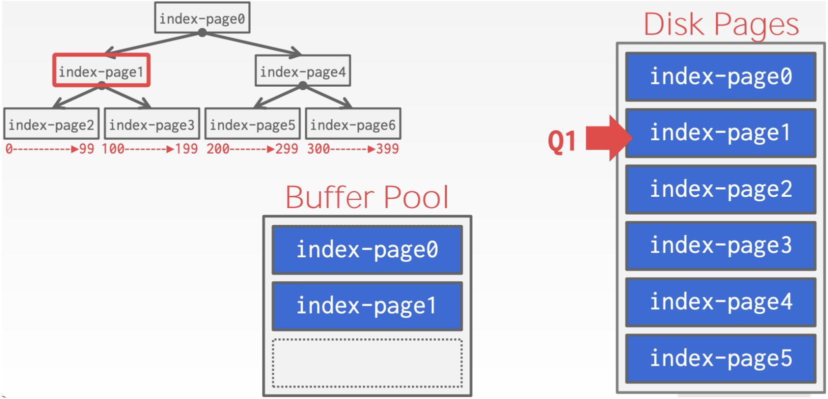
Pre-Fetching



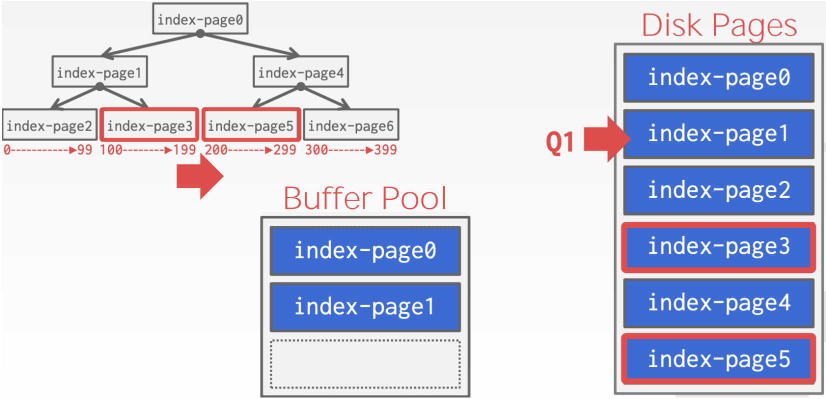
Pre-Fetching



Pre-Fetching



Pre-Fetching



Scan Sharing

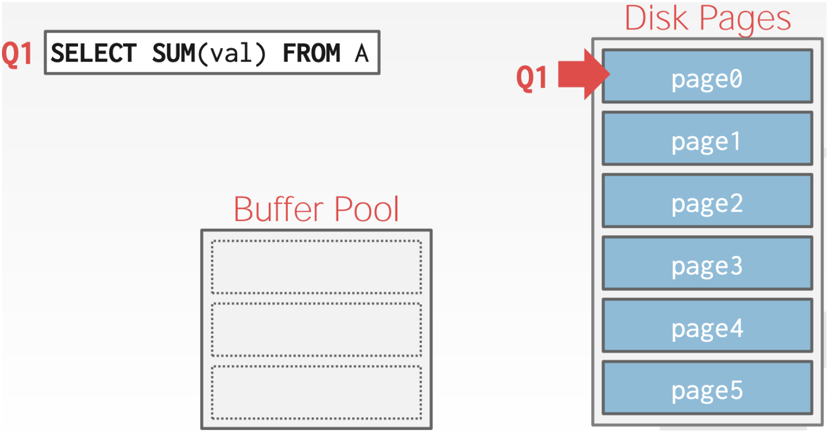
- Queries are able to **reuse data** retrieved from storage or operator computations
  - Also called `synchronized scans`.
  - This is different from result caching.
- Allow multiple queries to attach to a single cursor that scans a table.
  - Queries do not have to be exactly the same.
  - Can also 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`, `MS SQL Server`, and `PostgreSQL`[1].
- Oracle only supports cursor sharing for identical queries.

[1] synchronize_seqscans (boolean): https://www.postgresql.org/docs/current/runtime-config-compatible.html

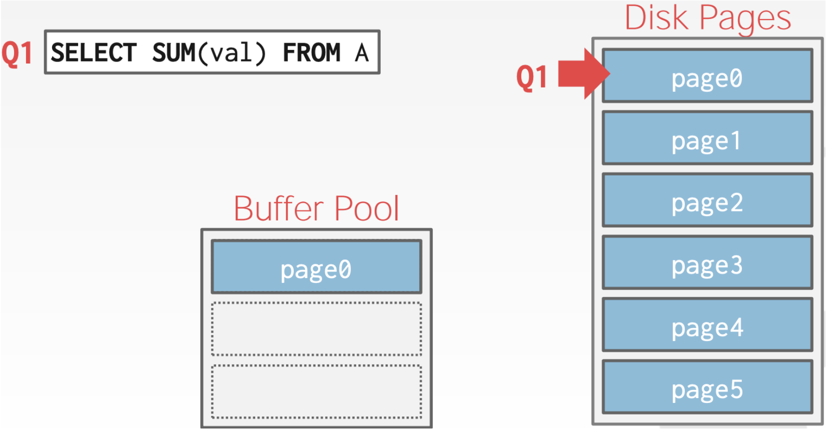
Scan Sharing



Q1

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

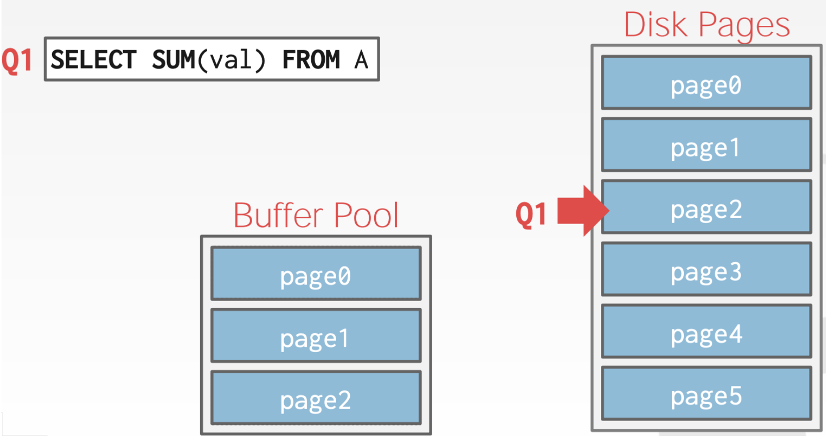
Scan Sharing



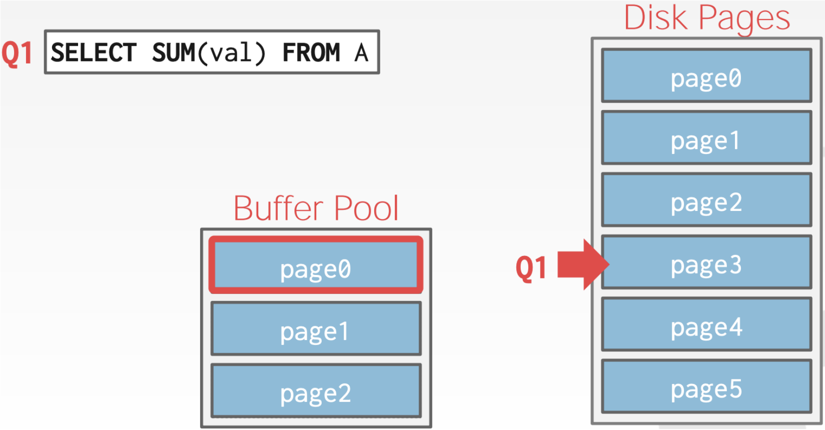
Q1

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

Scan Sharing

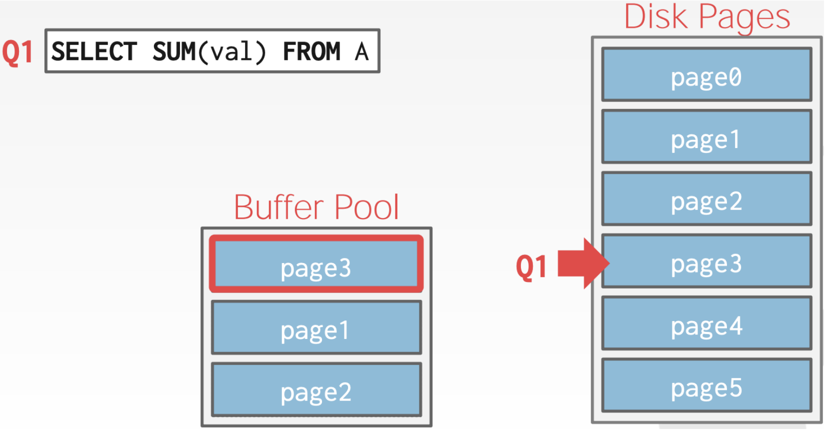


Scan Sharing



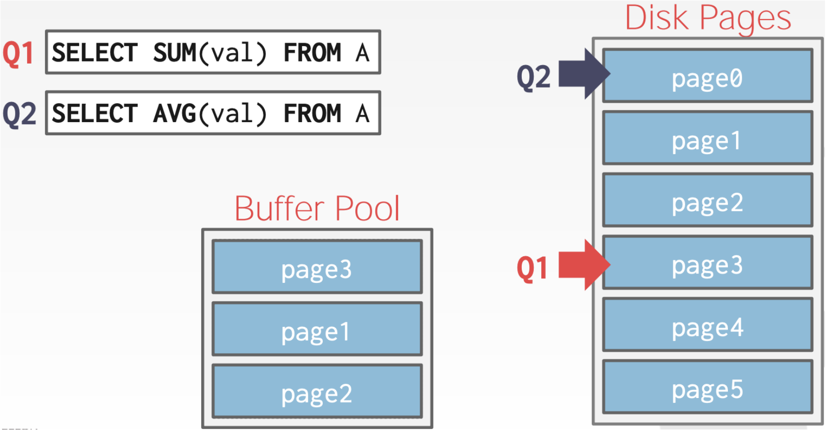
Q1 `SELECT SUM(val) FROM A`

Scan Sharing



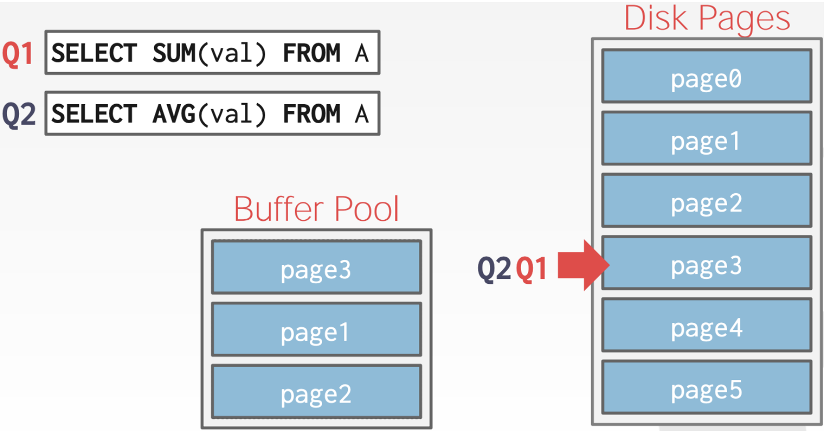
Q1 `SELECT SUM(val) FROM A`

Scan Sharing


Q1 `SELECT SUM(val) FROM A`

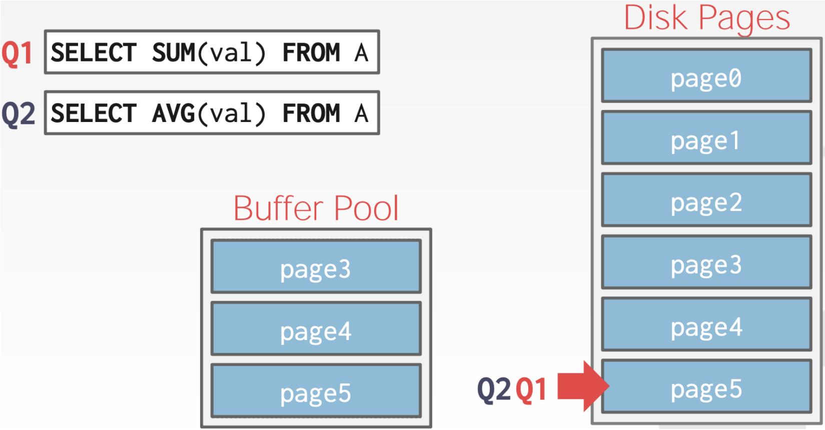
Q2 `SELECT AVG(val) FROM A`

Scan Sharing



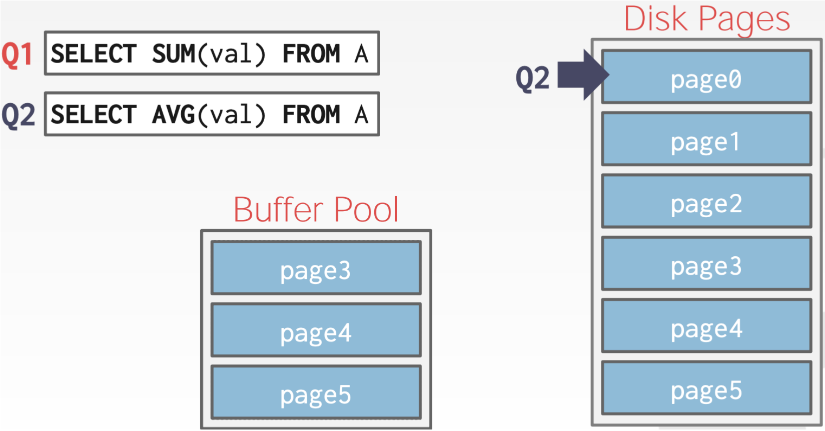
```
Q1 SELECT SUM(val) FROM A
Q2 SELECT AVG(val) FROM A
```

Scan Sharing



```
Q1 SELECT SUM(val) FROM A
Q2 SELECT AVG(val) FROM A
```

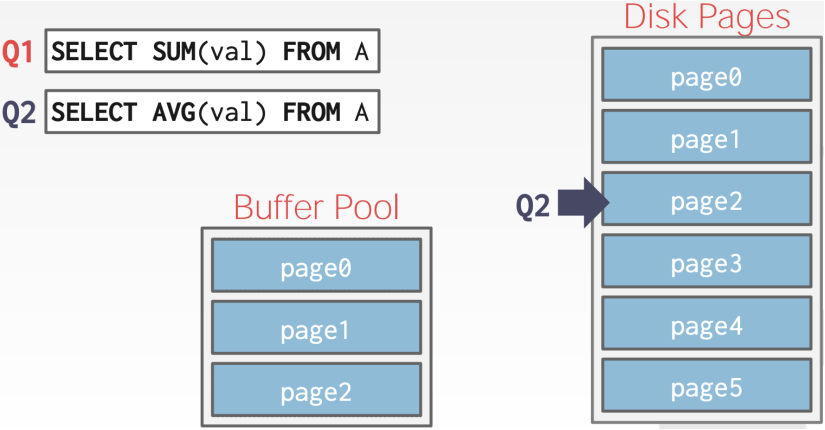
Scan Sharing



Q1 `SELECT SUM(val) FROM A`

Q2 `SELECT AVG(val) FROM A`

Scan Sharing



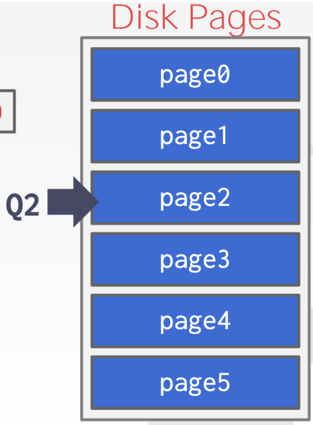
Q1 SELECT SUM(val) FROM A

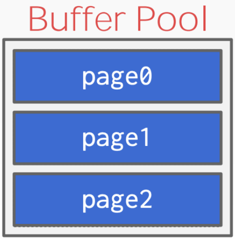
Q2 SELECT AVG(val) FROM A

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).
- Supported in `Oracle`, `MS SQLServer`, `PostgreSQL`, and `Informix`
- Called Light Scans[2] in `Informix`.

[2] IBM Informix Server 14.1

OS Page Cache

- Most disk operations go through the OS API.
- Unless you tell it not to, the OS maintains its own filesystem cache.
  - Called `OS Page Cache`
- Most DBMSs use direct I/O (`O_DIRECT flag`) to bypass the OS's cache
  - Redundant copies of pages.
  - Different eviction policies.
- `Postgres` as a cross platform database, relies heavily on the operating system for its caching.

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.
- A replacement policy is an algorithm that the DBMS implements that makes a decision on which pages to evict from buffer pool when it needs space.
- Implementation goals:
  - Correctness
  - Accuracy
  - Speed
  - Meta-data overhead

First In, First Out (FIFO)

- One of the simplest page replacement algorithm.
- Replace page that has been in the buffer for the longest time
  - *the oldest page, which has spent the longest time in memory is chosen and replaced.*
- Implementation: E.g., pointer to oldest page (circular buffer)
  - **`Pointer->next = Pointer++ %`** `M`, where `M` denote the buffer size in pages
- Simple, but not prioritizing frequently accessed pages

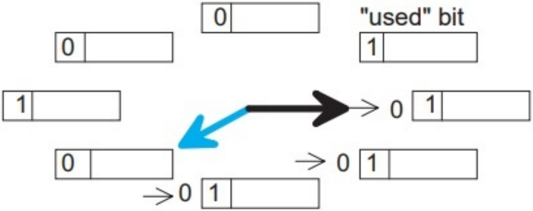
Least Recently Used (LRU)

- 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.
- Implementation
    - List, ordered by LRU
    - Access a page, move it to list tail
    - Keep the pages in sorted order to reduce the search time on eviction
    - DBMS picks to evict the page with the oldest timestamp

Clock

- Approximation of LRU, but much optimized and run much faster, without needing a separate timestamp per page.
  - Each page has a reference bit.
  - When a page is accessed, set to 1
    - *keep a flag for each page loaded in the pool. This flag indicates whether the page was referenced or not*
- Organize the pages in a circular buffer with a "clock hand"
  - Upon sweeping/looping on pages in the buffer pool, check if a page's reference flag is set to 1.
  - If yes, set to zero and and continue the loop. If no, then evict.
  - Clock hand remembers position between evictions

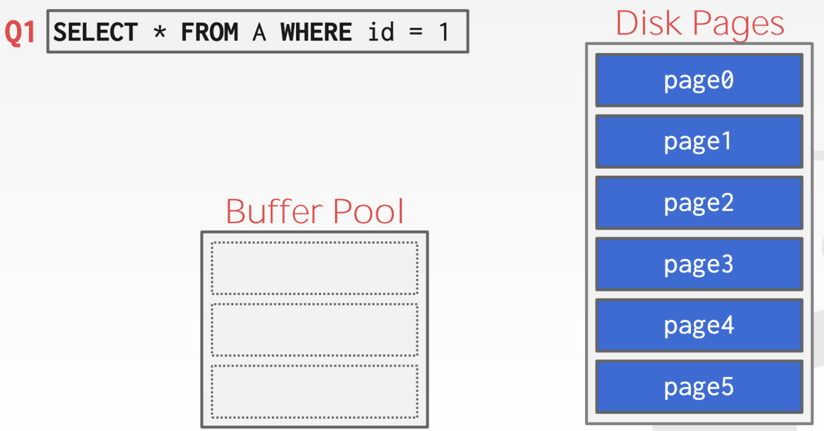
Clock: “second chance”



Problems

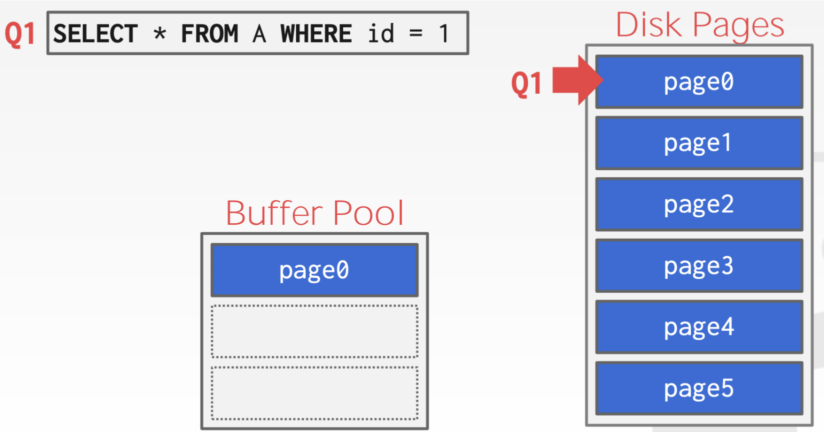
- Problems with `LRU` and `Clock` replacement policies:
  - `LRU` and `Clock` replacement policies are susceptible to sequential flooding where the buffer pool's contents are trashed due to a sequential scan.
    - A query performs a sequential scan that reads every page.
    - This pollutes the buffer pool with pages that will only be used once and then never again
  - Since sequential scans read every page, the timestamps of pages read may not reflect which pages we actually want.
    - *The most recently used page is actually the most unneeded page.*

Sequential Flooding



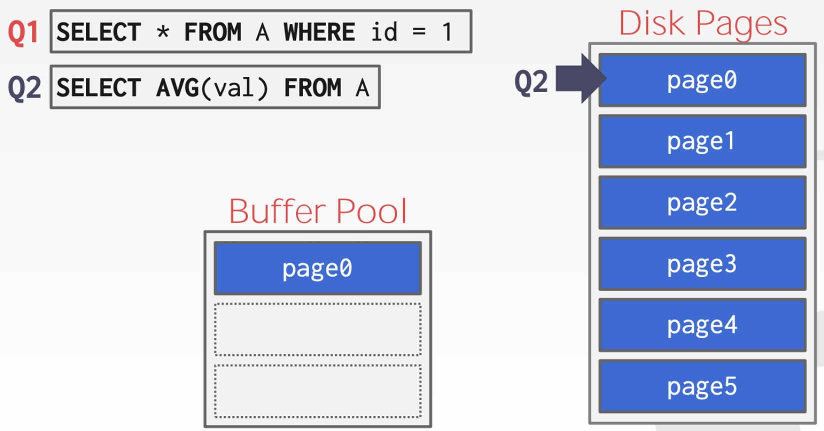
Q1
```
SELECT * FROM A WHERE id = 1
```

Sequential Flooding



Q1 `SELECT * FROM A WHERE id = 1`

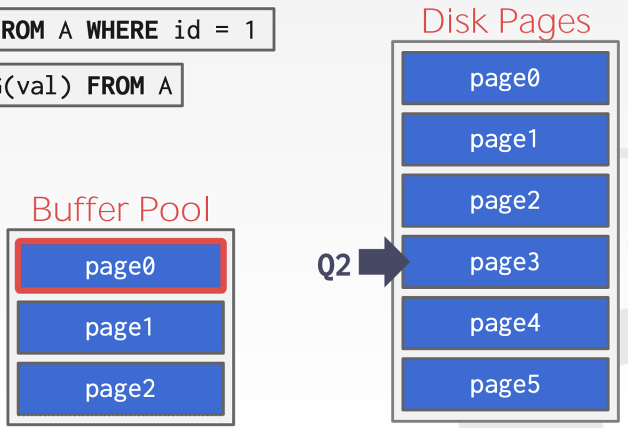
Sequential Flooding



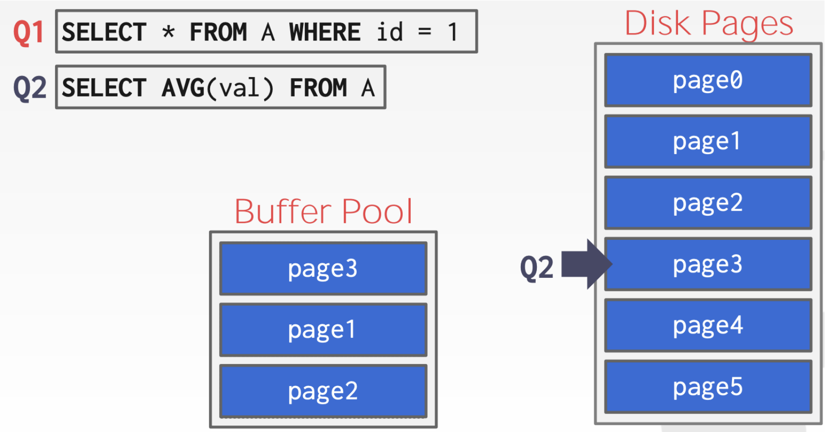
Sequential Flooding

Q1 `SELECT * FROM A WHERE id = 1`

Q2 `SELECT AVG(val) FROM A`



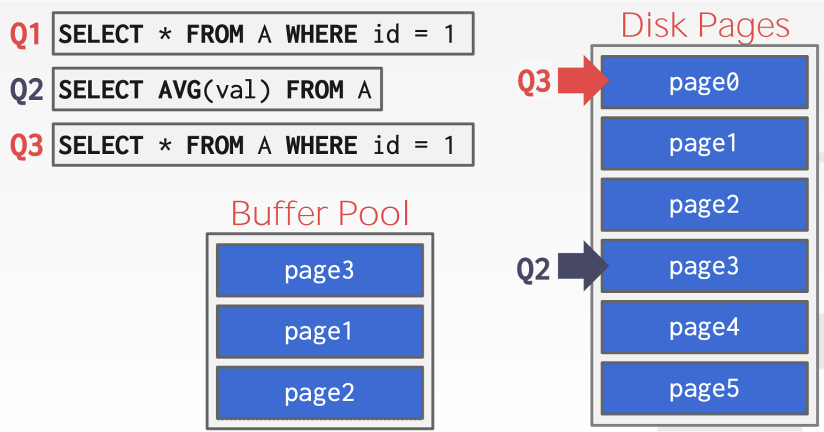
Sequential Flooding



Q1 `SELECT * FROM A WHERE id = 1`

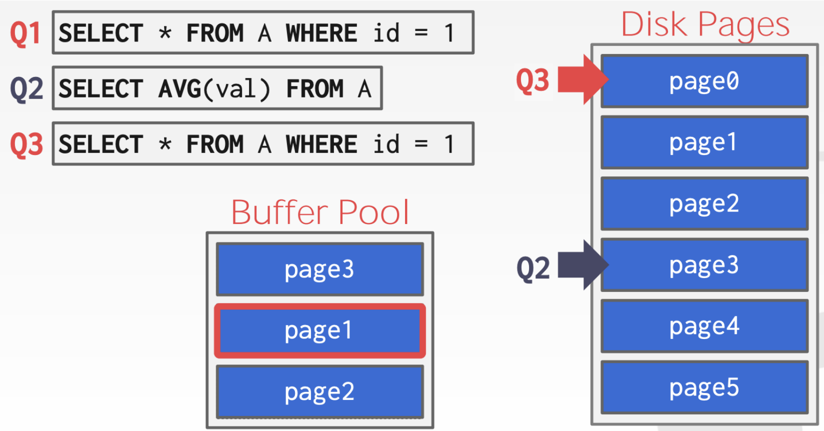
Q2 `SELECT AVG(val) FROM A`

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/solutions

- `LRU-K`: Take into account history of the last `K` references.
- `Localization`: Choose pages to evict on a per txn/query basis.
- `Priority hints`: Allow txns to tell the buffer pool whether page is important or not.

Better Policies: `LRU-K`

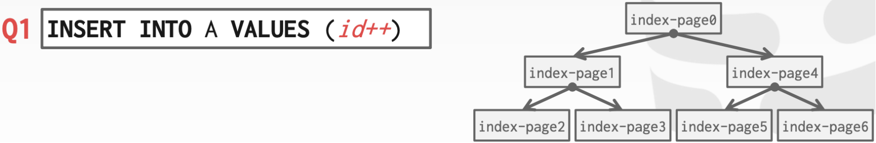
- Take into account/Track the history of the last `K` references to each page as timestamps and compute the interval between subsequent accesses.
- The DBMS then uses this history to estimate the next time that page is going to be accessed.

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 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.

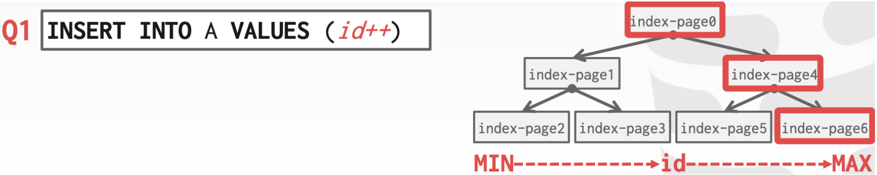


Q1

```
INSERT INTO A VALUES (id++)
```

Better Policies: Priority Hints

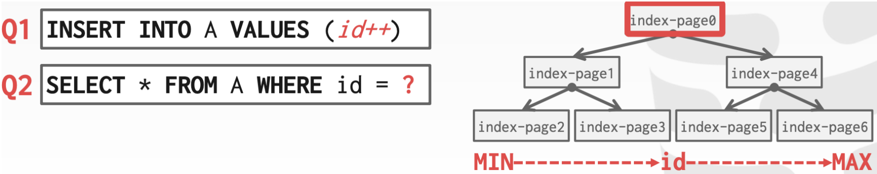
- The DBMS knows what the context of each page during query execution.
- It can provide hints to the buffer pool on whether a page is important or not.



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

- There are two ways to make a decision which page to evict:
  - **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.

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

Next

- How to support the DBMS's execution engine to read/write data from pages