CS-300: Data-Intensive Systems

The Storage Layer

(Chapter 12.1-12.5 13.5)

Prof. Anastasia Ailamaki, Prof. Sanidhya Kashyap

EPFL

Today’s focus

- DBMS layers and storage hierarchy

- Disks
  - HDD
  - SSD

- Buffer management

The storage hierarchy

Volatile

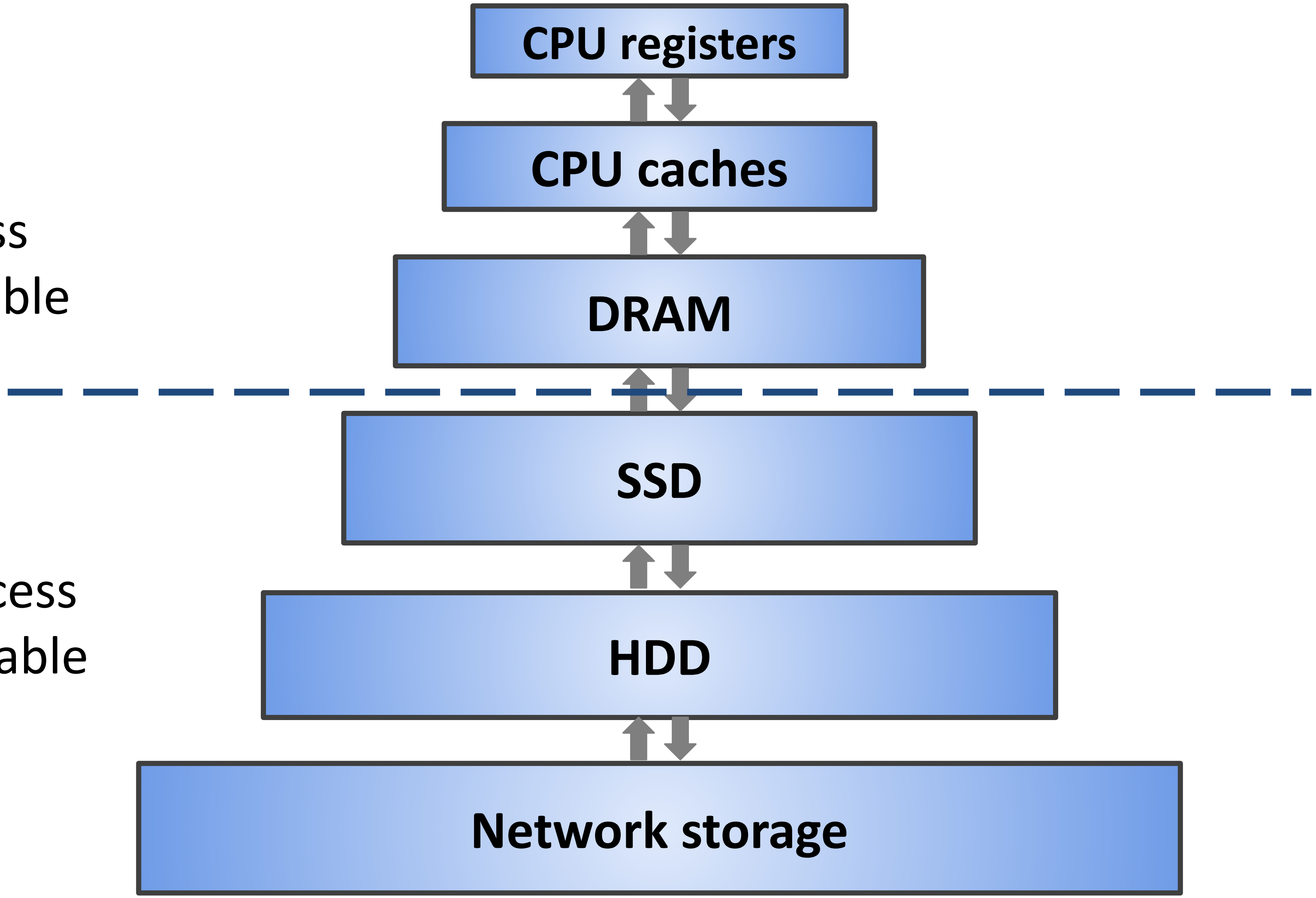
Random access

Byte-addressable

Non-volatile

Sequential access

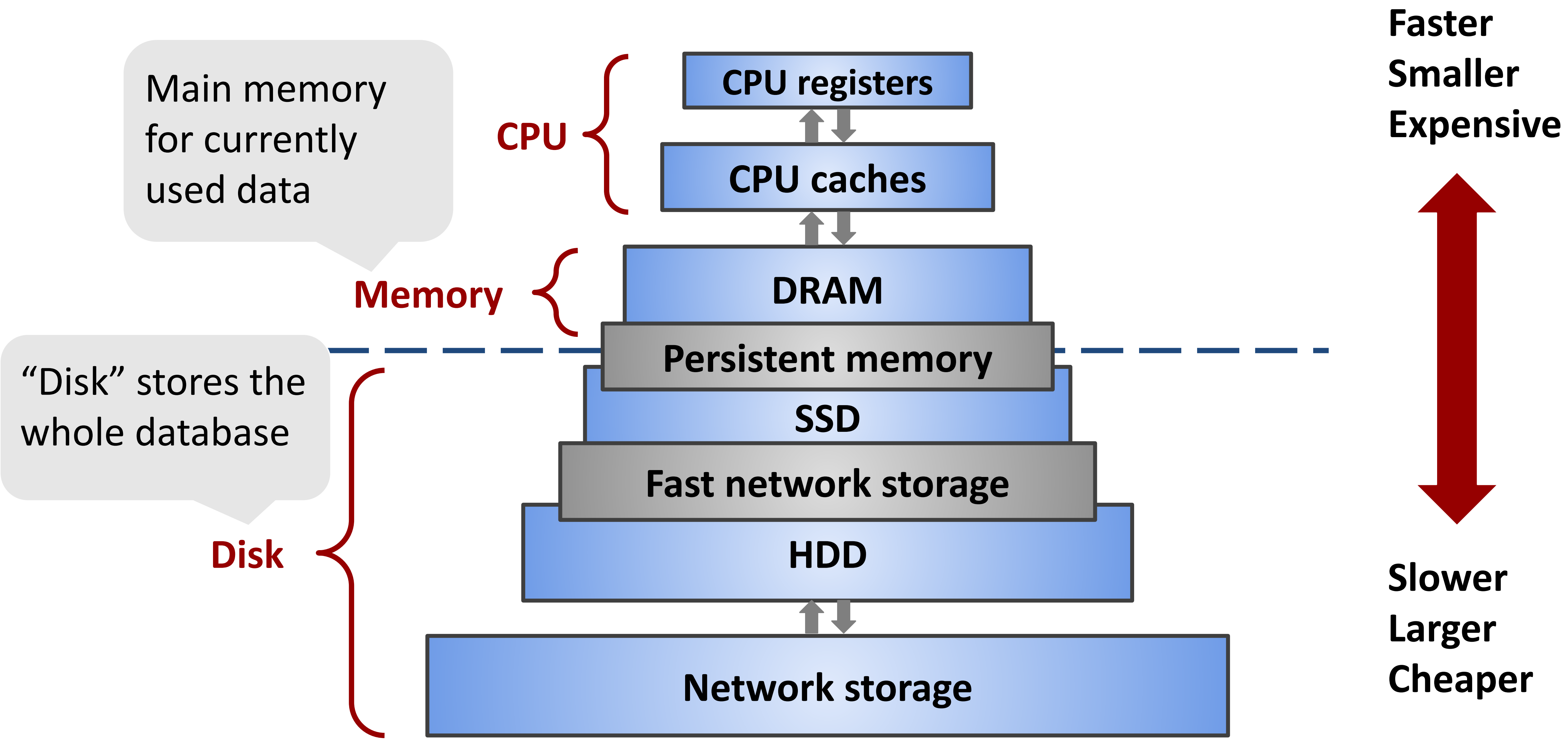
Block-addressable



Faster
Smaller
Expensive

Slower
Larger
Cheaper

The storage hierarchy



Latency numbers every programmer should know

1 ns	L1 cache reference	⇐	**1 sec**
4 ns	L2 cache reference	⇐	**4 sec**
100 ns	DRAM	⇐	**100 sec**
16,000 ns	SSD	⇐	**4.4 hours**
2,000,000 ns	HDD	⇐	**3.3. weeks**
~50,000,000 ns	Network storage	⇐	**1.5 years**
1,000,000,000 ns	Tape archives	⇐	**31.7 years**

https://colin-scott.github.io/personal_website/research/interactive_latency.html

Disks and Files

- DBMS stores information on disks
  - In an electronic world, disks are a mechanical anachronism!

- This has major implications for DBMS design:
  - **READ:** transfer data from disk to main memory (RAM)
  - **WRITE:** transfer data from RAM to disk
  - Both are high-cost operations, relative to in-memory operations, so must be planned carefully!

Why not store data all in the main memory?

- ***Costs too high***
  - High-end databases today in the Petabyte range
  - ~ 60% of the cost of a production system is in the disks

- ***Main memory is volatile***
  - We want data to be saved between runs (Obviously!)

- ***Main-memory database systems:***
  - The emerging hardware enables main-memory database systems development
  - Smaller size, performance optimized
  - Volatility is ok for some applications

Disks become a viable option to store data

Today’s focus

- DBMS layers and storage hierarchy

- Disks
    - HDD
    - SSD

- Buffer management

Disks

- Secondary storage device of choice
- Data is stored and retrieved in units called ***disk blocks*** or ***pages***
- Unlike RAM, time to retrieve a disk page varies:
  - Depends on the type of disk (magnetic vs flash vs PCM)
  - Depends on the access pattern (sequential vs random)

- Relative placement of pages on disk has major impact on DBMS performance!

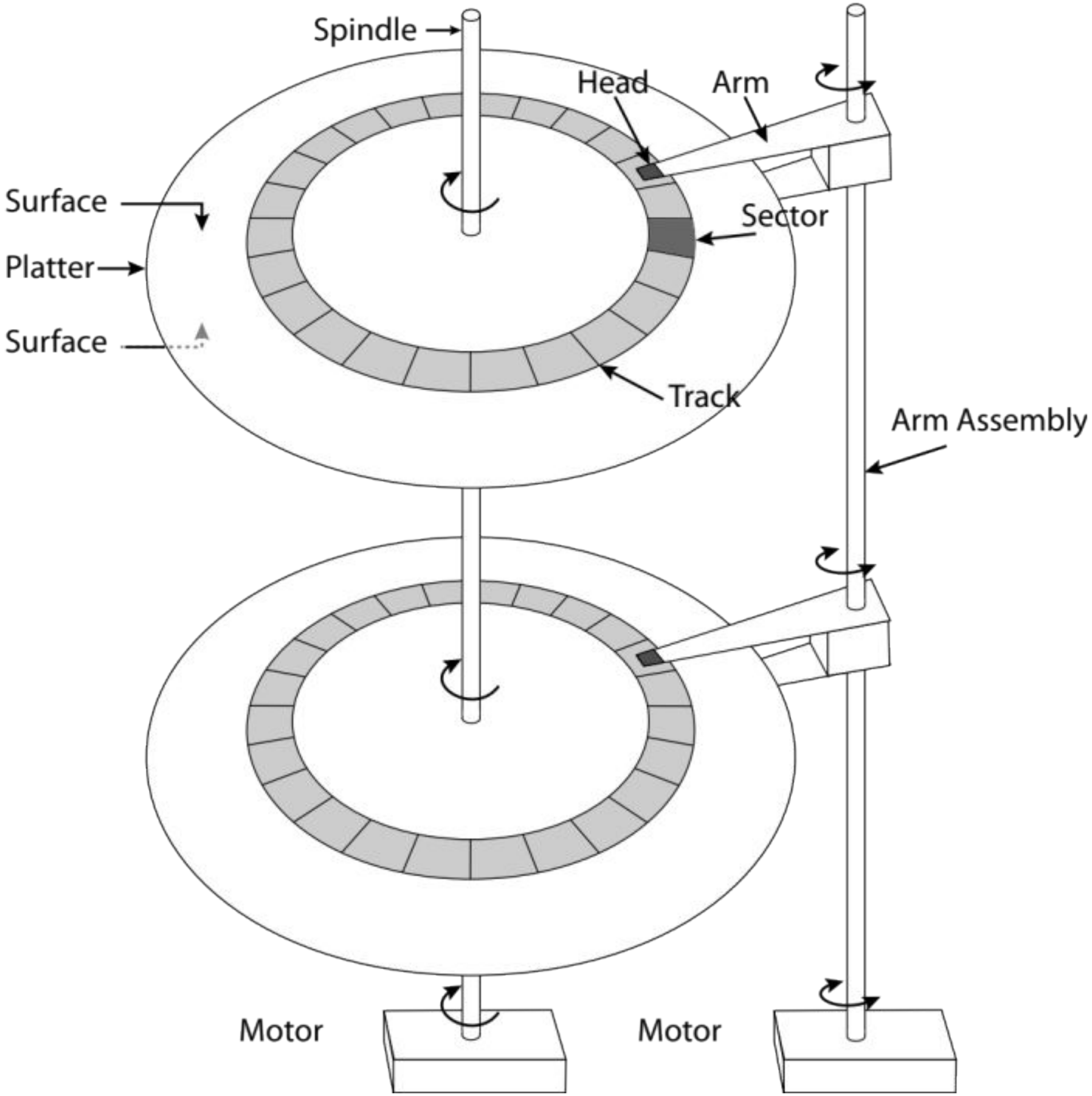
Storage devices

- **Magnetic disks** (market value: 20B+ CHF)
  - Large capacity at low cost
  - Block level random access
  - Slow performance for random access
  - Good performance for streaming access
- **Flash memory** (market value: 32B+ CHF)
  - Capacity at intermediate cost
  - Block level random access
  - Medium performance for random writes
  - Good performance otherwise

Magnetic disks are 60 years old!

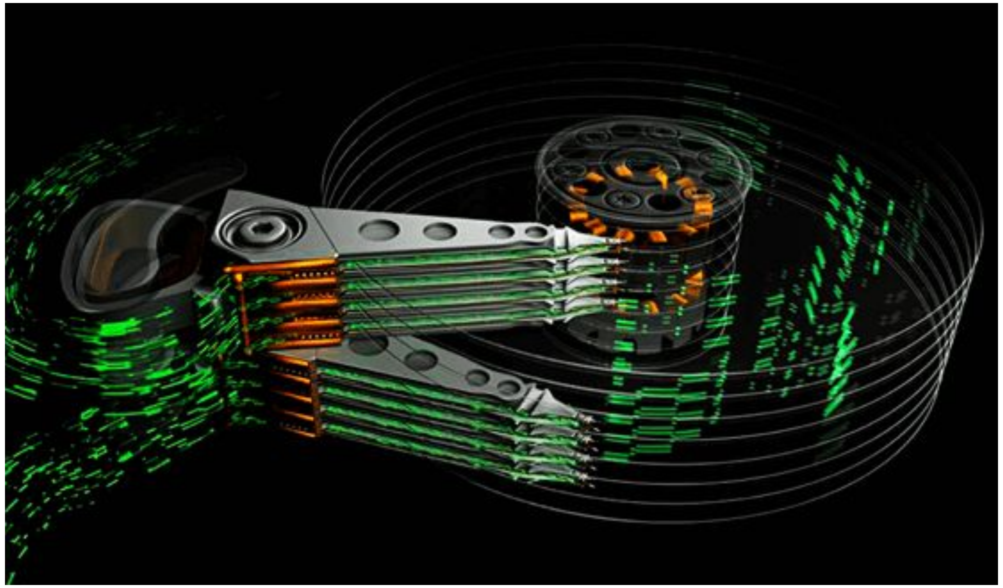
Magnetic disk

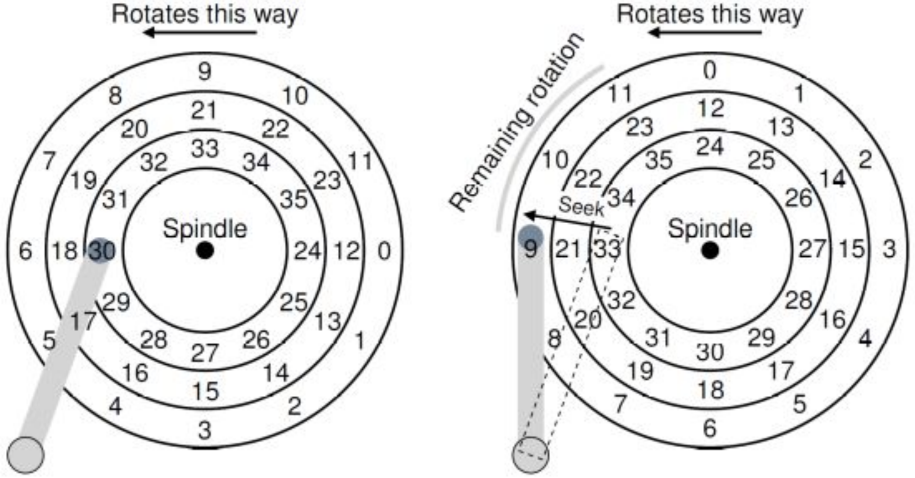
- Stores data magnetically on thin metallic film bonded to rotating disk of glass, ceramic, or aluminium
- **Track**: concentric circles on the disk
- **Sectors:** slice of a track; smallest addressable unit
- **Cylinder:** All the tracks under the head at a given point on all surfaces



Magnetic disk

- Length of tracks vary across disk: outer tracks can have more sectors than inner tracks and higher bandwidth
- Disk is organized into regions (“zones”) of tracks with the same number of sectors per track
- Disk has a sector-addressable address space (sectors: 512 or 4096 bytes)
  - *Block size* is a multiple of *sector size* (fixed)
- Main operations: read/write





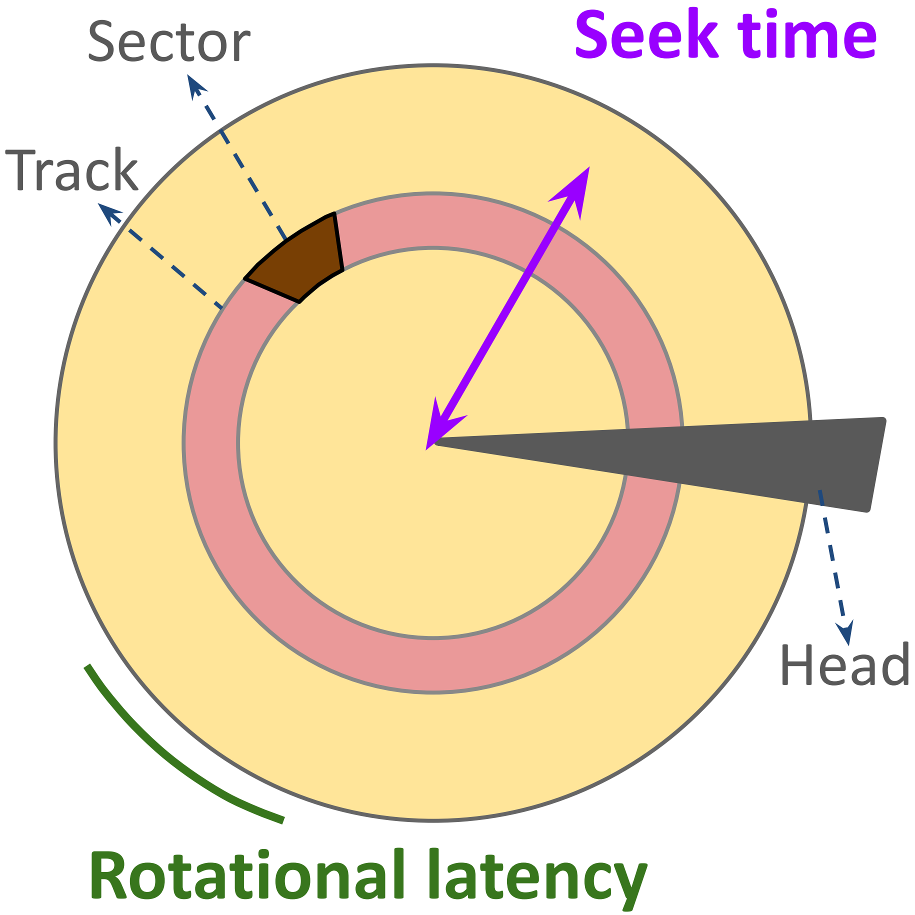
Accessing a disk page

- Time to access (read/write) a disk block:
  - *Seek time* (moving arms to position disk head on track)
  - *Rotational delay* (waiting for block to rotate under head)
  - *Transfer time* (actually moving data to/from disk surface)

Disk overhead

Disk latency = Seek time + Rotation time + transfer time

- **Seek:** position the head/arm over the proper track
  - To get to the track (5–15 ms)
- **Rotational delay:** Wait for desired sector to rotate under read/write head (4–8 ms)
  - Only need to wait for half a rotation on average
- **Transfer time:** Transfer a block of bits (sectors) under read/write head (25–50 usec)
- Sequential access >10x faster than random access



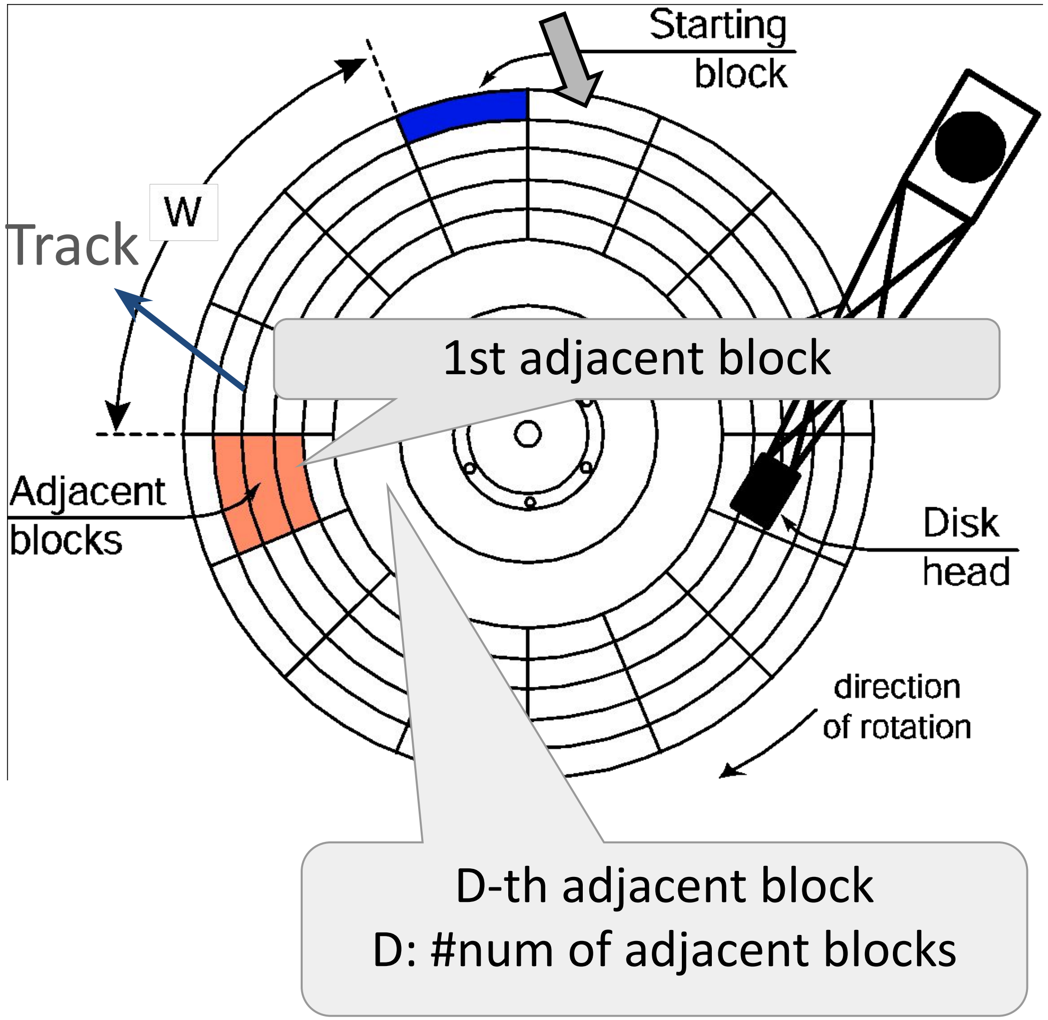
Arranging pages on the disk

- *"Next"* block concept:
  - Blocks on same track, followed by
  - Blocks on same cylinder, followed by
  - Blocks on adjacent cylinder

- Blocks in a file should be arranged sequentially on disk (by "next"), to minimize seek and rotational delay.

- An important optimization: pre-fetching

- Adjacent blocks are equidistant wrt access time from starting block



Disk block has more than one neighbor

Disk performance example

Using disk efficiently: Minimize seek time and rotational delay

Q. How do access pattern influence reads/writes to disk?

- Average seek time: 5ms

- 7200 RPM ⇒ Time for rotation: 60,000 (ms/min)/7200(rev/min) ≅ 8ms

- Transfer rate of 50 MBps, with block size of 4096 bytes

  ⇒ 4096 bytes / (50×10^6) = 81.92×10^{-6} sec ≅ 0.82 ms for 1 sector

Disk performance example

- Read block from **random place on a disk** (random reads):
  - Seek (5ms) + Rot. delay (4ms) + Transfer (0.082ms) = 9.082 ms
  - Approx. 9ms to fetch/put data: 4096 bytes / 9.082×10^{-3} s ≅ 451 KB/s
- Read block from a **random place in the same cylinder on a disk**:
  - Rot. delay (4ms) + Transfer (.082ms) = 4.082 ms
  - Approx 4ms to fetch/put data: 4096 bytes / 4.082×10^{-3} s ≅ 1.03 MB/s
- Read next block **on the same track** (sequential reads):
  - Transfer (0.082ms): 4096 bytes / 9.082×10^{-3} s ≅ 50 MB/s

Enter Flash

- Flash chips used for >20 years
- Flash evolved

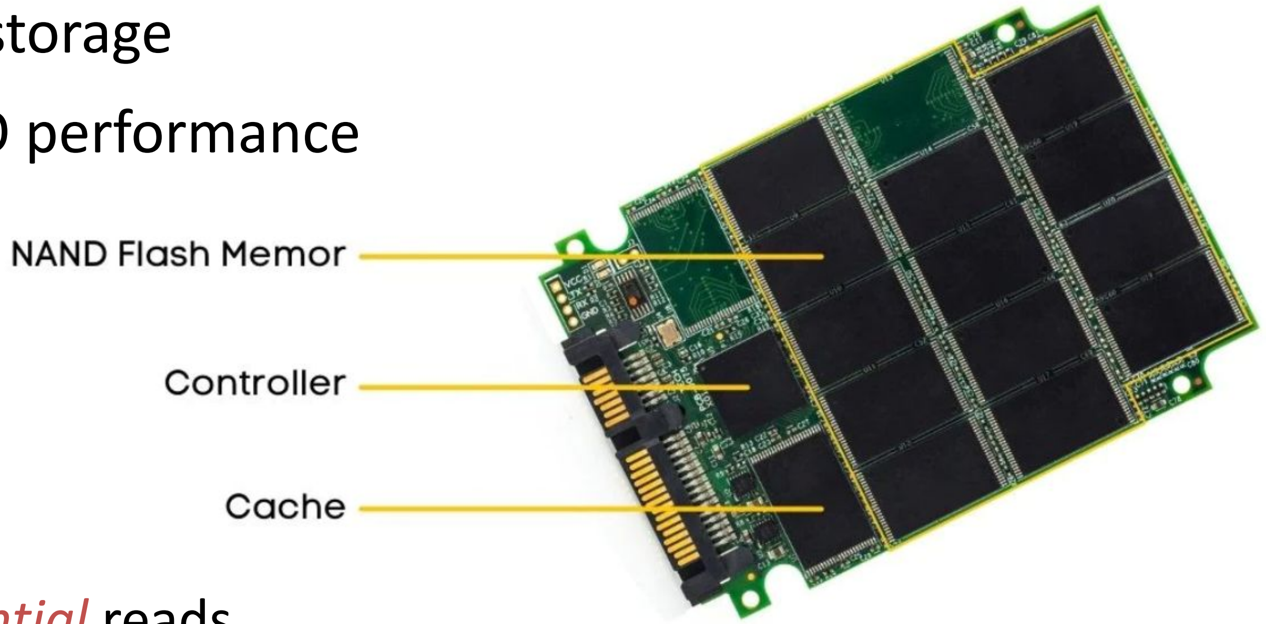
  - USB keys
  - Storage in mobile devices
  - Consumer and enterprise flash disks (SSD)

- Flash in a DBMS
  - Main storage
  - Accelerator/enabler (Specialized cache, logging device)

Flash disks

- One of today's go-to solutions for fast storage
- Provides orders of magnitude higher IO performance than HDDs

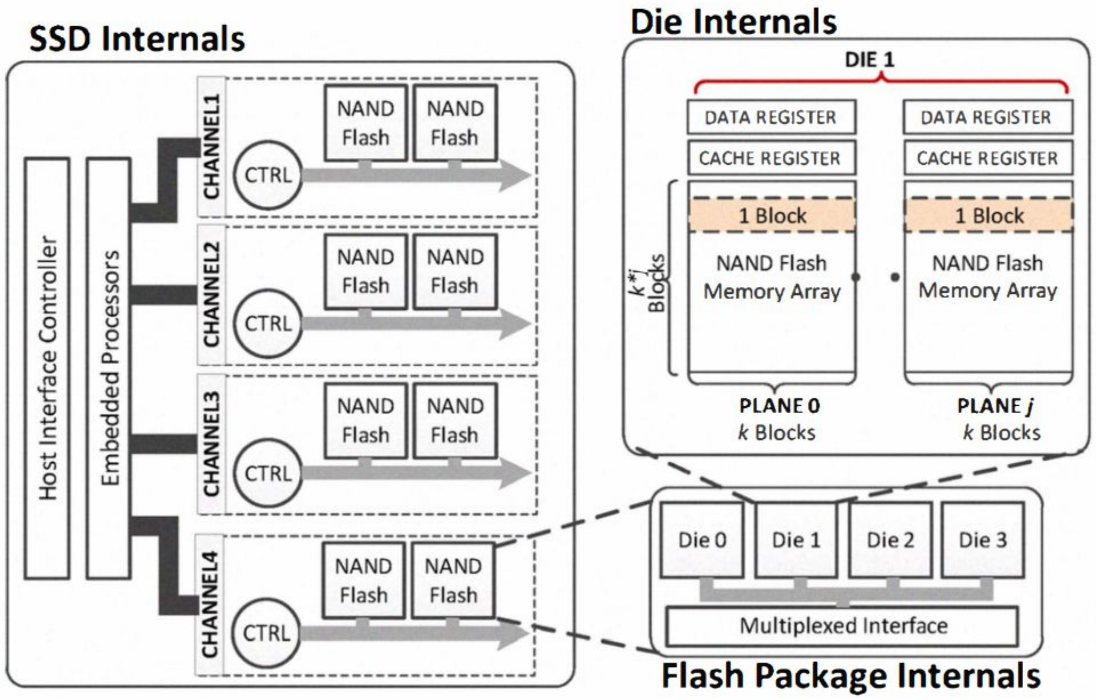
- For DBMS:
  - Secondary storage *or* caching layer
  - <u>*Random reads*</u> "equally fast" as *sequential* reads
  - Data organized in *pages* (similarly to disks) and pages organized in *flash blocks*
  - *Like RAM*, time to retrieve a disk page is <u>not related</u> to location on flash disk



https://premioinc.com/blogs/blog/ssd-vs-hdd-which-is-best-for-your-storage-solution-full-overview

The internals of flash disks

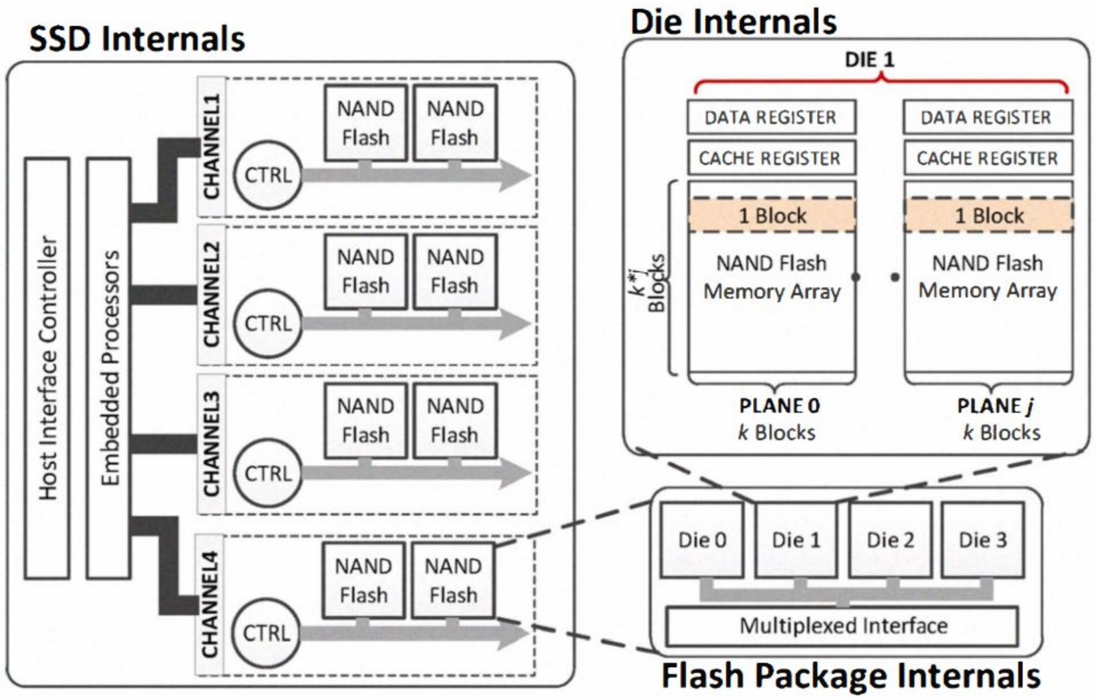
- Interconnected flash chips
- No mechanical limitations
- Maintain the block API – compatible with disks layout
- Internal parallelism in read/write
- Complex software driver



https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6237035

The internals of flash disks

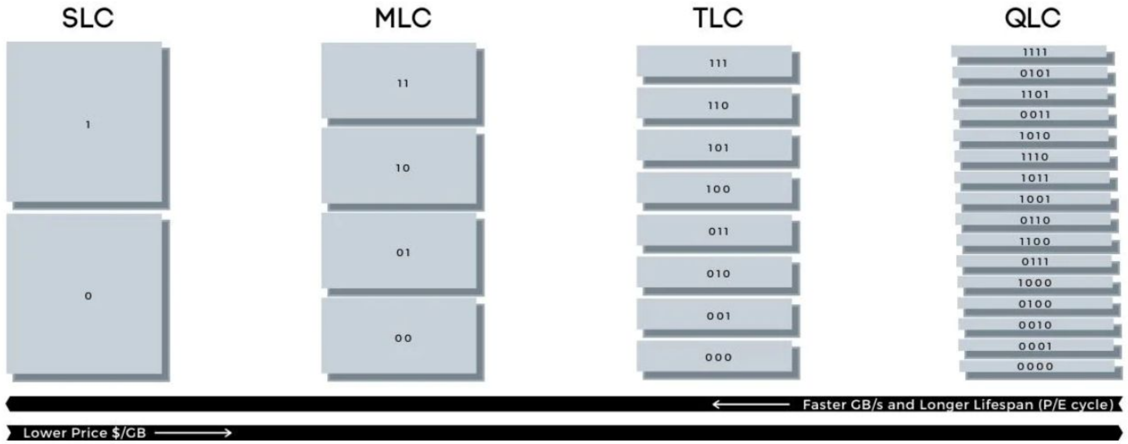
- SSDs have processors for IO operations
- Processors connected via channels to NAND flash package
- Each package have multiple dies (chips)
  - Eg. 4 2GB dies (0–3)
- Each die has multiple blocks organized into planes
  - A block is 64 4 KB pages
  - Can independently access dies



https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6237035

Various types of flash and supported operations

- Operations: read/write/erase
  - Read/write work at page granularity
  - Write only sets bit for cells from 1 to 0
  - Erase sets bits from 0 to 1 (block level)
- Various types of flash: SLC/MLC/TLC/QLC
  - Lifespan of devices vary from 100K–1K P/E (program/erase) cycles
  - Need to handle wear-leveling problem
    - Technique for prolonging the service life of wearable media



https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6237035

Accessing a flash page

- Access time depends on
  - Device organization (internal parallelism)
  - Software efficiency (driver)
  - Bandwidth of flash packages

- Flash Translation Layer (FTL)
  - Provides similar interface as that of HDDs
  - Complex device driver (firmware)
  - Tunes performance and device lifetime

Flash-based vs HDD-based cloud storage

HDD

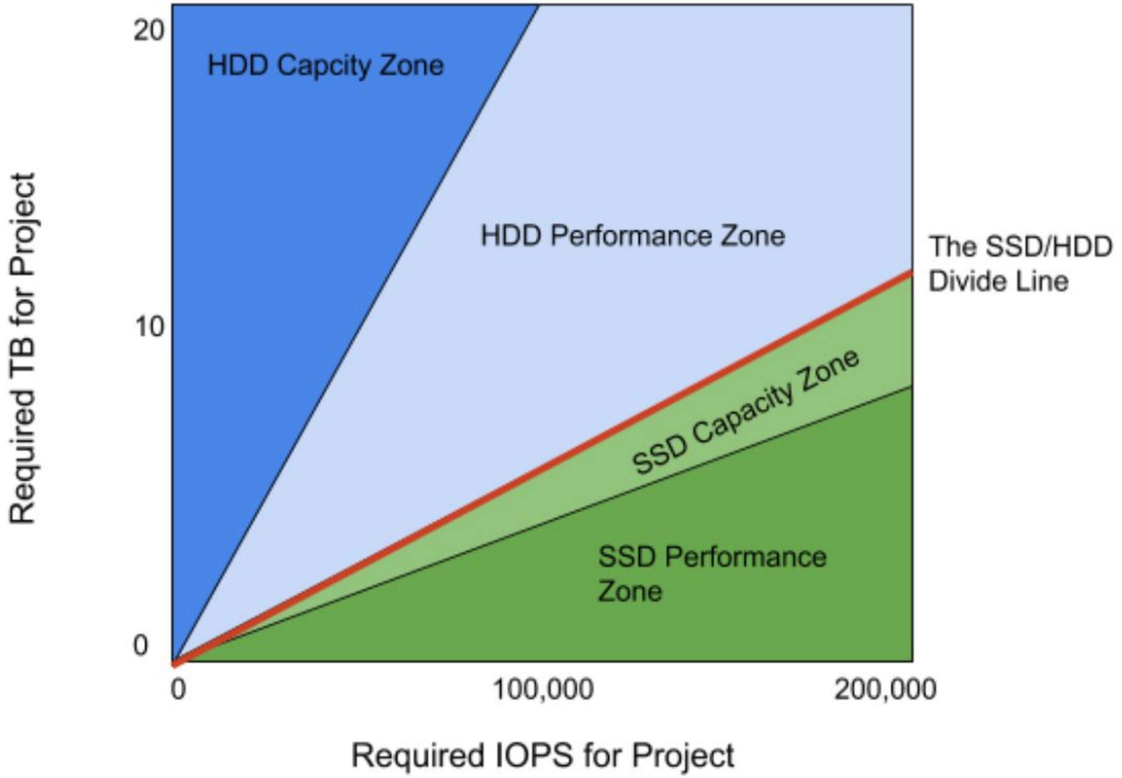
✔ Large – inexpensive capacity

x Inefficient random reads

Flash disks

x Small – expensive capacity

✔ Very efficient random reads



https://www.linkedin.com/pulse/ssd-hdd-practical-rule-storage-selection-public-cloud-cohen-phd/

Rules of thumb for accessing pages …

1. Memory access much faster than disk I/O (~ 1000x)

2. “Sequential” I/O faster than “random” I/O (~ 10x)

Storage requirements

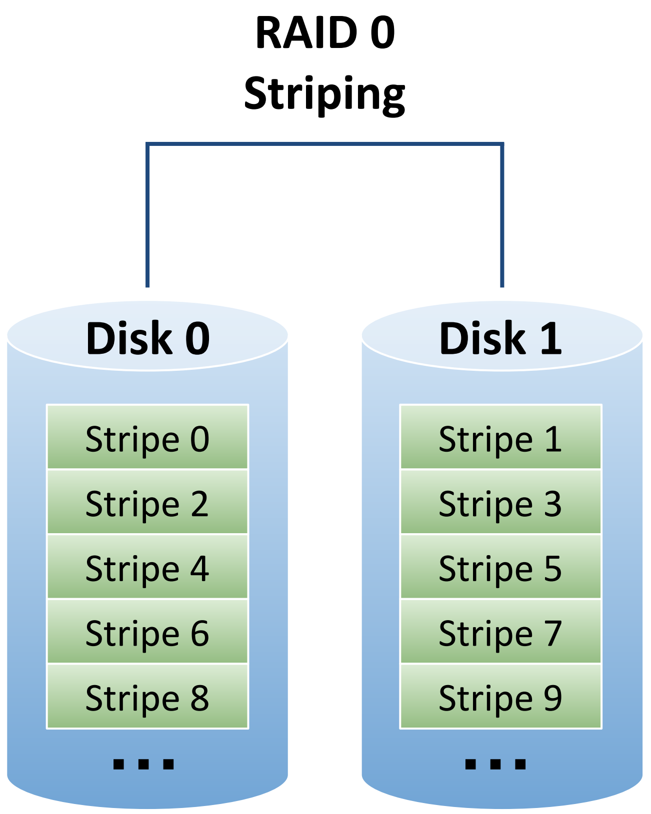
- **Fast:** data is there when you want it
- **Reliable:** data fetched is what you stored
- **Affordable:** won't break the bank

⇒ **Redundant Array of Inexpensive Disks (RAID)**

- Build a logical disk from (many) physical disks
- Benefits:
  - **Higher throughput** (via data "stripping")
  - **Large MTTF** (for redundancy)
    - MTTF: Mean time to failure

RAID 0

- File is striped across disks
- No redundancy of data (no fault tolerance)
  - Data is not mirrored on a different disk
- Provides best performance
  - **Cumulative bandwidth utilization**
- Total storage capacity: sum of capacities of all disks
- **No data security**



Striping and reliability

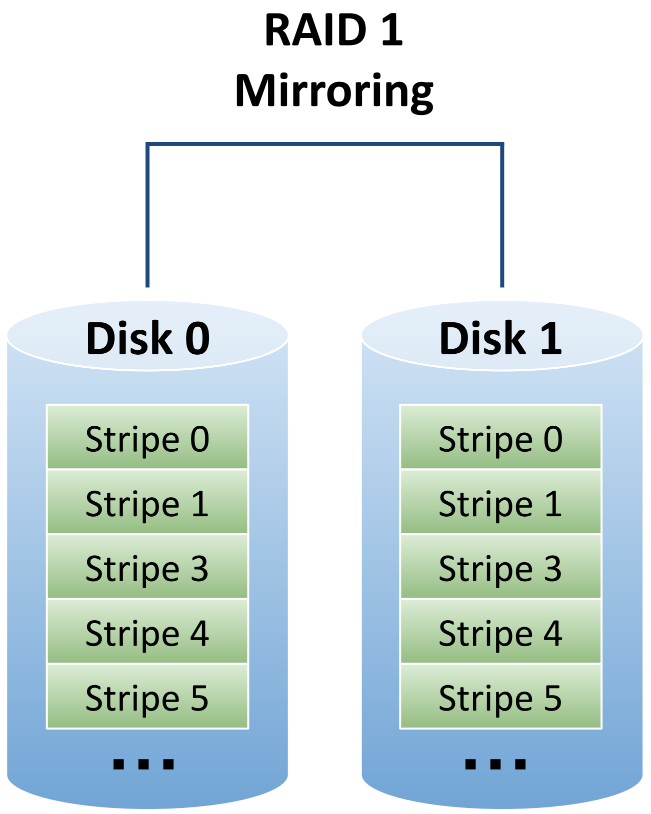
Striping reduces reliability

- More disks → higher probability of some disks failing
- N disks → $1/N^{th}$ mean time between failures of 1 disk

Q. How can we improve disk reliability?

RAID 1

- Duplicate file blocks across storage drives
  - **Deals well with disk loss**
  - **Does not handle corruption**
- Total storage capacity: 1 disk only
- Performance:
  - Reads can be parallelized
  - Writes too,so equivalent to writing to one disk
- **Expensive**
- Used in critical infrastructure (sensitive information)

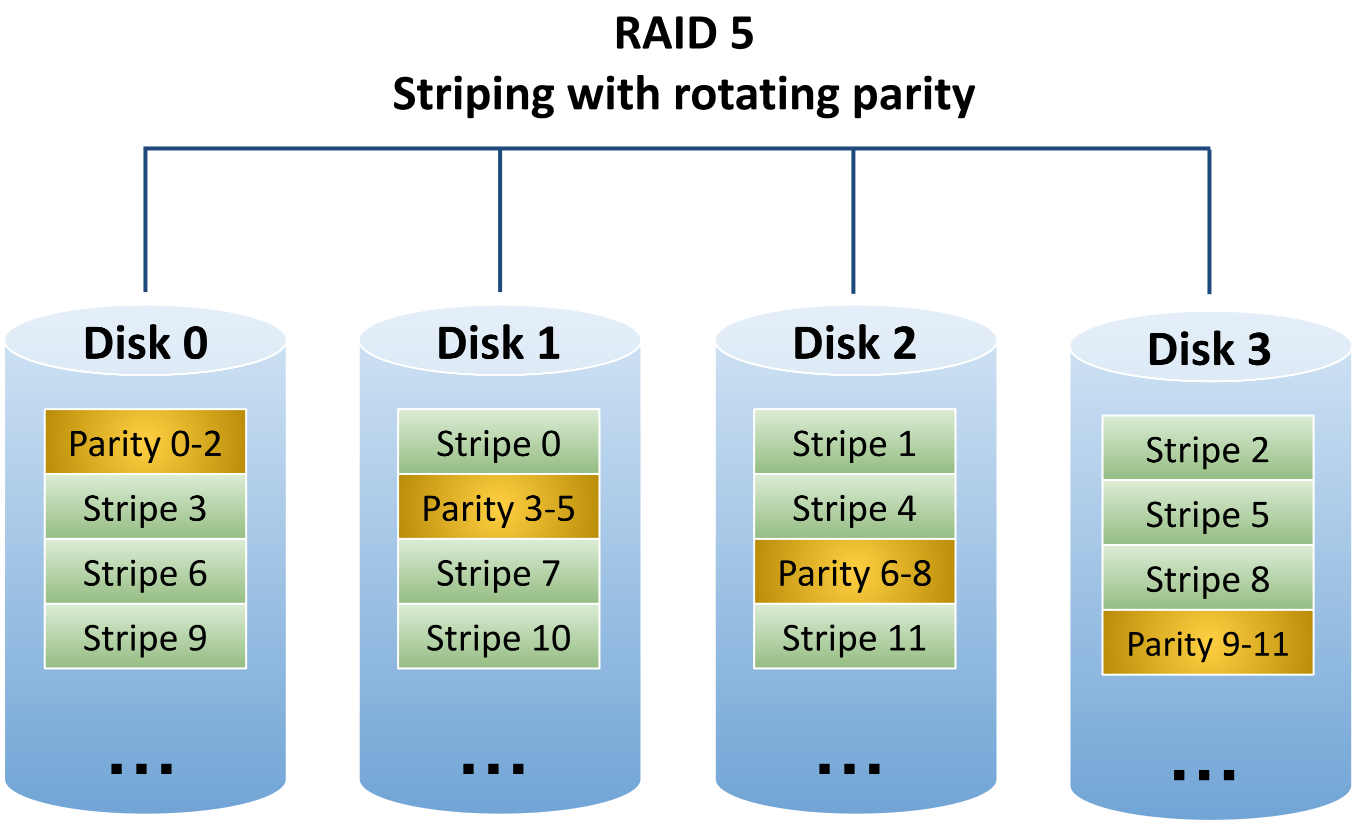


RAID 5

- **Parity:** Another mechanism for fault tolerance.

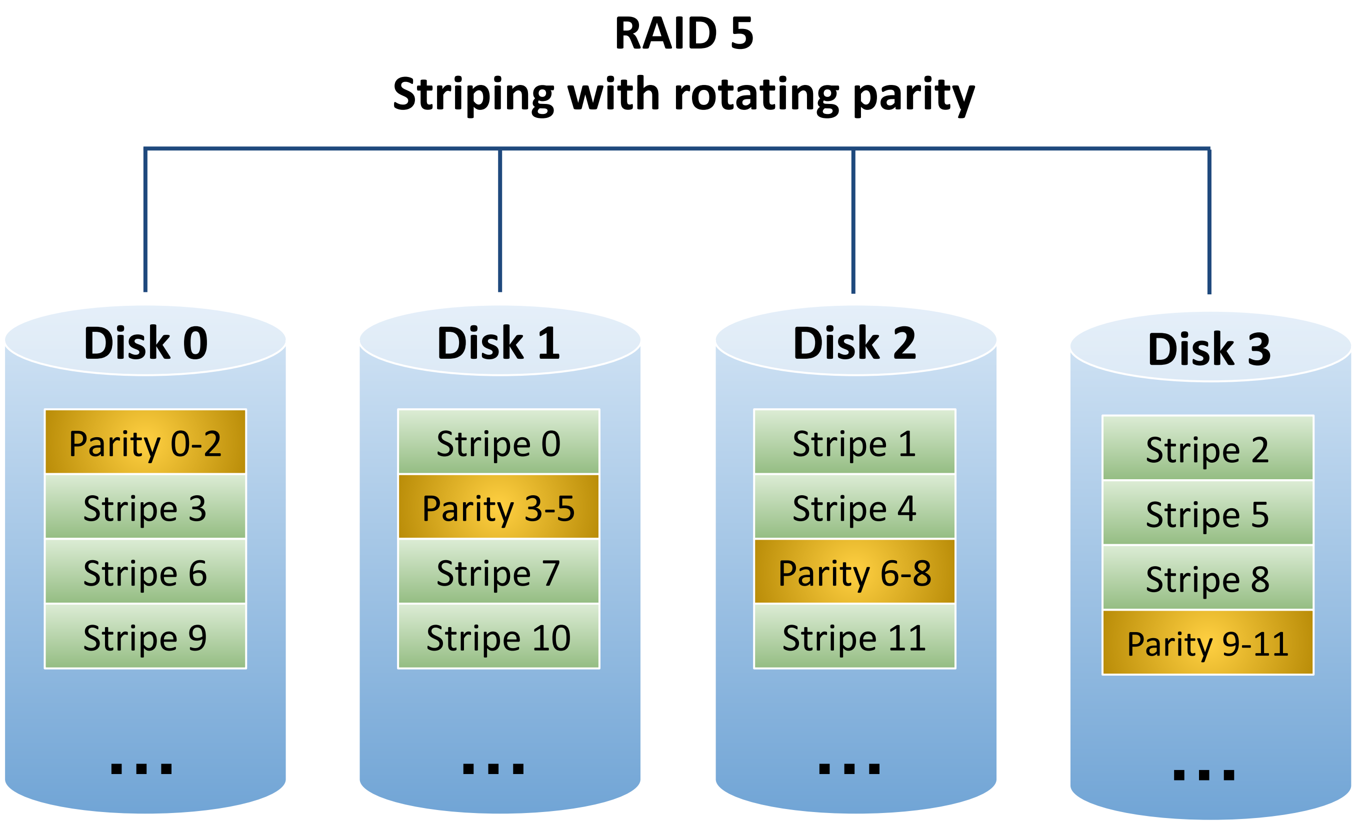
$$P_{i-j} = S_i \oplus S_{i+1} \oplus \ldots S_j$$

- If one disk fails, one can reconstruct its data by XOR-ing all remaining drives



RAID 5

- **Reliable**
  - Still works by losing one disk
- **Fast**
  - **3x** seq. reads of one disk, i.e., (N-1), N ⟶ # drives
  - Writes become complicated
- Affordable
- Used: datacenter environments



Today’s focus

- DBMS layers and storage hierarchy

- Disks
  - HDD
  - SSD

- Buffer management

Readings: Chapter 13.5

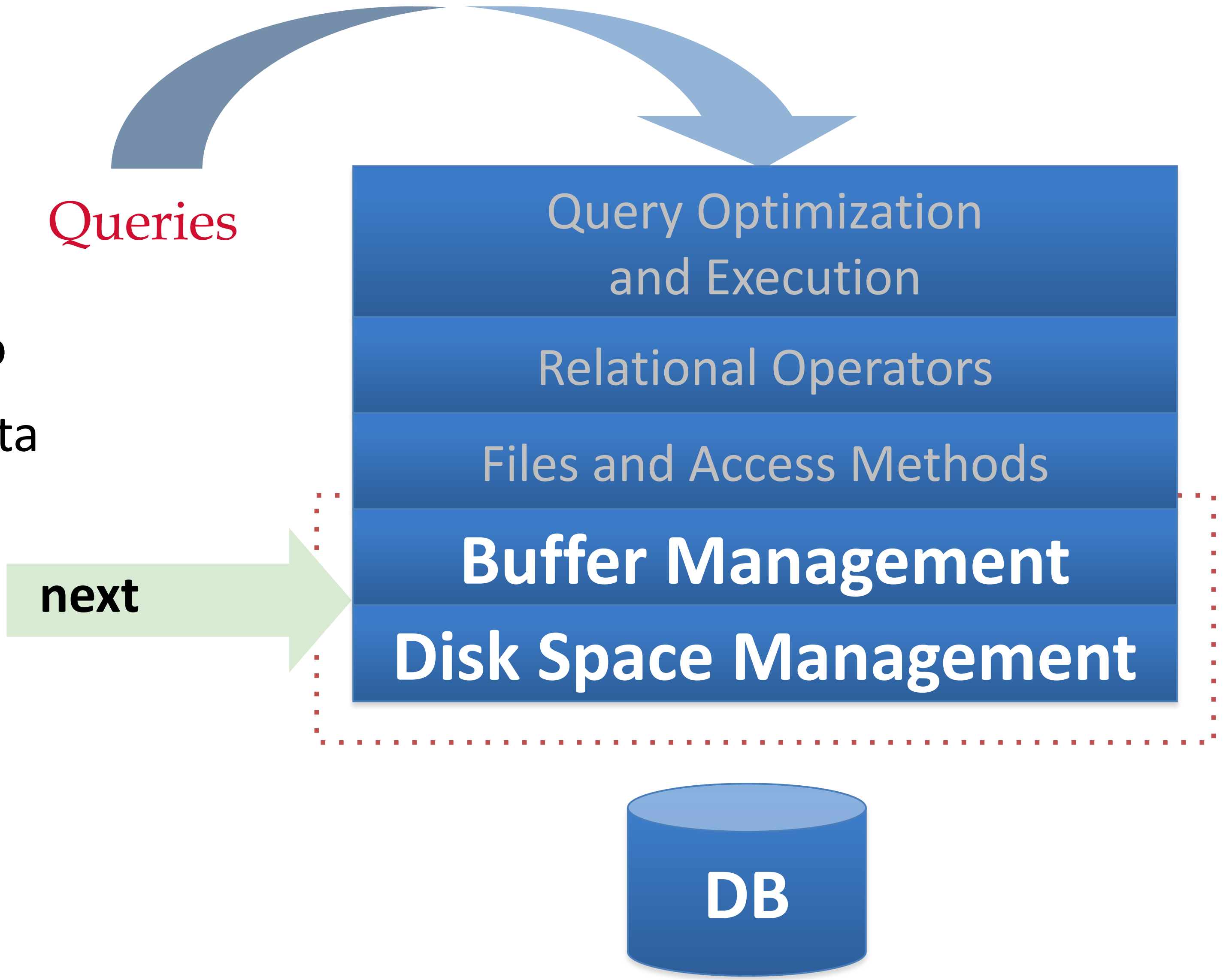
Disk space management

- Lowest layer of DBMS software manages space on disk
- Higher levels call upon this layer to:
  - Allocate/deallocate a page
  - Read/write a page

- Best case scenario: A request for a *sequence* of pages is satisfied by pages stored sequentially on disk
  - Higher levels don’t need to know if/how this is done, or how free space is managed

Recall the big picture

DBMS relies on buffer manager to manage its memory and move data back-and-forth from disk



Buffer management for DBMS

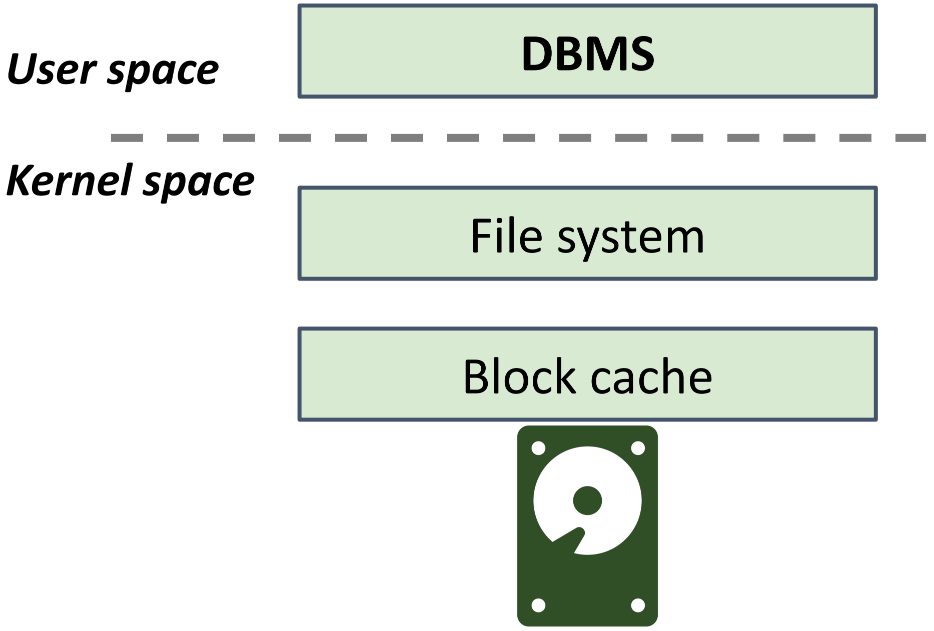
Q. Why not use existing OS services?

Q. How does the buffer pool **maintain** pages?

Q. What is the policy to move pages (eviction/replacement policy)?

Why not rely on OS?

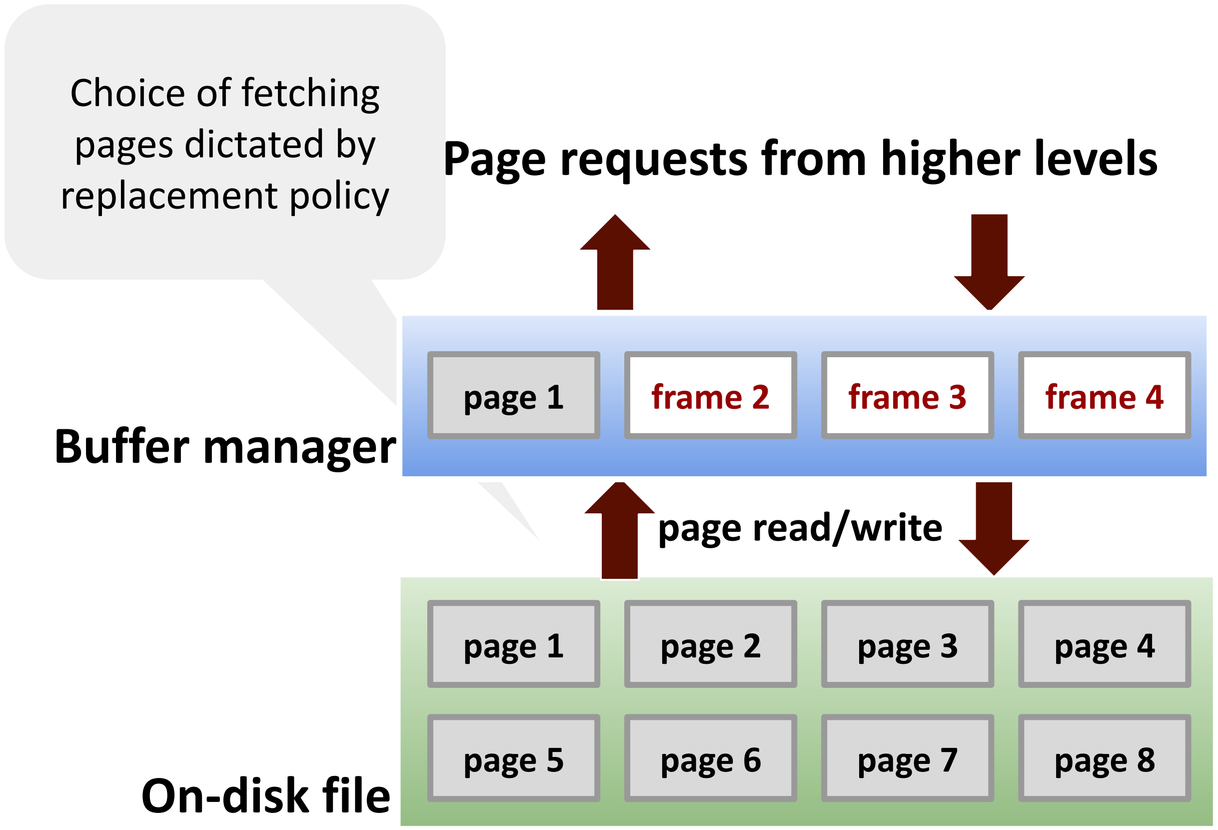
- Layers of abstraction are good … but:
  - Unfortunately, OS often gets in the way of DBMS



- DBMS requires to do things "its own way"
  - **Specialized prefetching**
  - **Control over buffer replacement policy**
  - **Control over flushing data to disk**
    - WAL protocol requires flushing log entries to disk
  - **Control over thread/process scheduling**
    - Convoy problem: arises when OS scheduling conflicts with DBMS locking

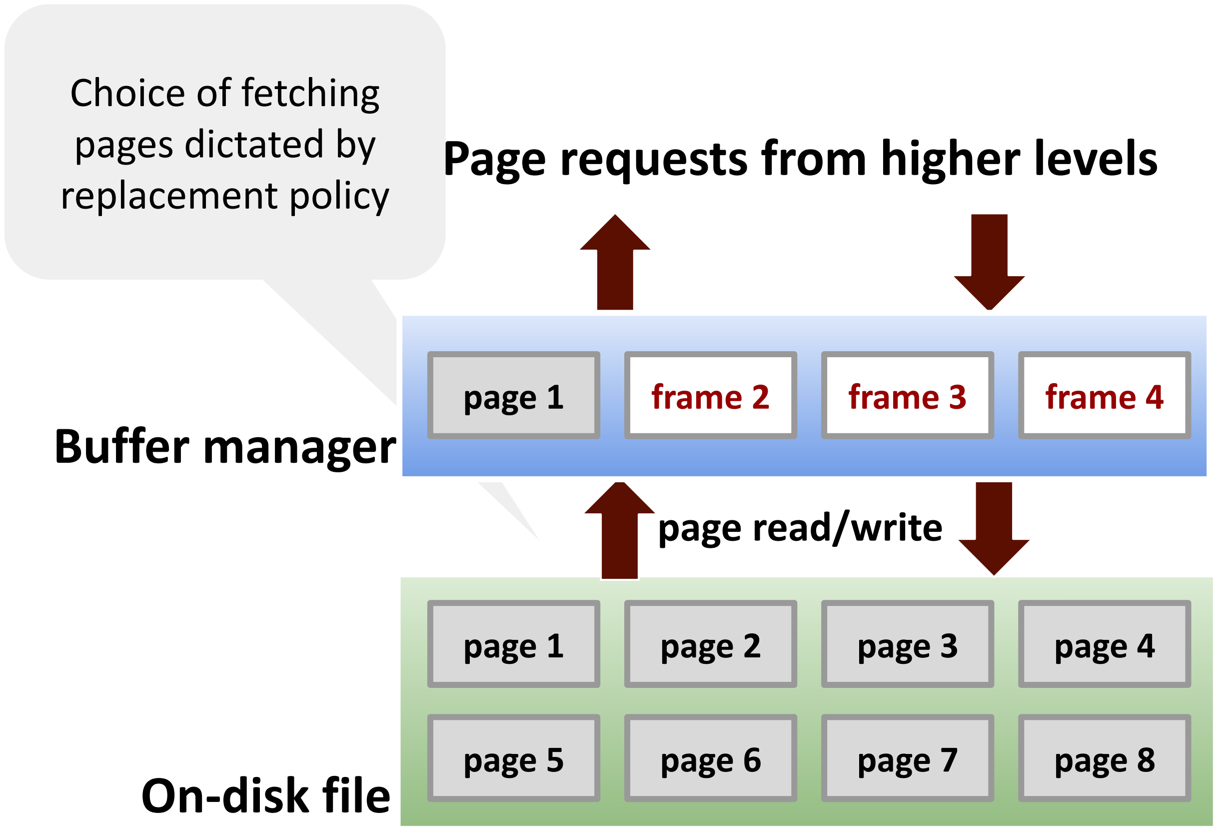
Buffer management in a DBMS

- Data must be in DRAM for DBMS to operate on it
- Buffer manager hides the fact that not all data is in DRAM
    - Similar to hardware caches that hide that not all data are in the caches
- Similar to block cache, replacement policies determine which pages to keep in DRAM vs. write them back



Buffer manager

- Reduces IO by buffering/caching
- Keeps active pages in memory
- Limited size, discards/write back pages when needed
- Also important for recovery, using logging

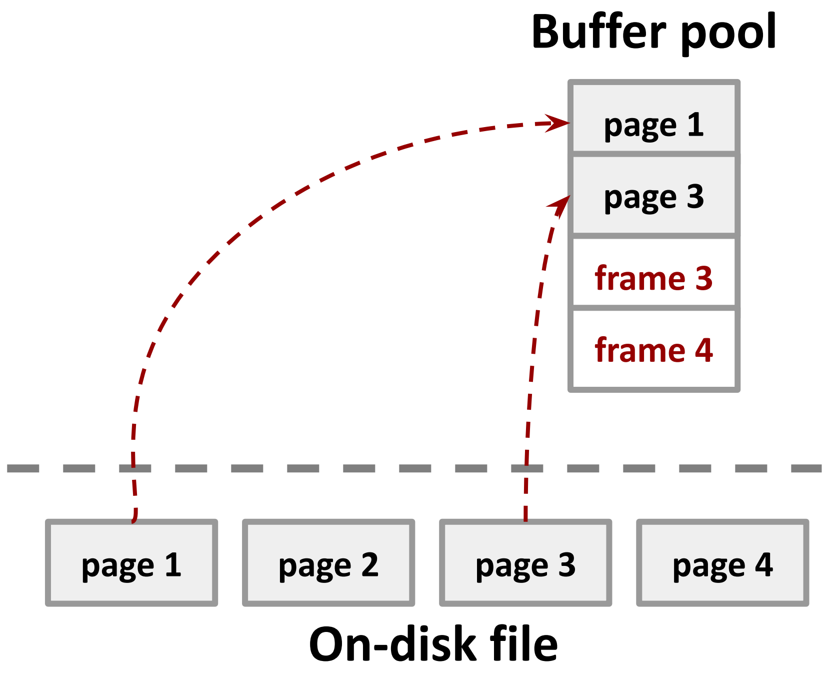


Buffer manager keeps track of pages

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

- On requesting a page, an exact copy is placed into one of these frames

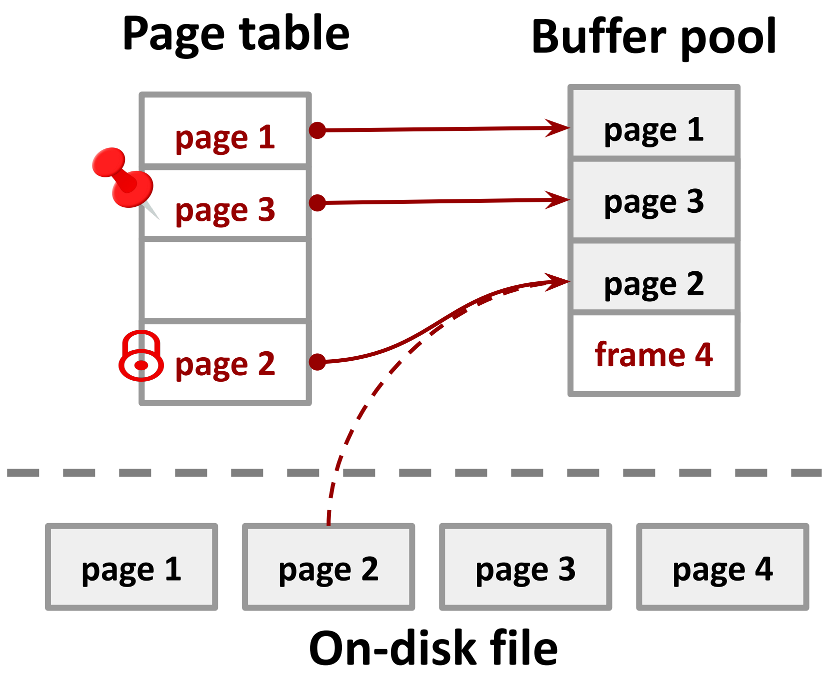
- Dirty pages are buffered in the pool to avoid disk stalls



Buffer manager also maintains metadata for pages

- The **page table** maps page IDs to a copy of the page in buffer pool frames
  - In-memory data structure that is not stored on the disk

- Also, maintains additional metadata per-page:
  - **Dirty flag:** A page has been updated
  - **Pin/reference counter:** A marker or a counter indicating if a page is being referenced by higher levels of DBMS (within a transaction)



Getting a page to the pool when requested …

- If a requested page is not present in the pool, first find a frame:
  - If there is an empty frame, read the requested page into the frame
  - OR choose a frame for ***replacement***

    (only un-pinned pages are candidates)
  - If the frame is "dirty", write it to disk
  - Read requested page into chosen frame
- *Pin* the page and return its address

- If requests can be predicted (e.g., sequential scans)
  - Buffer manager can **pre-fetch** several pages at a time!

More on buffer management

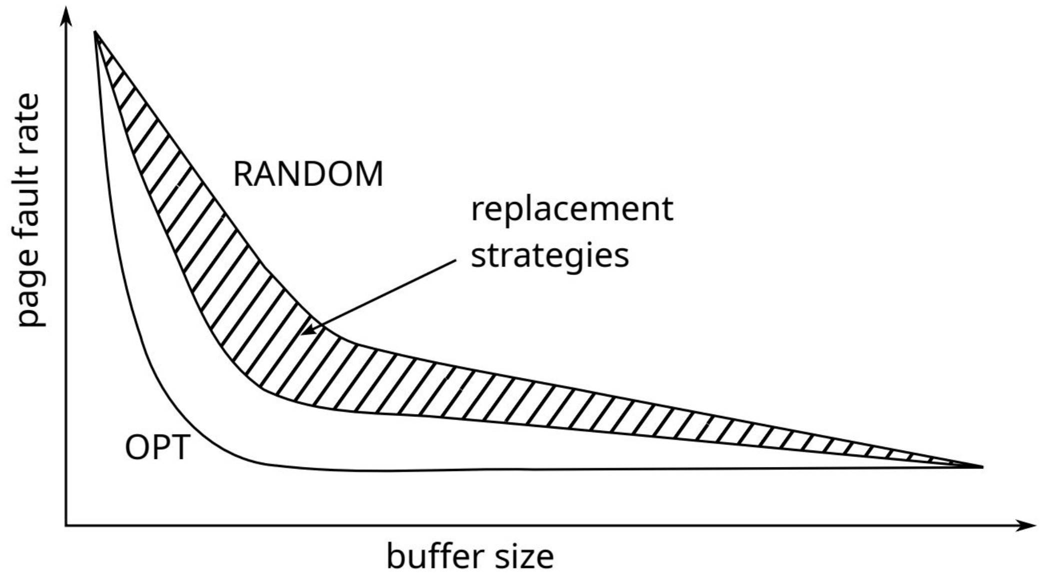
- Requestor of page must unpin it, and indicate whether page has been modified:
  - ***dirty*** bit is used for this

- A page in pool may be requested several times
  - A ***pin count*** is used
  - A page is a candidate for replacement iff *pin count* is 0 (*“unpinned”*)

- CC & recovery may entail additional I/O when a frame is chosen for replacement (*write-ahead log* protocol; more later)

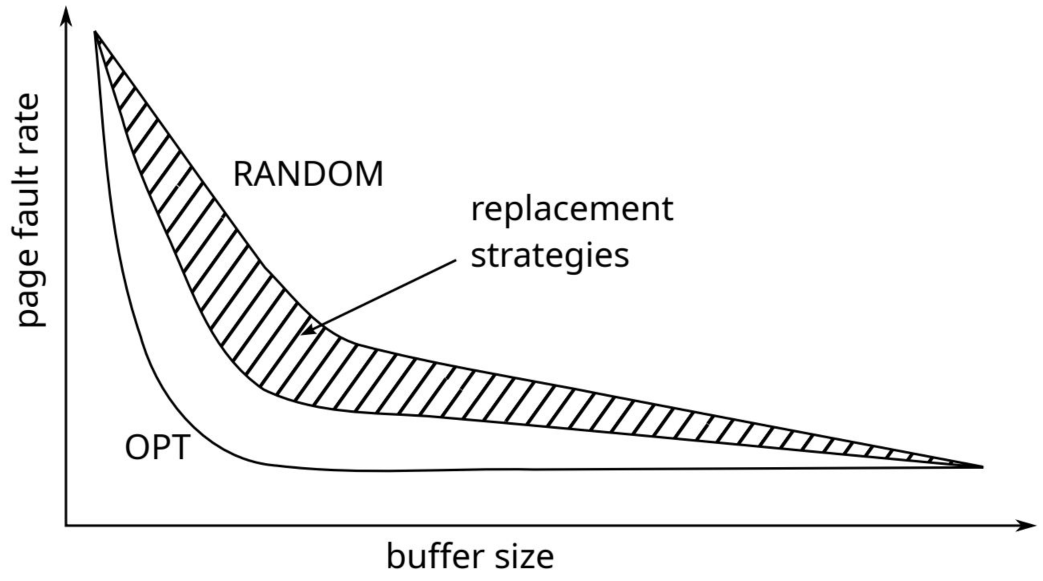
Buffer replacement policies

- When memory is full, some buffer pages have to be replaced
  - Discard clean pages
  - First write the dirty pages back to to the disk
  - Then replaced the discarded pages with new pages
- Goals: Correctness, accuracy, speed, metadata overhead



Buffer replacement policies

- Several policies have been explored:
    - **First in-first out (FIFO)**
    - **Least recently used (LRU)**
    - Most frequently used (MFU)
    - **Clock**
    - LRU-K
    - 2Q



Buffer replacement policy: FIFO

First in-first out (FIFO)

- Simple replacement policy
- Keeps those pages that were most recently added to the buffer pool
- Maintain a linked list of buffer frames
- Insert at the end, remove from the head
- “Old” pages are removed first

Does not retain frequently-used pages (or locality)

Buffer replacement policy: LRU

Least recently used (LRU)

- Simple replacement policy
- Maintain a timestamp of when each page was last accessed (***unpinned***)
- When the DBMS needs to evict a page, select the one with **oldest access timestamp**
    - Keep the pages in sorted order to reduce the search time on eviction
    - Doubly-linked list stores the buffer frames
    - Remove from the head
    - When a frame is unpinned, move it to the end of the list
    - “Hot” pages are retained in the buffer

One of the most widely-used policy

Buffer replacement policy: Clock

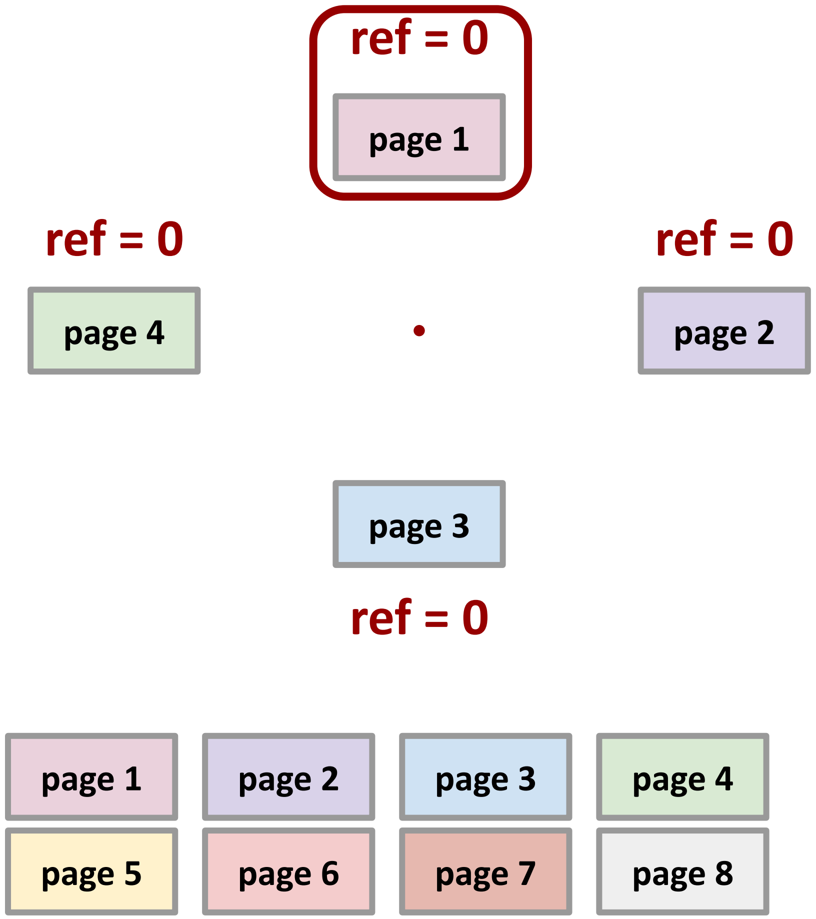
- An approximation of LRU that does not need a separate timestamp per page
  - Each page has a **reference bit**
  - When a page is accessed, set it to 1

- 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

Buffer replacement policy: Clock

- Replacing pages:

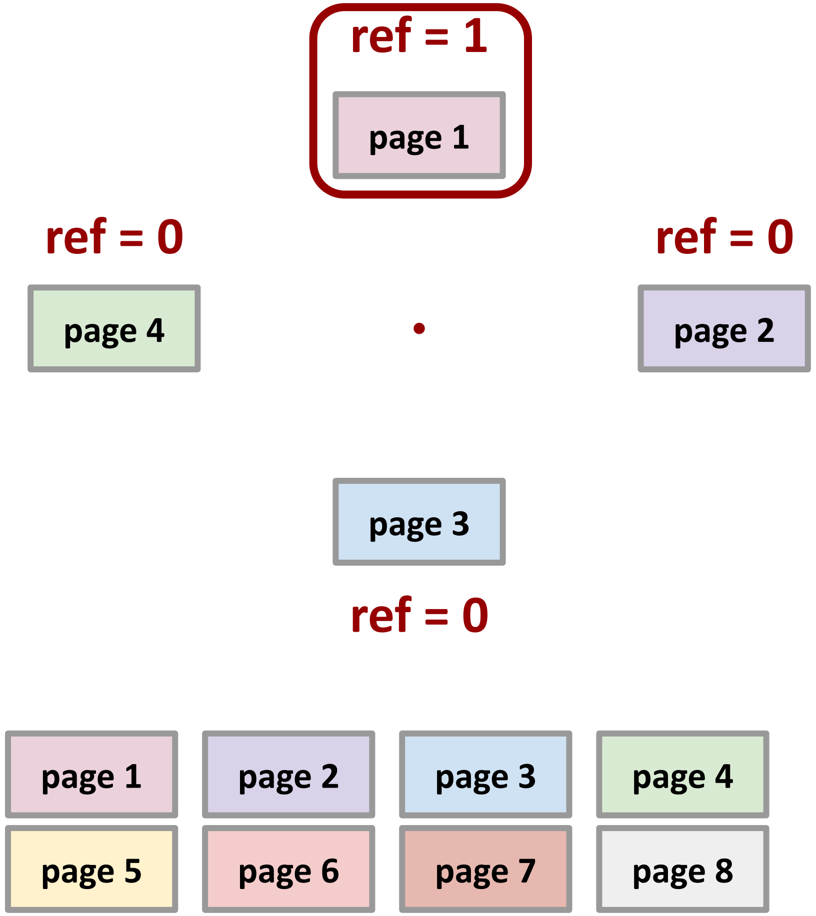
```
do {
  if (pincount == 0 && ref bit is off)
     choose current page for replacement;
  else if (pincount == 0 && ref bit is on)
     turn off ref bit;
  advance current frame;
} until a page is chosen for replacement;
```



Buffer replacement policy: Clock

- Replacing pages:

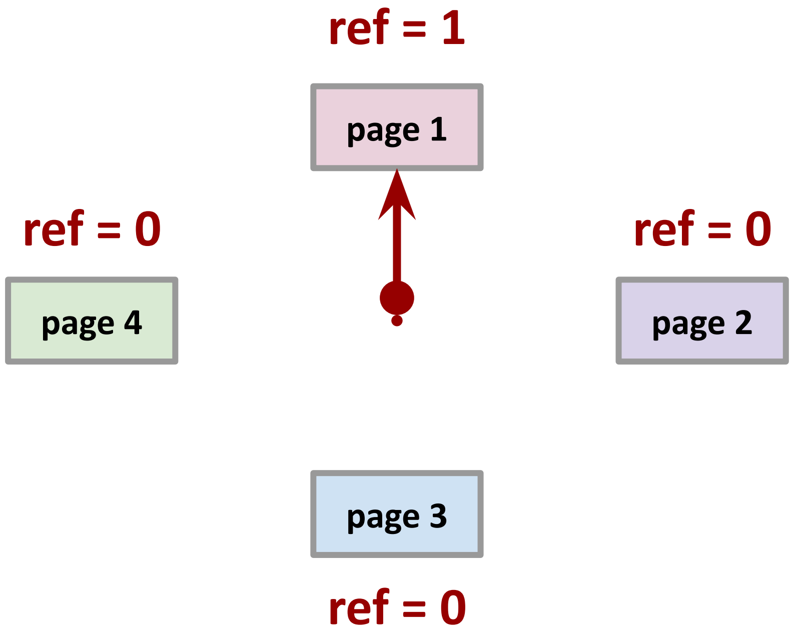
```
do {
  if (pincount == 0 && ref bit is off)
     choose current page for replacement;
  else if (pincount == 0 && ref bit is on)
     turn off ref bit;
  advance current frame;
} until a page is chosen for replacement;
```

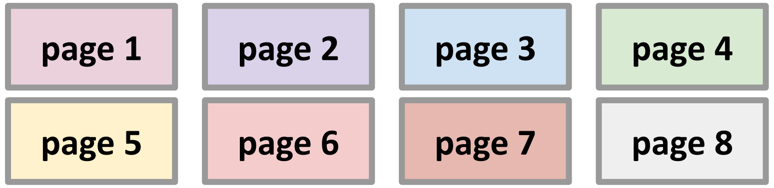


Buffer replacement policy: Clock

- Replacing pages:

```
do {
  if (pincount == 0 && ref bit is off)
     choose current page for replacement;
  else if (pincount == 0 && ref bit is on)
     turn off ref bit;
  advance current frame;
} until a page is chosen for replacement;
```

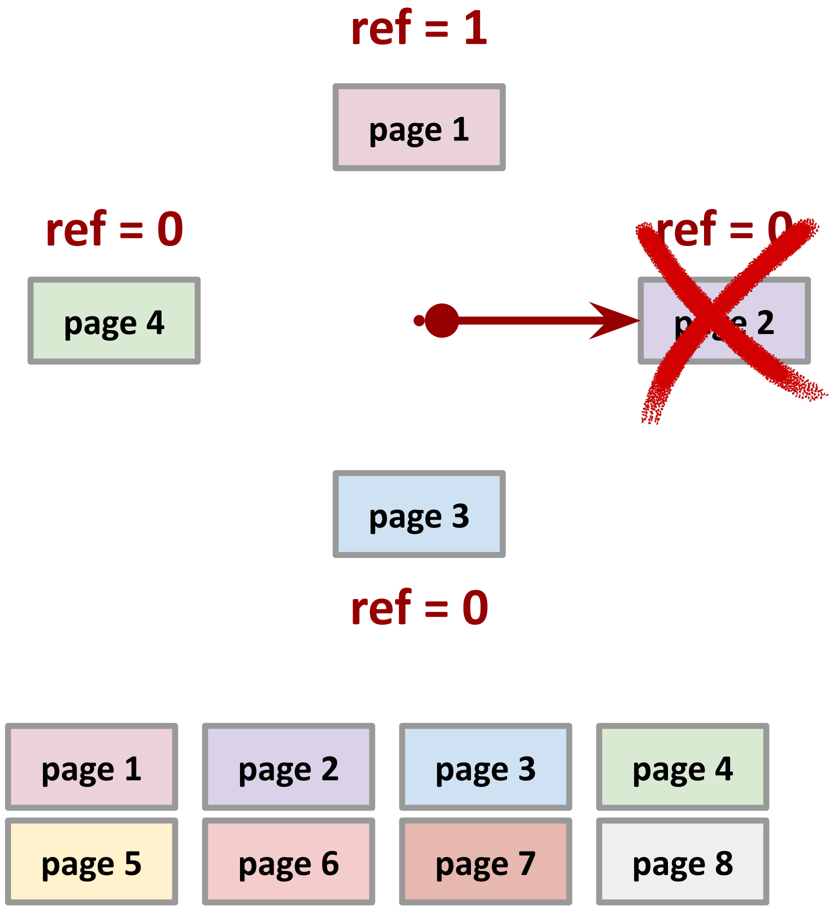




Buffer replacement policy: Clock

- Replacing pages:

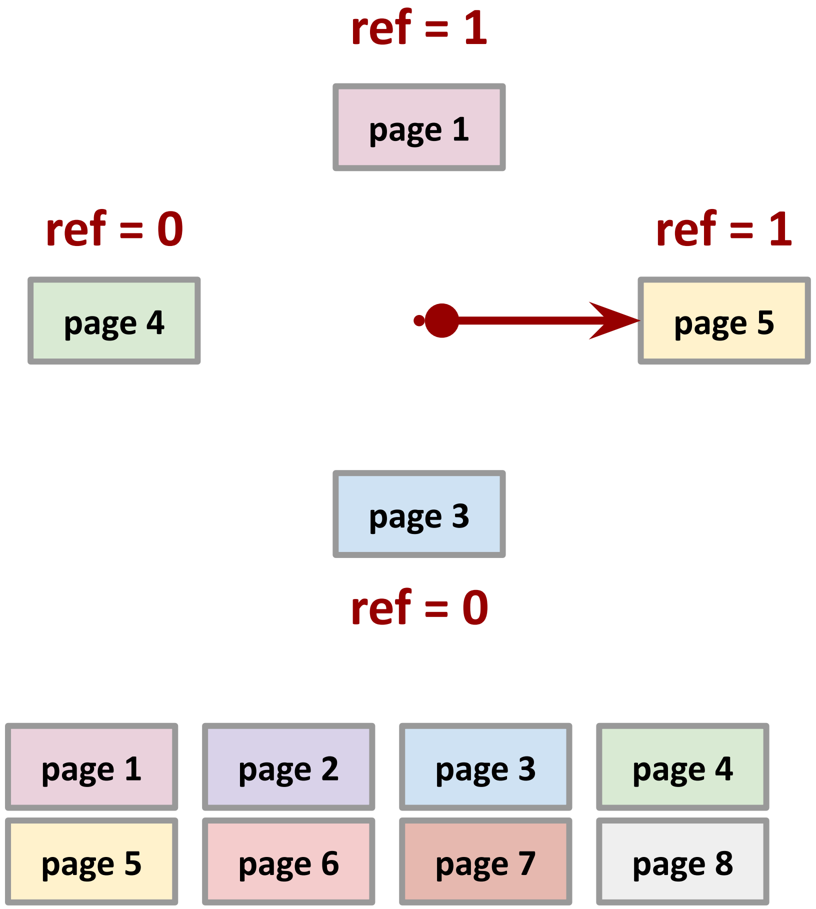
```
do {
  if (pincount == 0 && ref bit is off)
     choose current page for replacement;
  else if (pincount == 0 && ref bit is on)
     turn off ref bit;
  advance current frame;
} until a page is chosen for replacement;
```



Buffer replacement policy: Clock

- Replacing pages:

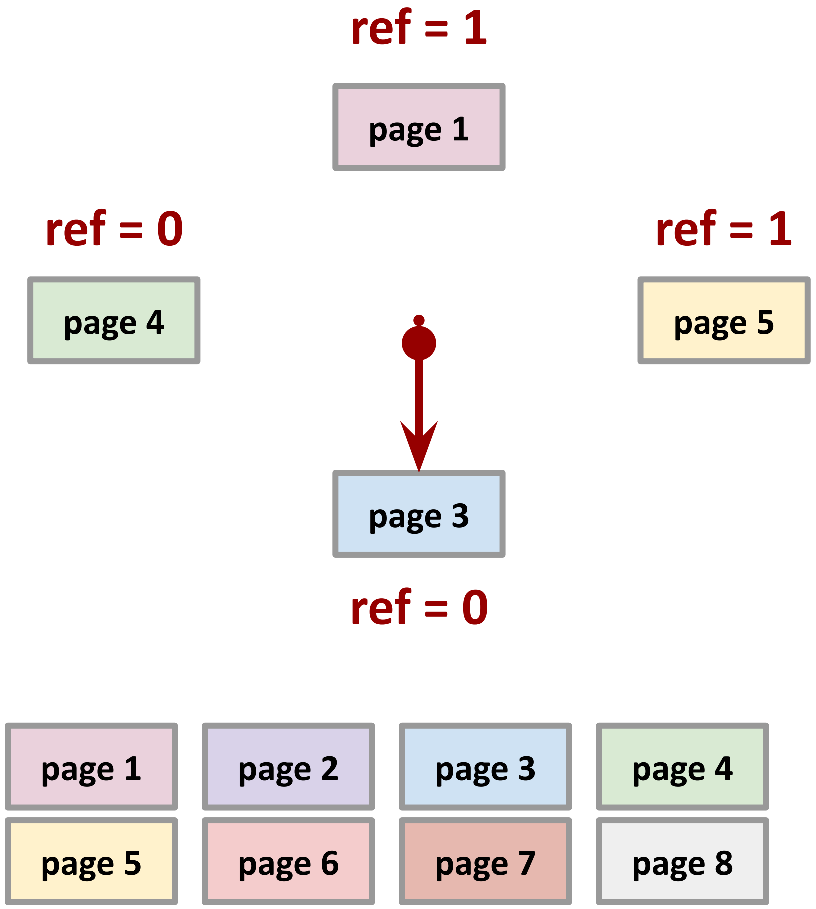
```
do {
  if (pincount == 0 && ref bit is off)
     choose current page for replacement;
  else if (pincount == 0 && ref bit is on)
     turn off ref bit;
  advance current frame;
} until a page is chosen for replacement;
```



Buffer replacement policy: Clock

- Replacing pages:

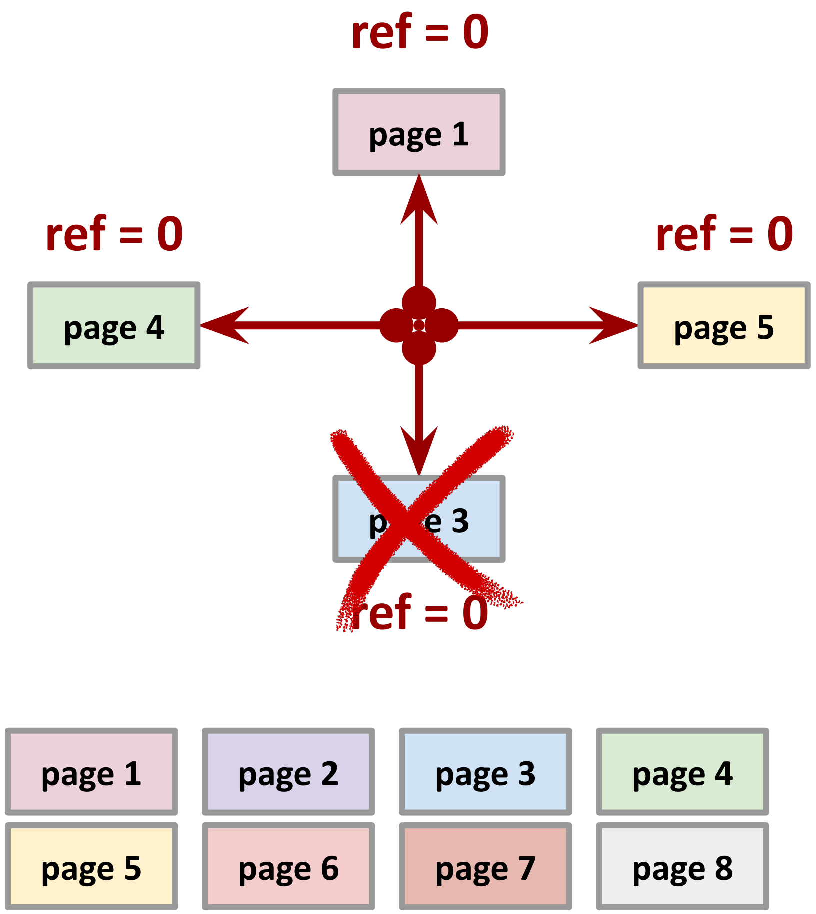
```
do {
  if (pincount == 0 && ref bit is off)
     choose current page for replacement;
  else if (pincount == 0 && ref bit is on)
     turn off ref bit;
  advance current frame;
} until a page is chosen for replacement;
```



Buffer replacement policy: Clock

- Replacing pages:

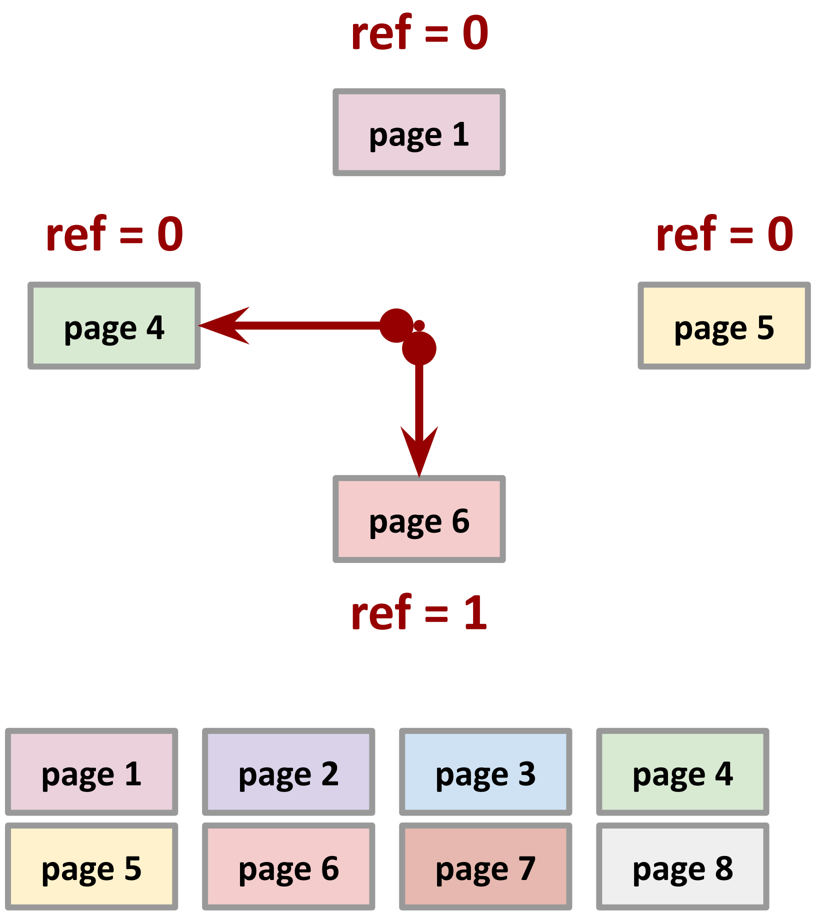
```
do {
  if (pincount == 0 && ref bit is off)
     choose current page for replacement;
  else if (pincount == 0 && ref bit is on)
     turn off ref bit;
  advance current frame;
} until a page is chosen for replacement;
```



Buffer replacement policy: Clock

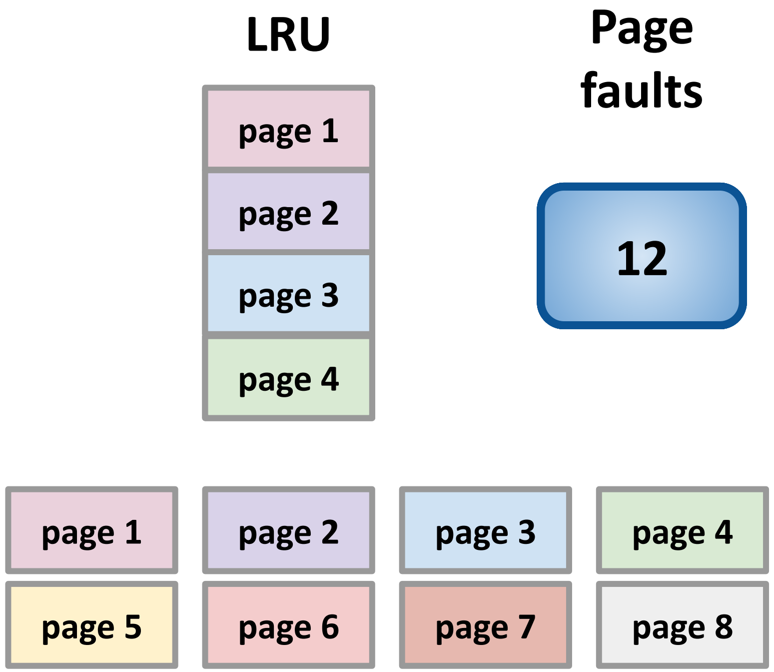
- Replacing pages:

```
do {
  if (pincount == 0 && ref bit is off)
     choose current page for replacement;
  else if (pincount == 0 && ref bit is on)
     turn off ref bit;
  advance current frame;
} until a page is chosen for replacement;
```



Problem with LRU and clock

- LRU and clock are susceptible to **sequential flooding** (no scan resistance)
  - 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
  - # buffer frames < # pages in file means each page request causes an I/O
  - LRU-K and 2Q minimize this issue



Buffer replacement policy: LRU-K

Least recently used - K (LRU)

- Track the history of last K references to each page as timestamps and compute the interval between subsequent accesses
- DBMS uses this history to estimate the next time that page is going to be accessed
- Becomes classic LRU when K = 1
- LRU-K is scan resistant

Buffer replacement policy: 2Q

Maintains two queues (FIFO and LRU)

- Some pages are accessed only once (eg, sequential scan)
- Some pages are hot and accessed frequently
- Maintain separate lists for those pages
- Scan resistant policy

Approach:

1. First maintain all pages in FIFO queue
2. When a page that is currently in FIFO is referenced again, upgrade it to LRU queue
3. Prefer evicting pages from FIFO queue

Hot pages are in LRU, read-once pages are in FIFO: good strategy for DBMS

Summary

- Disks provide cheap, non-volatile storage
  - Random access, but cost depends on location of page on disk; important to arrange data sequentially to minimize *seek* and *rotation* delays

- Buffer manager brings pages into RAM
  - Page stays in RAM until released by requestor
  - Written to disk when frame chosen for replacement (which is sometime after requestor releases the page)
  - Choice of frame to replace based on *replacement policy*
  - Good to *pre-fetch* several pages at a time

Buffer manager implementation

- The hash table uses latches for mutual exclusion
- PageNo: The page number
- Latch: A read/write lock to protect the page
- LSN: log sequence number

  (a unique identifier associated with every record in a transactional log)
- State: Clean/dirty/new created etc.
- Data: The actual data contained on the page