Physical Storage: Buffer Manager

COM 3563: Database Implementation

Avraham Leff

Yeshiva University

avraham.leff@yu.edu

COM3563: Fall 2020

Today's Lecture: Overview

1. Disk-Space Management (Buffer Manager)
2. Buffer Page Replacement Policies
3. Pre-Fetching
4. Bonus: "Why Not Use The OS?"

Some figures taken from lectures created by Professor Zvi Kedem, used with permission

Some material from lectures created by Professor Pavlo (CMU), used with permission

Context

- Previous lecture began discussion of physical storage
  - Focused on characteristics & performance implications of various physical media, with a focus on HDD
  - The existence of, and approaches for dealing with, the memory hierarchy
    - Each level of the hierarchy serves as a cache for lower levels
    - Reality of physical media's specific performance characteristics and how these "facts" can have multiple, and varying tradeoffs

Today's Lecture

- Continue discussion of physical storage: how does DBMS deal with the reality of (usually) "data can't simply all be dumped into main-memory"
- Dig deeper into how data are moved back-and-forth from disk ⇔ main-memory
  - Also: "why not simply delegate to the OS?"
- Discuss buffer manager role and responsibilities
  - With a focus on page replacement policies

Disk-Space Management (Buffer Manager)

Disk Oriented Architecture

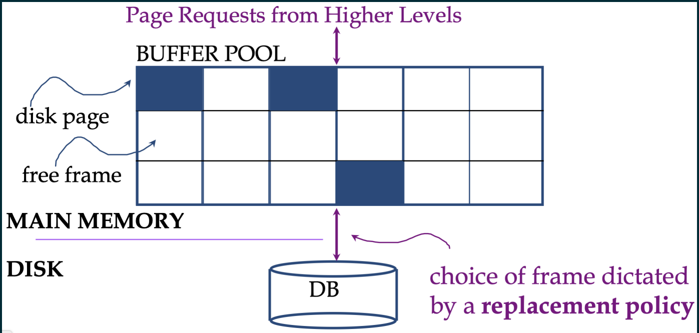
Quick review:

- The DBMS assumes that the primary storage for data is non-volatile disk
- The DBMS is responsible for moving data from primary storage
  - Where it can't be used to satisfy client requests …
  - To main-memory, where it can be used to satisfy client requests

Managing data movement:

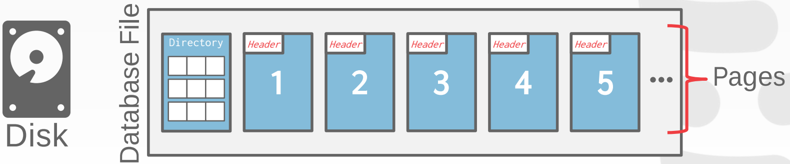
- DBMS "disk space" (or "buffer") managers support the concept of a page as a unit of data
  - The "page" is a main-memory construct
- Page size is usually chosen to be equal to disk block size so that reading or writing a page can be done in one disk I/O operation
- Buffer manager allocates and deallocates pages, reading/writing data from/to disk blocks, as necessary
  - The "buffer" is the portion of main memory available to store copies of disk blocks

Buffer Manager: Goals

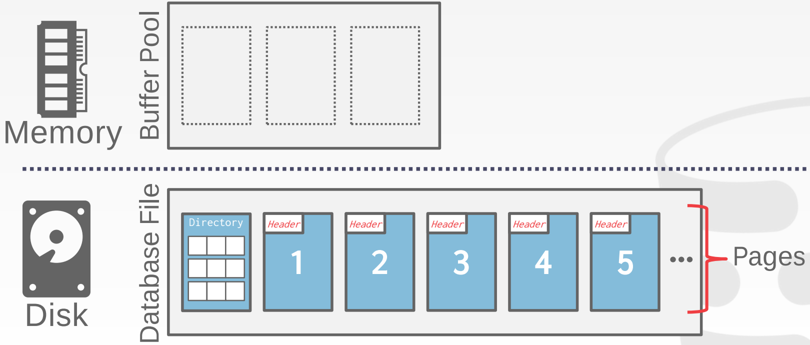


- DBMS buffer manager abstracts hardware and OS details from "higher levels" of the DBMS
  - Hides the fact that most data do not reside in RAM
- Because I/O cost is a major part of overall performance, DBMS wants to minimize the number of block transfers between disk and memory
- Implication: keep as many "blocks in current use" as possible in main memory

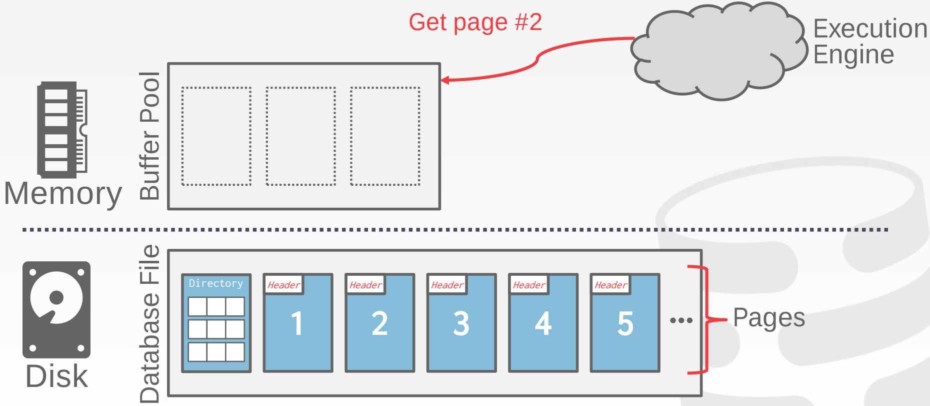
DISK-ORIENTED DBMS



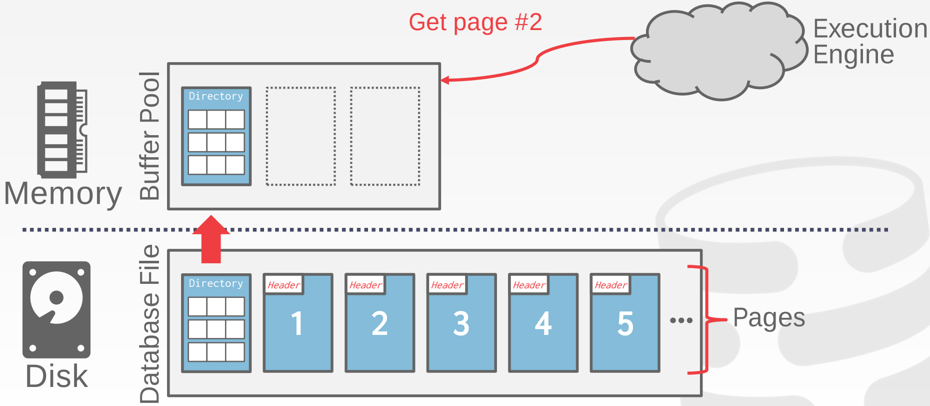
DISK-ORIENTED DBMS



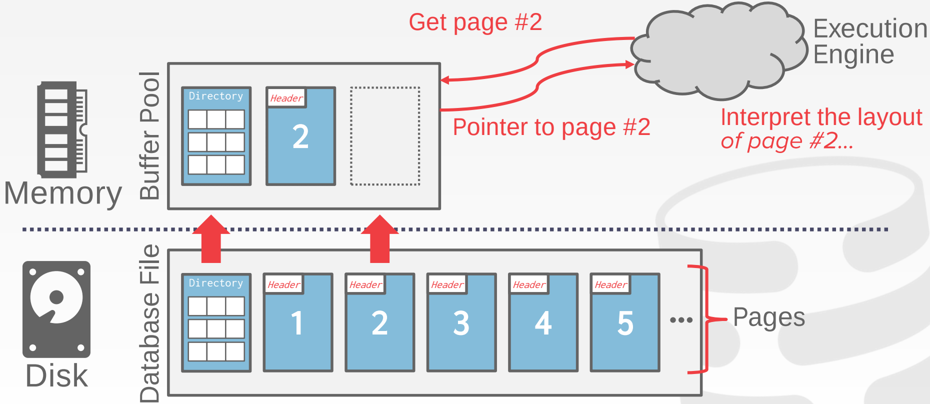
DISK-ORIENTED DBMS



DISK-ORIENTED DBMS



DISK-ORIENTED DBMS



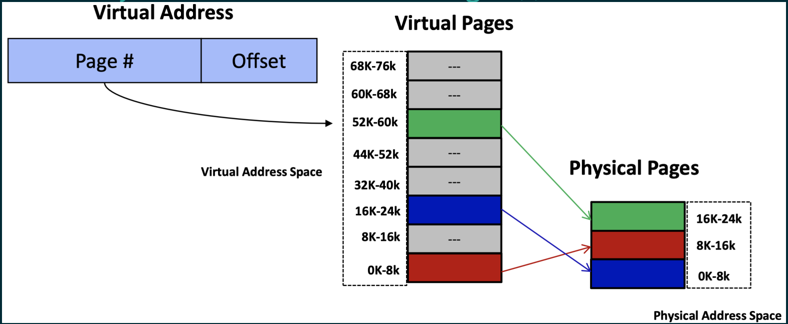
Buffer Manager Responsibilities

- DBMS "higher level" components call the buffer manager when they need a block from disk
- If the block is already in the buffer, buffer manager returns the address of the block in main memory
- If the block is not in the buffer, the buffer manager
  - Allocates space in the buffer for the required block
  - If the buffer is "out of space", will replace ("evict") some other block from main-memory
  - Must write the evicted block's contents back to disk if it's been modified since it was fetched from disk
  - Reads the block from the disk, stores in the buffer, and returns the address of the block in main memory to client
- DBMS maintains a `pin_count` variable: "number of users of a page"
  - Cannot reuse a buffer pool frame until its `pin_count == 0`: i.e., the frame has been unpinned
- DBMS maintains a `dirty` variable: "whether a page has been modified or not"

Buffer Manager: Track Disk Block Allocation

- Buffer manager tracks
  - Which disk blocks are in use?
  - Which pages are on which disk blocks?
- Even though blocks may be initially allocated contiguously . . .
  - A database's contents are dynamic
  - CUD operations will allocate and deallocate blocks and thus create "holes"
- DBMS needs a way to track "free blocks"
  - Can maintain a dynamic list of free blocks
  - Or: maintain a bitmap with one bit per each disk block

Virtual Memory *Versus* DBMS Buffer Manager (I)



- As you know from OS lectures, an OS already employs a buffer management technique known as virtual memory
- This fact raises the obvious question: *"why does a DBMS have its own buffer manager component?"*
  - Wouldn't it make more sense for the DBMS to delegate to the OS?
  - First, we'll explore this issue from the "policy" perspective
  - We explore this issue further at the end of today's lecture (from the "implementation" perspective)

Buffer Page Replacement Policies

Introduction

- Earlier slide: *if the buffer is "out of space", will replace ("evict") some other block from main-memory*
- Replacement policy: "which page gets evicted?"
- Q: didn't Bellady (1966, *"A study of replacement algorithms for virtual storage computers"*) determine the optimal page replacement policy?
  - *"The page that will be accessed the furthest in the future should be the one that is evicted"*
- A: unfortunately, we can't implement this strategy ☹
  - I hope you can see why ☺
- Implication: DBMS must use a "sub-optimal" replacement policy
- Algorithm should:
  - Be accurate (despite lack of "oracle knowledge")
  - Be fast!
  - Require only little meta-data

Some Classic Replacement Policies

- First-in-first-out (FIFO)
  - Easy to implement, performs poorly in practice ☹
- Least recently used (or LRU)
  - Use past pattern of block references as a predictor of future pattern
- LRU implementation
  - Maintain a timestamp of when each page was last accessed
  - When DBMS needs to evict a page, select the one with the oldest timestamp
  - Refinement: keep the pages in sorted order to reduce the search time when doing an "eviction"
- LRU is almost optimal but expensive to implement
  - Space issue: needs a separate timestamp per-page ☹
- The "clock algorithm" is a good approximation of LRU that addresses the "space" problem

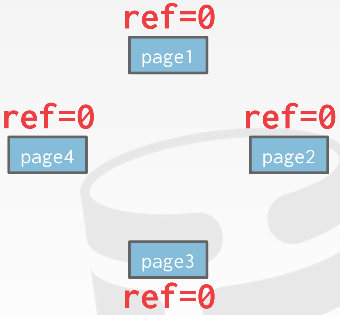
“Clock” Replacement Policy

- Variant of FIFO: maintains a circular list of pages
- But: replace the page currently pointed to by the iterator only if the referenced bit is “clear”
- Otherwise: “clear” the referenced bit, but advance the iterator
  - That is: don’t evict that page (because its bit “was (previously) set”)
- Intuition: even if a page is an “old page” (FIFO policy) …
  - If the page has been referenced, it’s probably in use
  - Let’s not evict that page if we can find another page (even a “more recent” page) that has not been referenced
- “Clock” is really a probabilistic approximation of LRU
  - Most recently used page will never be evicted

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.

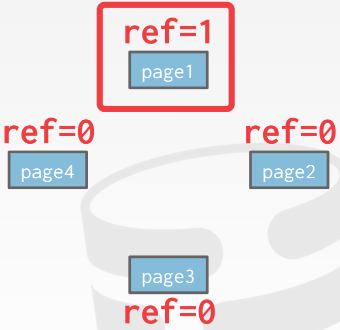
Organize the pages in a circular buffer with a "clock hand":
→ Upon sweeping, check if a page's bit is set to 1.
→ If yes, set to zero. If no, then evict.



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.

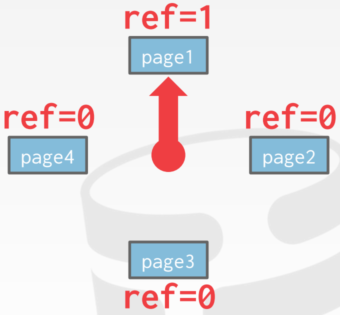
Organize the pages in a circular buffer with a "clock hand":
→ Upon sweeping, check if a page's bit is set to 1.
→ If yes, set to zero. If no, then evict.



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.

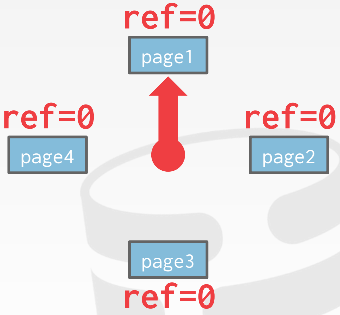
→ Each page has a reference bit.

→ 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.

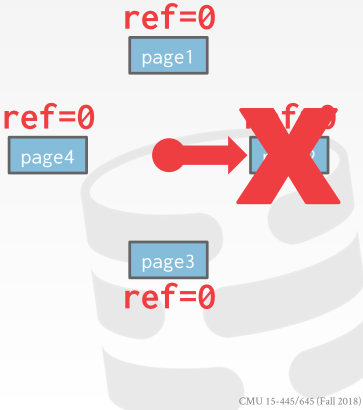
→ If yes, set to zero. If no, then evict.



CLOCK

Approximation of LRU without needing a separate timestamp per page.
→ Each page has a 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.

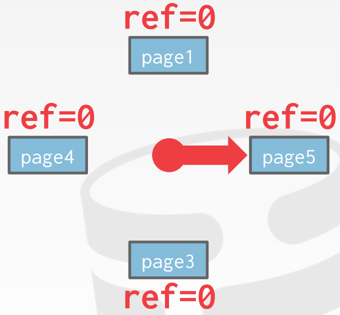
→ Each page has a <u>reference bit</u>.

→ 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.

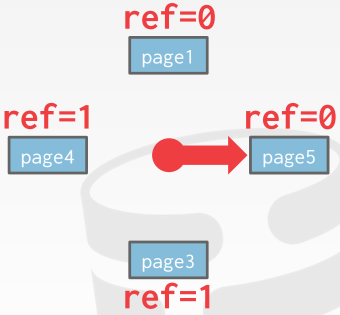
→ If yes, set to zero. If no, then evict.



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.

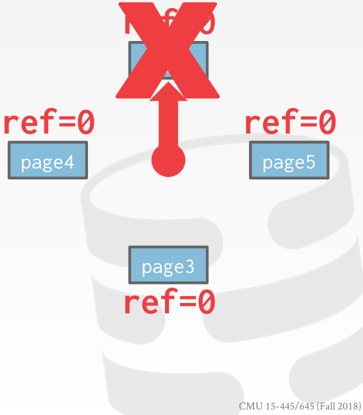
→ Each page has a reference bit.

→ 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.



DBMS: Why Not Use LRU?

- Most OS use LRU as their replacement policy
- As we'll see, LRU is not the replacement policy of choice for a DBMS
- Q: why do DBMS insist on doing their own thing ☺
- A: because LRU is only the best approach in the absence of "other" predictive information
- Database queries typically involve multiple steps, such as:
  1. JOIN this table with that table
  2. SELECT specified attributes from the JOIN result
- Key point: as the entity determining each of these steps, the DBMS is well-placed to predict what blocks are going to be needed in the (short-term) future
- Given this extra information, it often makes sense for a DBMS to use a different replacement policy than LRU
- Next slides will illustrate this important point

“Toss-Immediate” Policy (I)

```sql
select * from instructor natural join department;
```

- Consider this vanilla SQL query which involves a JOIN
- We haven’t yet discussed the important topic of query processing and optimization ...
- But: the pseudo-code below is a good start

```
for each tuple i of instructor do
    for each tuple d of department do
        if i[dept_name] = d[dept_name]
        then begin
                let x be a tuple defined as follows:
                x[ID] := i[ID]
                x[dept_name] := i[dept_name]
                x[name] := i[name]
                x[salary] := i[salary]
                x[building] := d[building]
                x[budget] := d[budget]
                include tuple x as part of result of instructor ⋈ department
            end
    end
end
```

“Toss-Immediate” Policy (II)

```
for each tuple_r of r do
    for each tuple_s of s do
        if  tuple_r matches tuple_s
            join the tuples into a  new tuple
            include the new tuple in the result set
        fi
    done
done
```

- Once a block of “r” tuples has been processed (*instructor* in our example), the DBMS knows that they will not be used again!
- LRU says “keep that block in memory”
- “Toss-immediate” policy: DBMS should release that block once the final tuple in the “r” block has been processed

Sequential Flooding Phenomenon (I)

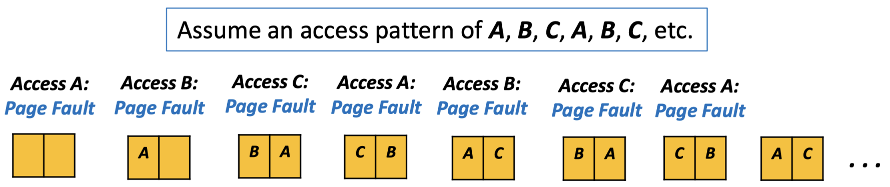
```
for each tuple_r of r do
    for each tuple_s of s do
        if  tuple_r matches tuple_s
            join the tuples into a  new tuple
            include the new tuple in the result set
        fi
    done
done
```

- Now consider the inner loop of this typical JOIN processing …
- In effect, the DBMS is doing a sequential scan of the "s" tuples (*department* in our example)
- Once a block of "s" tuples has been processed, it will only be accessed again after all other "s" tuple blocks have been processed
  - Note: the outer "r" block will be reused, but only after the "inner loop" finishes processing all "s" tuple blocks

Sequential Flooding Phenomenon (II)

- An OS doesn't have access to the "query-specific" information that tells the DBMS that the query involves an inner loop of block processing
  - Due to the fundamental structure of a JOIN
- Using LRU, the OS will keep the most recently used department block pinned and release the "least recently used" block
- In contrast, the DBMS should use the most recently used (or MRU) replacement policy!
  - If a department block must be removed from the buffer, the MRU policy pins the department block currently being processed
  - After the final department tuple in that block has been processed, the block is unpinned, because it's now the "most recently used block"

Sequential Flooding Phenomenon (III)

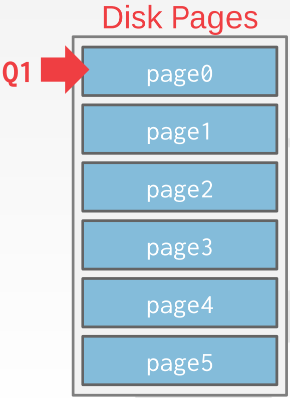


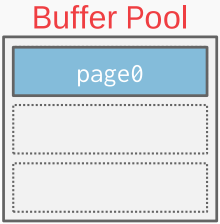
Summary:

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

SEQUENTIAL FLOODING

Q1 `SELECT * FROM A WHERE id = 1`

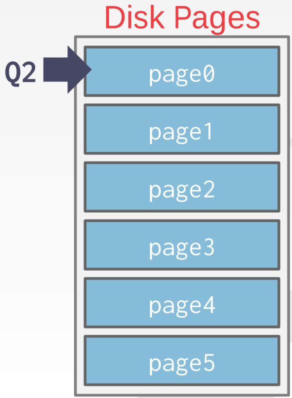


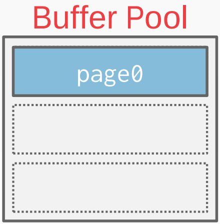


SEQUENTIAL FLOODING

Q1 `SELECT * FROM A WHERE id = 1`

Q2 `SELECT AVG(val) FROM A`

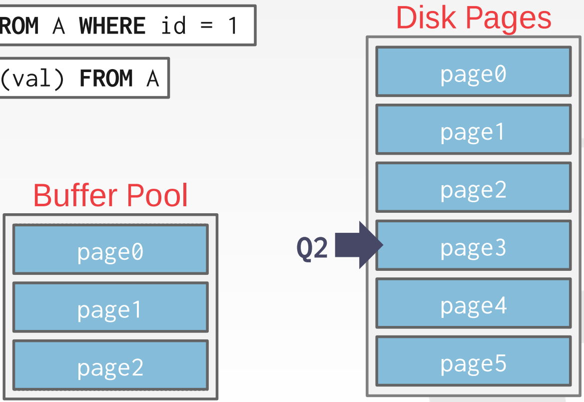




SEQUENTIAL FLOODING

Q1 `SELECT * FROM A WHERE id = 1`

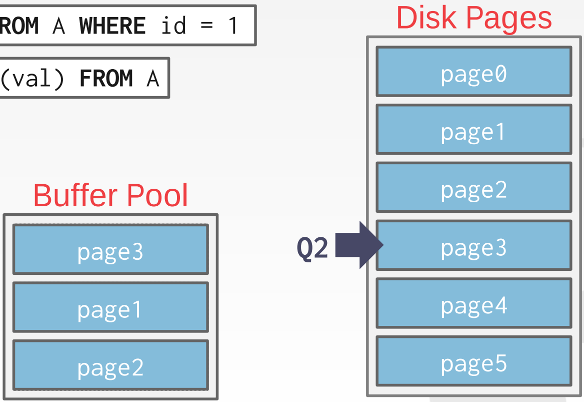
Q2 `SELECT AVG(val) FROM A`



SEQUENTIAL FLOODING

Q1 `SELECT * FROM A WHERE id = 1`

Q2 `SELECT AVG(val) FROM A`

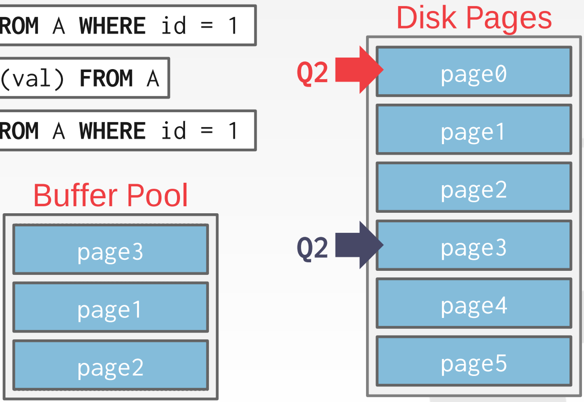


SEQUENTIAL FLOODING

Q1 `SELECT * FROM A WHERE id = 1`

Q2 `SELECT AVG(val) FROM A`

Q3 `SELECT * FROM A WHERE id = 1`

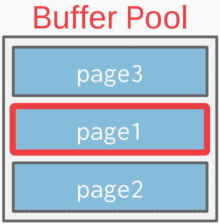


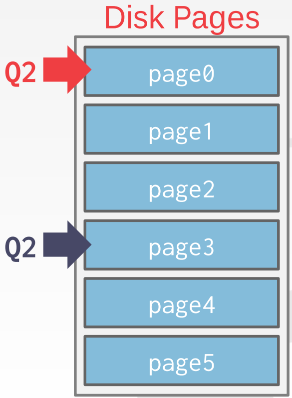
SEQUENTIAL FLOODING

Q1 `SELECT * FROM A WHERE id = 1`

Q2 `SELECT AVG(val) FROM A`

Q3 `SELECT * FROM A WHERE id = 1`





Other Reasons For DBMS-Specific Replacement Policies (I)

- Previous slides illustrated scenarios in which the DBMS can do "short-term" prediction of access patterns using its knowledge of query structure
- In addition: the DBMS can use do "long-term" prediction of access patterns using its collected statistical information
  - Example: *"what is the probability that a request will reference a particular relation?"*
  - Example: *"how important is $block_i$ compared to $block_j$?"*
- Such information is not available to an OS, but the DBMS can use it to make better quality replacement decisions
- Examples
  - Data dictionary (meta-data) is frequently accessed
  - Indices are frequently accessed (following slides calls this "priority hints")
  - Heuristic: be biased against swapping out these blocks

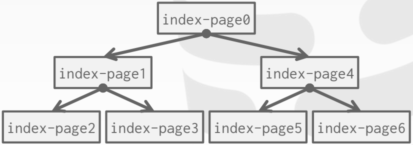
BETTER POLICIES: PRIORITY HINTS

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++)
```

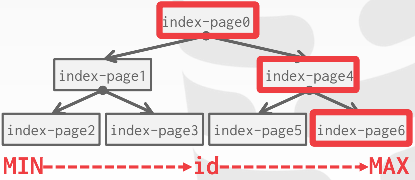


BETTER POLICIES: PRIORITY HINTS

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++)`



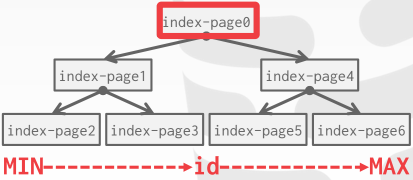
BETTER POLICIES: PRIORITY HINTS

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 = ?`



Other Reasons For DBMS-Specific Replacement Policies (II)

- The concurrency control module can tell the buffer manager *'I'm delaying processing these requests to satisfy ACID properties"*
- Implication: the buffer manager should keep blocks associated with "active requests" pinned
- Similarly: the crash recovery module can tell the buffer module not to "write $block_i$ to disk" just yet
  - Example: other blocks must first be "forced to disk" before it's safe to write $block_i$ to disk

More DBMS-Specific Ideas: “Localization”

- Instead of treating block requests “globally”, “localize” buffer management by making eviction decisions on a per query or per transaction basis
- Motivation: reduces the “pollution” (or “dilution”) of the buffer pool made by the global access pattern
- Localization keeps track of the pages that an individual query has accessed
  - Example: POSTGRESQL maintains a small ring buffer that is private to the query.

Before We Leave This Topic: Incorporate "Eviction Cost"

- We've been tacitly assuming that, when making "eviction" decisions, "costs are equal"
  - Meaning: the only issue that the buffer manager has to consider is "which page is least likely to be useful in the future?"
- But: consider the cost of evicting a page …
  - Cheap to evict a page that is not dirty: DBMS can "drop it", no need to interact with disk at all
  - Expensive to evict a page that is dirty: DBMS must write changes back to disk
- Implication: may be worth prioritizing the eviction of "non-dirty" pages, and use a background process to write dirty pages back to disk

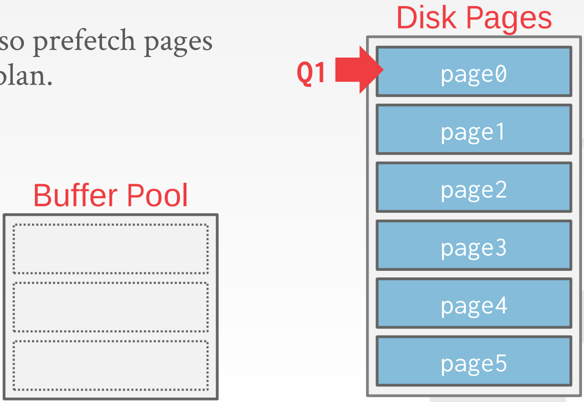
Pre-Fetching

Introduction

- You're aware (if only from *Introduction to Algorithms* lecture) that the OS can pre-fetch pages from disk into main-memory
  - Meaning: when the client actually requests the data, it's already in main-memory ☺
- Key idea: exploit "fact" of spatial locality
  - Note: exploiting the idea of temporal locality more closely resembles the idea of using "extra information" to make better eviction decisions
- Next set of slides further illustrate the theme we've been discussing: buffer manager can do a better job than the OS because the DBMS has additional "higher-level" information
  - Specifically: can do a better job at pre-fetching blocks from disk

PRE-FETCHING

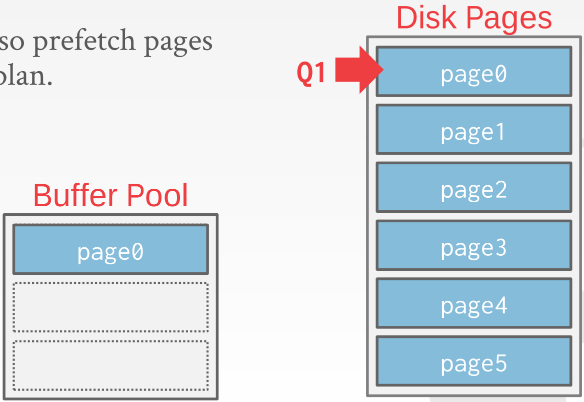
The DBMS can also prefetch pages based on a query plan.
→ Sequential Scans
→ Index Scans



PRE-FETCHING

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

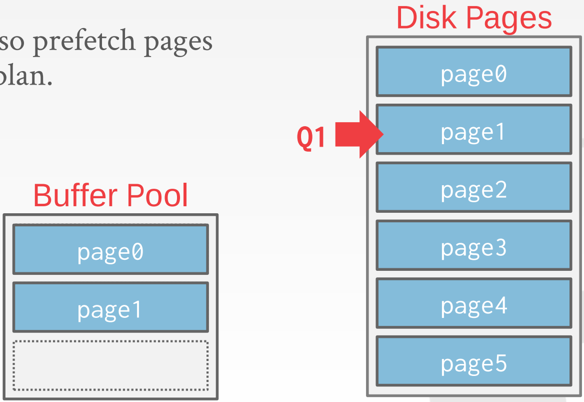
→ Sequential Scans
→ Index Scans



PRE-FETCHING

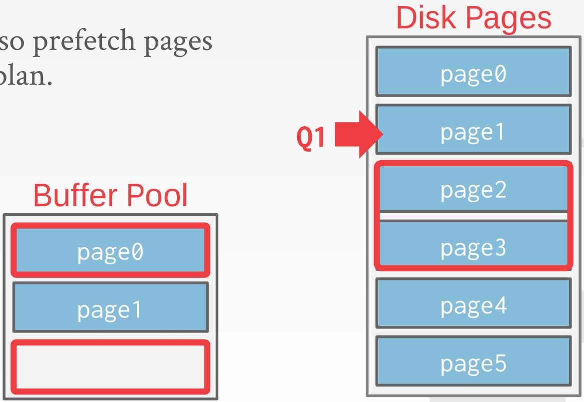
The DBMS can also prefetch pages based on a query plan.

→ Sequential Scans
→ Index Scans



PRE-FETCHING

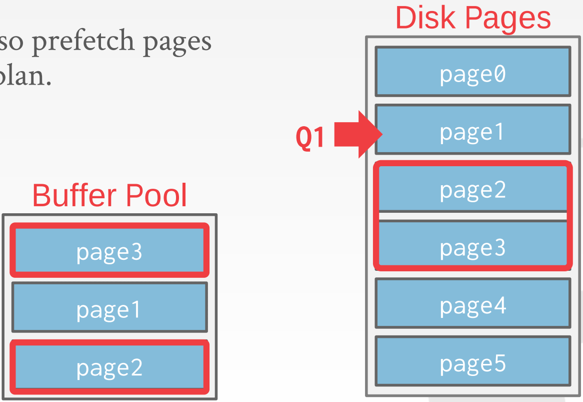
The DBMS can also prefetch pages based on a query plan.
→ Sequential Scans
→ Index Scans



PRE-FETCHING

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

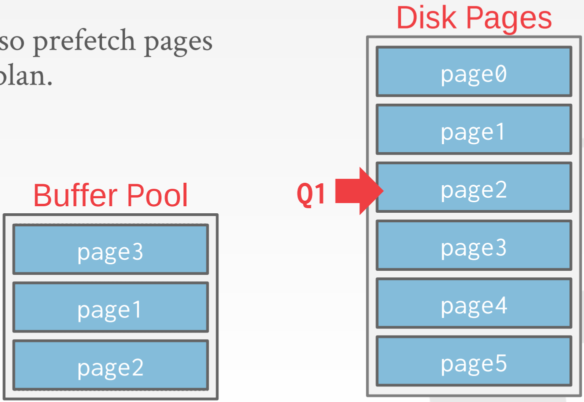
→ Sequential Scans
→ Index Scans



PRE-FETCHING

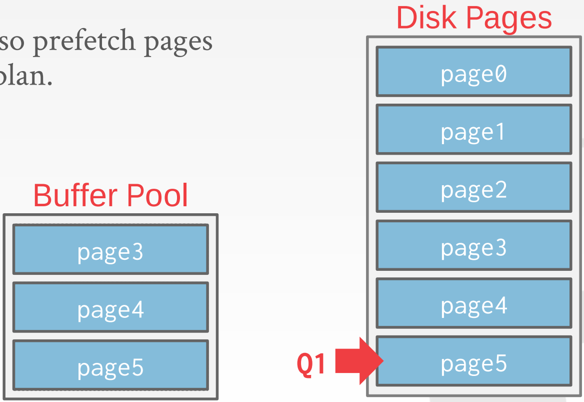
The DBMS can also prefetch pages based on a query plan.

→ Sequential Scans
→ Index Scans

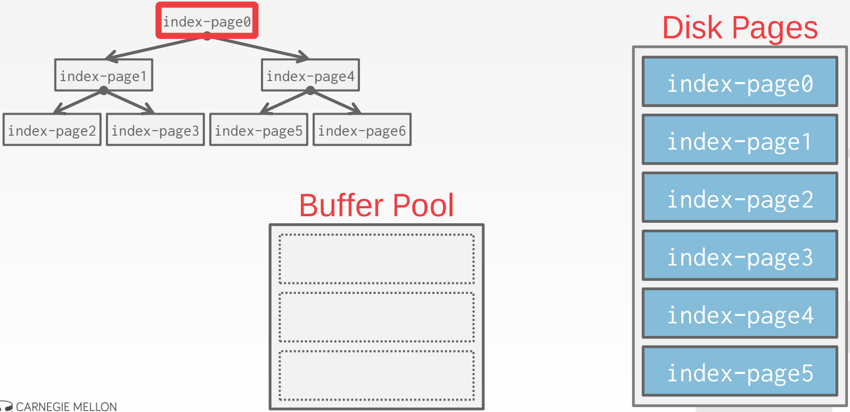


PRE-FETCHING

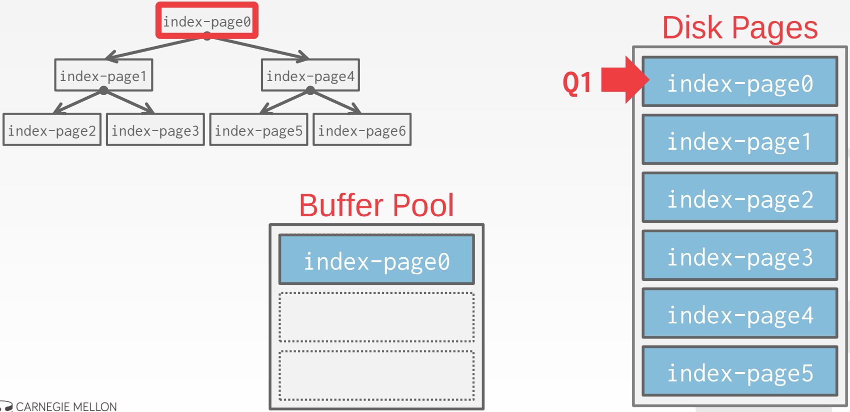
The DBMS can also prefetch pages based on a query plan.
→ Sequential Scans
→ Index Scans



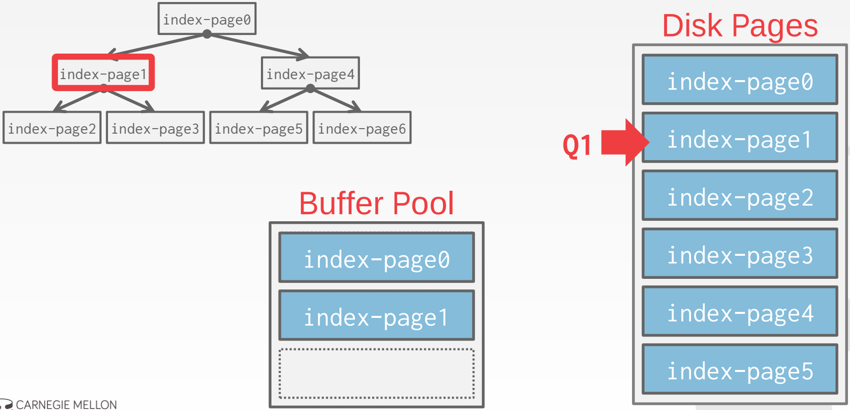
PRE-FETCHING



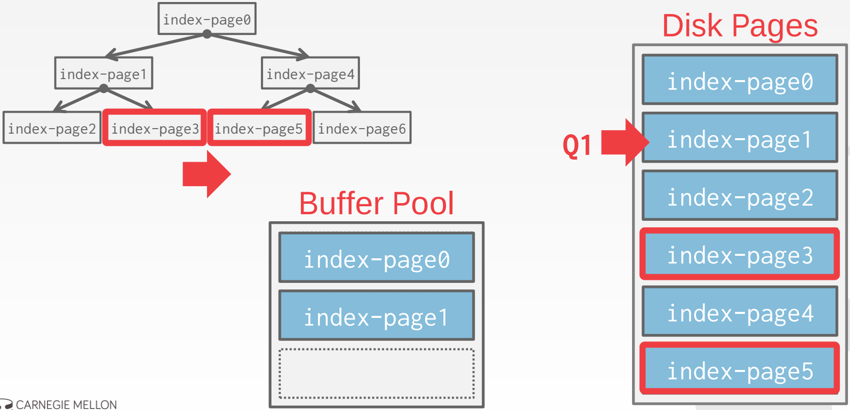
PRE-FETCHING



PRE-FETCHING



PRE-FETCHING



Bonus: “Why Not Use The OS?”

Be Aware of Low-Level OS APIs

- Even if a DBMS doesn't want to cede control to "high-level" OS virtual memory ...
- You should be aware of OS "lower-level" APIs such as `mmap`
- Faster than invoking `read` and `write` OS "system calls"
  - `mmap` is essentially "writing to a pointer" (faster, and OS can translate in the background)
  - No explicit crossing of "protection domains"
  - Program can access the data directly in the mapped region, doesn't have to perform a memory copy
  - See e.g., this thread

> Which raises the question in even more pointed fashion: why invent a "DBMS-specific" buffer manager?

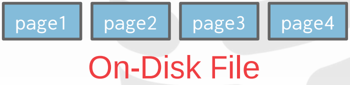
It's All About "Control ⇒ Performance" (I)

- In a nutshell: the DBMS wants to control memory movement because it claims that it can do a better job at it than the OS ☺
- Definitely can't beat the OS for "general purpose" applications
- But: DBMS is a very large application which places a high priority on performance
- As a single "focused" application, the DBMS has access to information (or specialized algorithms) that the OS doesn't know about
- Examples
  - Flushing dirty pages to disk in the correct order
    - Remember: this is needed for WAL algorithm
  - Specialized prefetching because DBMS can predict page reference patterns from query history (today's lecture)
  - Same idea applies for general buffer replacement policy (today's lecture)

WHY NOT USE THE OS?

One can use **mmap** to map the contents of a file into a process' address space.

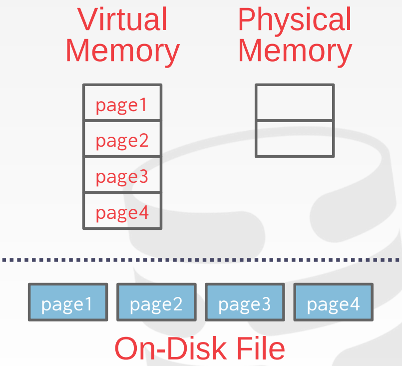
The OS is responsible for moving data for moving the files' pages in and out of memory.



WHY NOT USE THE OS?

One can use **mmap** to map the contents of a file into a process' address space.

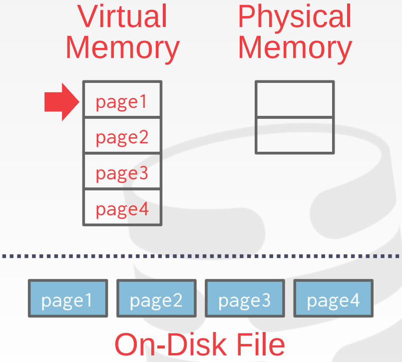
The OS is responsible for moving data for moving the files' pages in and out of memory.



WHY NOT USE THE OS?

One can use **mmap** to map the contents of a file into a process' address space.

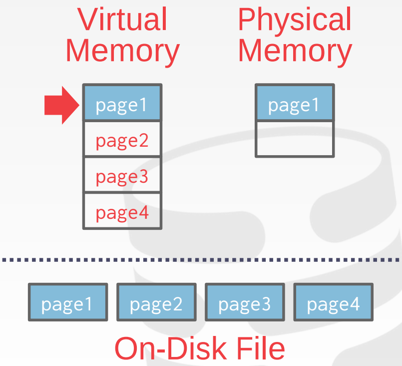
The OS is responsible for moving data for moving the files' pages in and out of memory.



WHY NOT USE THE OS?

One can use **mmap** to map the contents of a file into a process' address space.

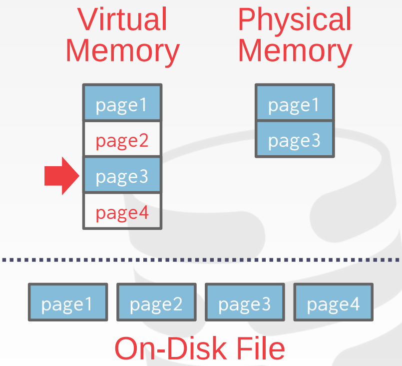
The OS is responsible for moving data for moving the files' pages in and out of memory.



WHY NOT USE THE OS?

One can use **mmap** to map the contents of a file into a process' address space.

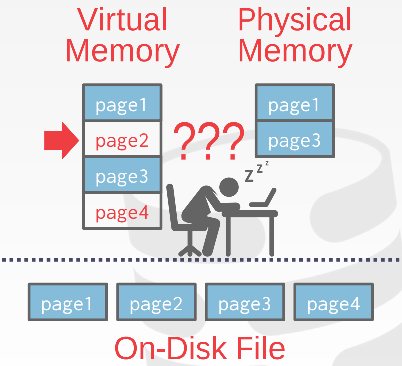
The OS is responsible for moving data for moving the files' pages in and out of memory.



WHY NOT USE THE OS?

One can use **mmap** to map the contents of a file into a process' address space.

The OS is responsible for moving data for moving the files' pages in and out of memory.



It's All About "Control ⇒ Performance" (II)

- DBMS buffer managers optimize access to disk: e.g., by "writing" in-memory version of data, and postponing "write to disk" ...
  - We've previously discussed some of these ideas in DBMS recovery lectures
  - Briefly reviewing these ideas to further justify why DBMS buffer manager doesn't simply delegate to the OS
- OS ("vanilla file manager") cares less about performance than DBMS
- DBMS may therefore write to non-volatile buffers to allow (eventual) "catch-up" of disk writes
  - Allows DBMS to reorder writes to minimize disk "arm movement"
- Or: DBMS may use a log disk: a disk devoted to writing a sequential log of block updates
  - Very fast: no need to do a "seek"
- But keep in mind that general-purpose journaling file systems perform similar (non-DBMS-specific) function

Disk-Space Management (Buffer Manager)

Buffer Page Replacement Policies

Pre-Fetching

Bonus: "Why Not Use The OS?"

Readings

- The textbook places much less importance on today's material than I'd like ☹
- Read Chapter 13.5