THE UNIVERSITY of EDINBURGH

Advanced Databases

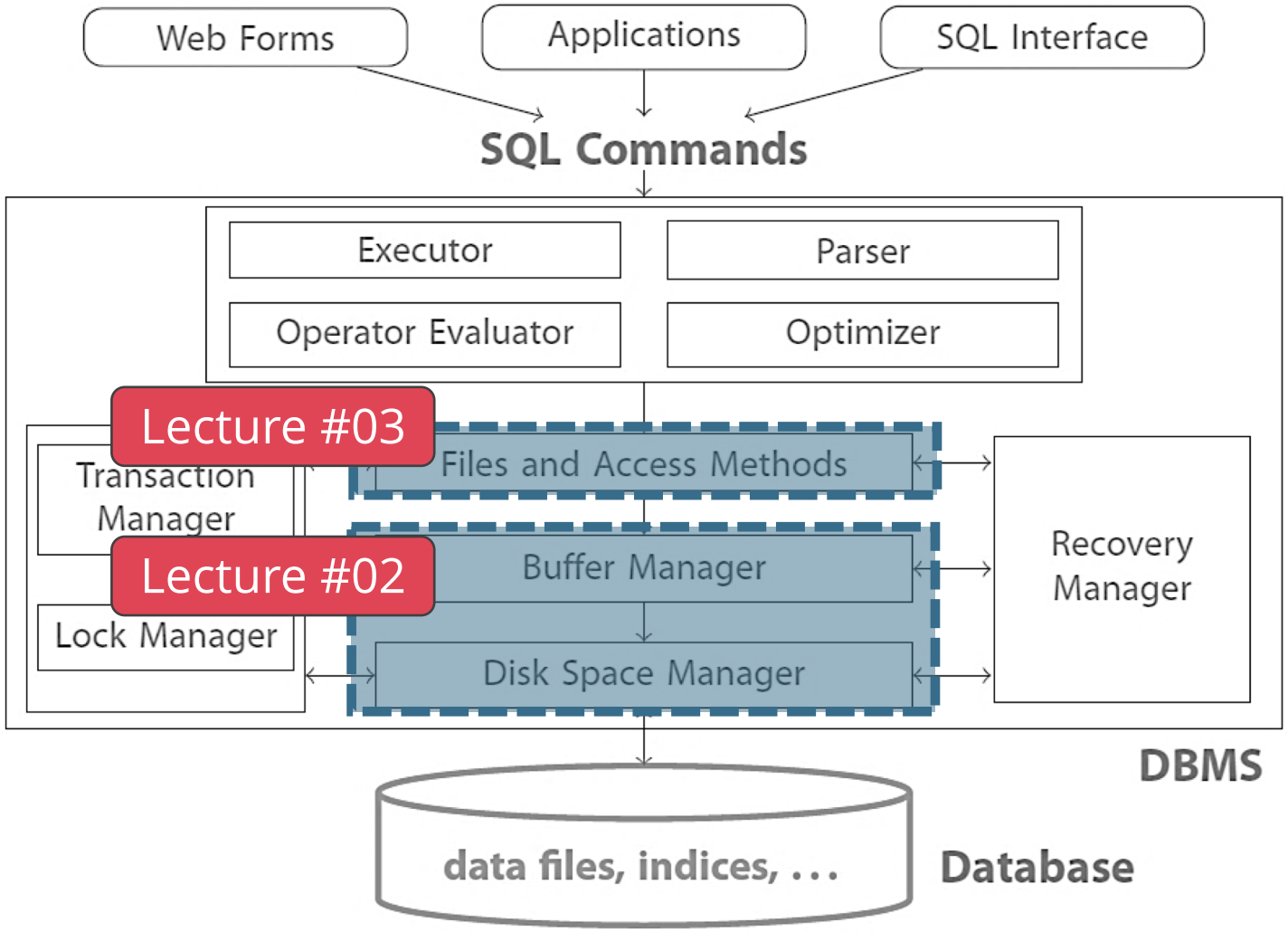
Spring 2020

Lecture #02:

Database Storage I

Milos Nikolic

DATABASE ARCHITECTURE



DISK-ORIENTED ARCHITECTURE

The DBMS assumes the primary storage location of the database is on a non-volatile disk

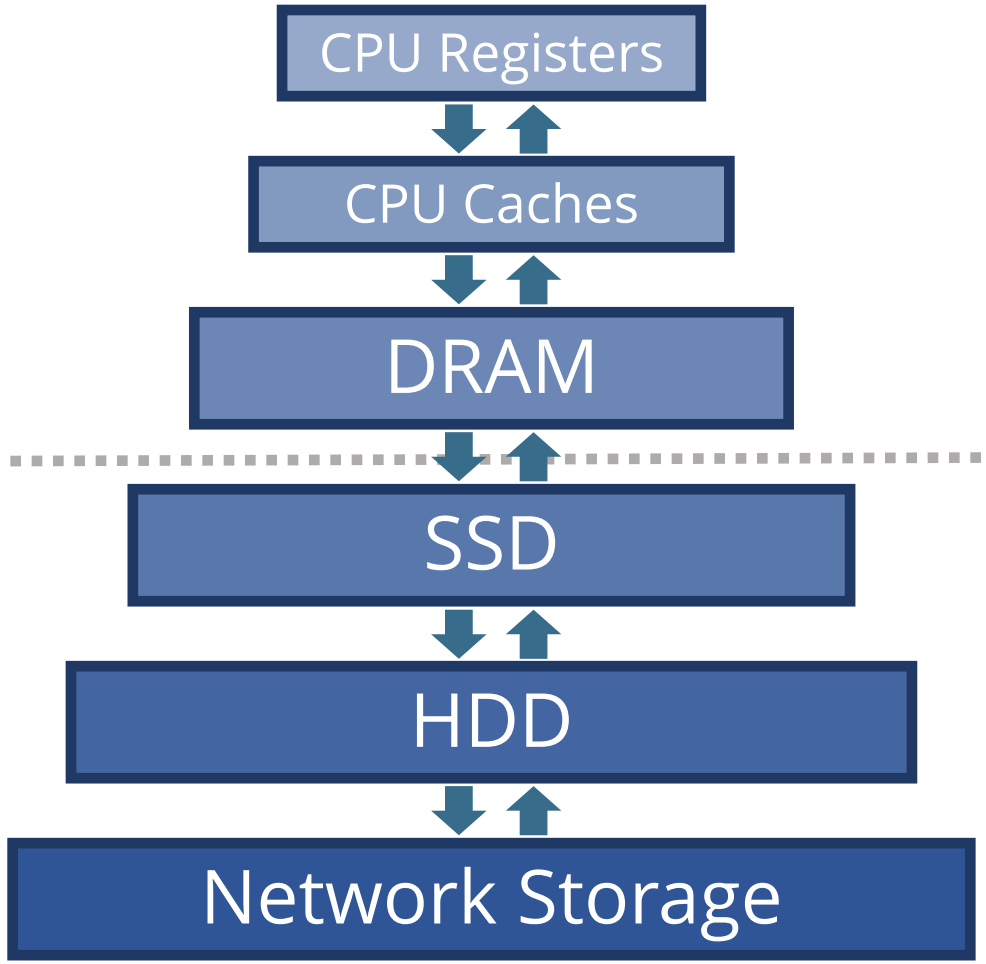
The DBMS's components manage the movement of data between non-volatile and volatile storage

* Volatile storage only maintains its data while the device is powered

Storage Hierarchy

Volatile
Byte-Addressable

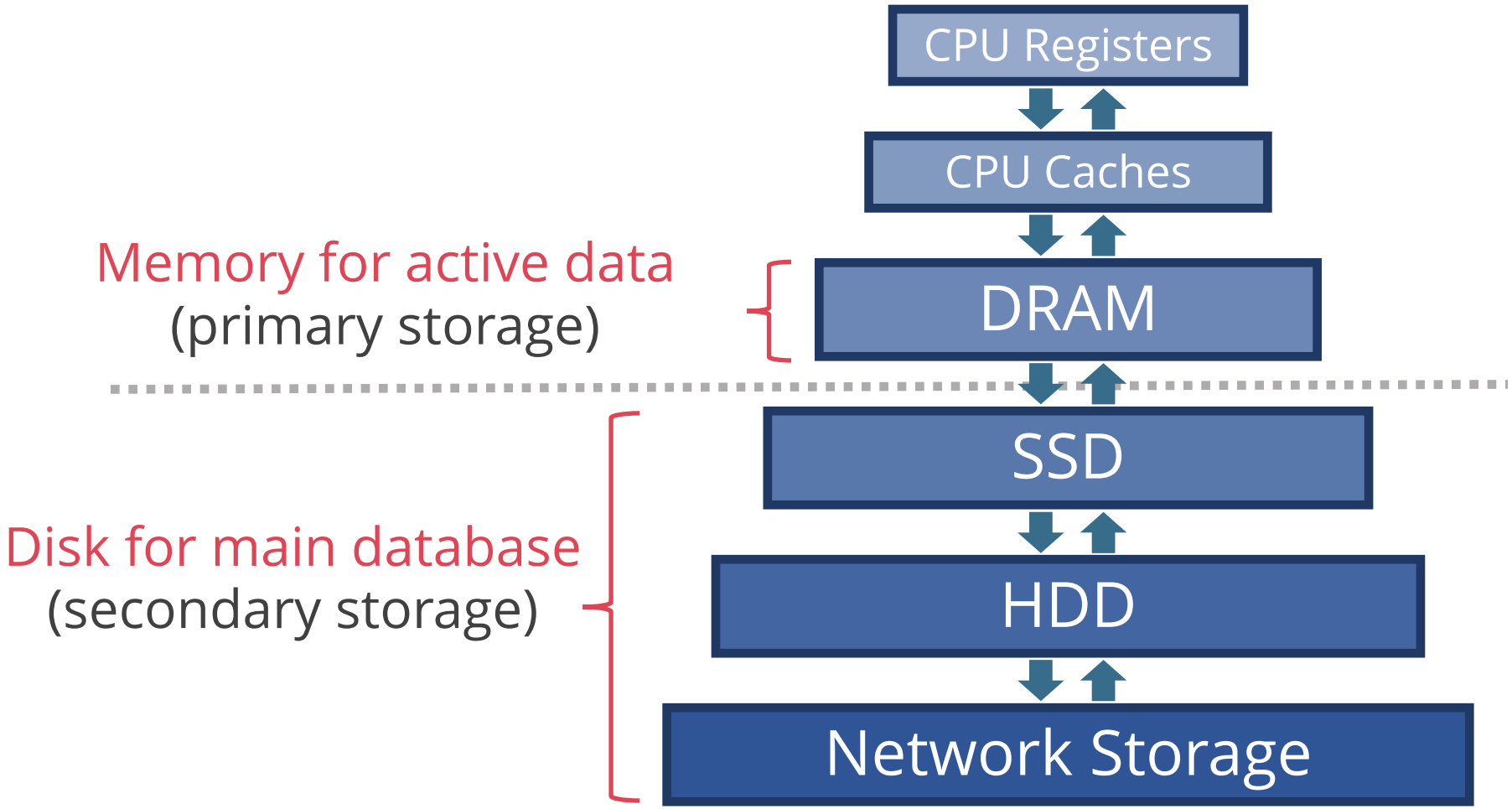
Non-Volatile
Block-Addressable



Faster
Smaller

Slower
Larger

Storage Hierarchy



Latency	Capacity
< 1ns	B
< 10ns	KB/MB
100ns	GB
0.1ms	GB/TB
10ms	TB
30ms	PB

Why Not Store All in Main Memory?

Costs too much

1TB of disk costs ~50$ vs. 1TB of RAM costs ~6000$

High-end databases today in the petabyte range!

Roughly 60% of the cost of a production system is in the disks

Main memory is volatile

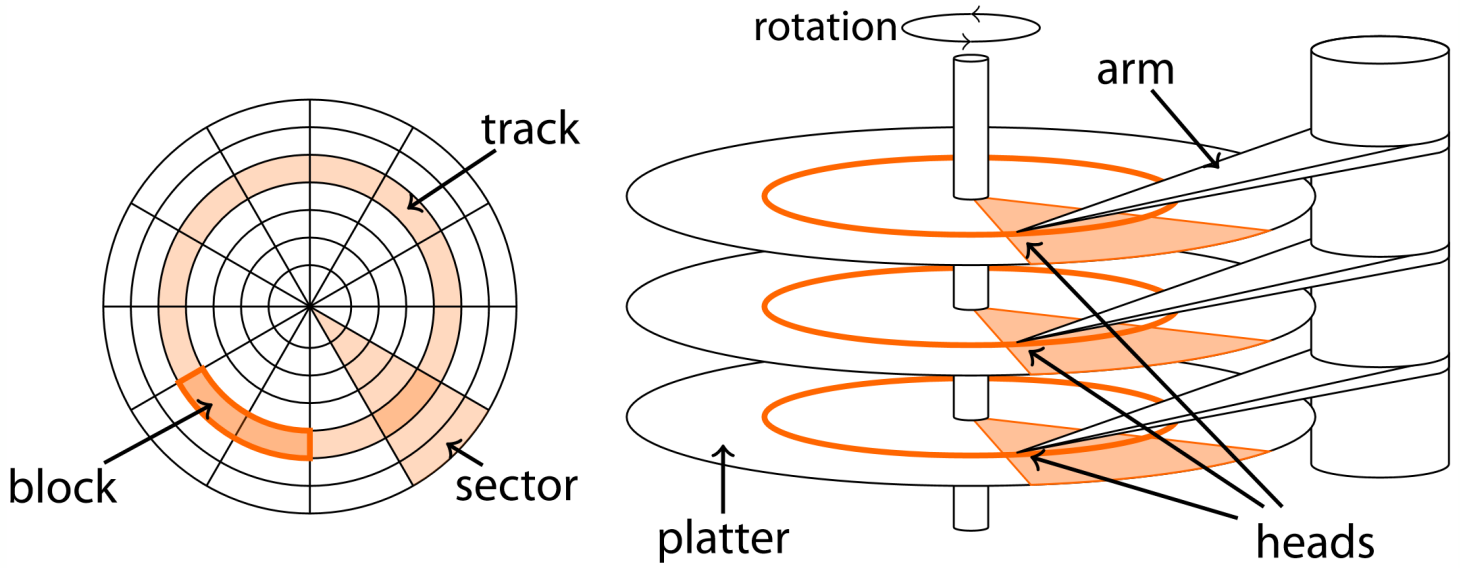
Obviously important if DB stops/crashes. We want data to be saved!

* In-memory DBMSs **do** store entire databases in main memory **(not covered in this course)**

Faster than disk-oriented but with orders of magnitude higher cost/GB

Anatomy of a Disk

Video on how disk drives work



Data is stored and retrieved in units called disk blocks

Unlike RAM, time to retrieve a block depends on its location

Access time = seek time + rotational delay + transfer time

move arms to track

wait for target block in track

read/write data

Seagate Cheetah 15K.7

4 disks, 8 heads, avg. 512 KB/track, 600GB capacity

rotational speed: 15 000 rpm

average seek time: 3.4 ms

transfer rate ≈ 163 MB/s

Access time to read one block of size 8KB

Average seek time		3.40 ms
Average rotational delay	1/2 · 1/15000 min	2.00 ms
Transfer time	8KB / 163 MB/s	0.05 ms
Total access time		**5.45 ms**

Seek time and rotational delay dominate!

Sequential vs. Random Access

What about accessing 1000 blocks of size 8 KB

random: $1000 \cdot 5.45\,\text{ms} = 5.45\,\text{s}$

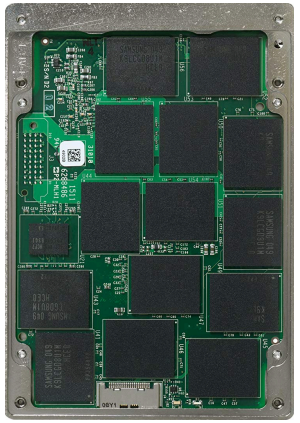
sequential: $3.4\,\text{ms} + 2\,\text{ms} + 1000 \cdot 0.05\,\text{ms} \approx 55\,\text{ms}$

tracks store only 512KB $\Longrightarrow$ some additional ($< 5\,\text{ms}$) track-to-track seek time

Sequential I/0 orders of magnitude faster than random I/O

avoid random I/O at all cost

What About Solid State Drives?



Alternative to conventional hard disks

- Low latency for read access (~50 µs), writes are more expensive
- Data is still accessed in blocks, blocks are organised in pages

Random access **still slower** than sequential access

Limitations

- Price per GB: SSD is ~5x **more expensive** than HDD
- Limited endurance: 100K – 1M write cycles

Most of the things discussed in this course are applicable to SSDs

Database Storage

The DBMS stores a database as one or more **files** on disk

- Files consist of **pages** (loaded in memory), pages contain **records**

The OS does not know anything about these files

- Only the DBMS knows how to decipher their contents
- The standard filesystem protections are used
- Early DBMSs in the 1980s used custom ‘filesystems’ on raw storage

System Design Goals

Goal: allow the DBMS to manage databases > available memory

Disk reads/writes are expensive ⟹ must be managed carefully

Spatial control

Where to write pages on disk

Goal: keep pages often used together as physically close as possible on disk

Temporal control

When to read pages into memory and when to write them to disk

Goal: minimise the number of stalls from having to read data from disk

Disk Space Manager

Hides details of the underlying storage (raw storage or OS files)

Introduces the concept of a **page** as a unit of storage

- Typical page size: 4 – 64KB (a multiple of 4KB)
- Each page has a unique identifier: **page ID**

Higher levels call upon this layer to:

- allocate/de-allocate a page
- read/write a page

Disk Space Manager (Cont.)

Maintains a **mapping** from page IDs to physical locations

(typical) physical location = filename + offset within that file

Keeps track of used and free blocks

1. Maintains a linked list of free pages, or
2. Maintains a bitmap reserving one bit for each page

Which one would you choose to support this?

Aims to allocate **contiguous sequences of pages** for data frequently accessed in **sequential order**

Higher levels do not need to know if / how this is done

Buffer Manager

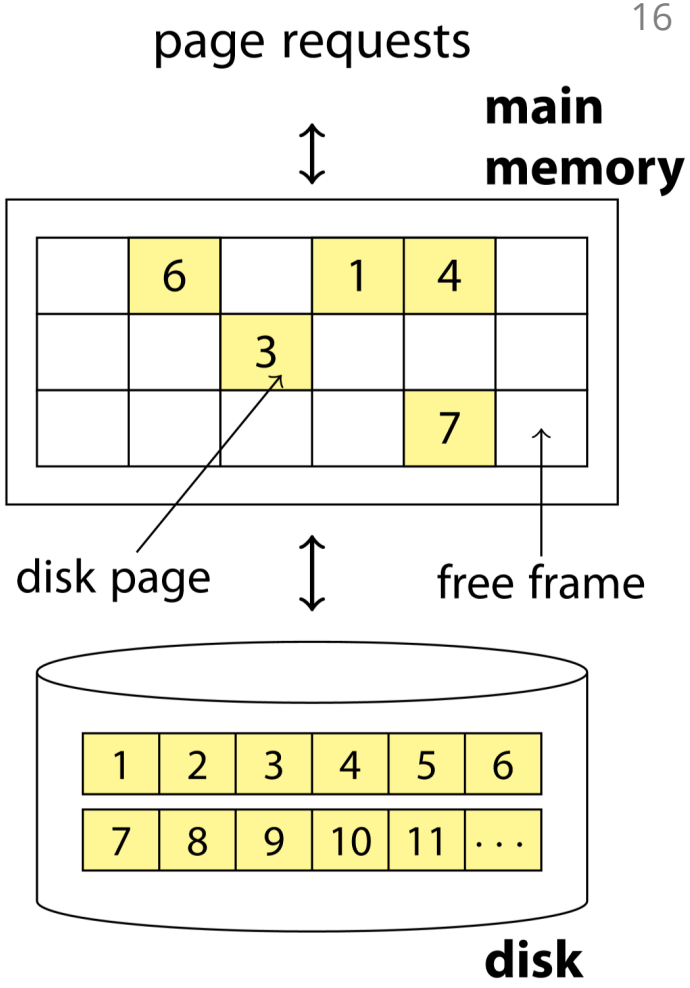
Brings pages from disk to memory and back, when needed

Higher levels need not to worry about whether data is in memory or not

Manages a designated main memory area – the **buffer pool** – for this task

Uses a **replacement policy** to decide which pages to evict when the buffer is full

Buffer Manager



Buffer pool: in-memory cache of disk pages

- Partitioned into **frames**, each frame can hold a page

Higher-level code can

- request (**pin**) a page from the buffer manager
- release (**unpin**) a page

Metadata per frame

Dirty flag	Indicates if the page is modified or not. Set during page release
Pin counter	# of concurrent users of the page if **pin counter = 0**, the page is a candidate for replacement

Proper Pin/Unpin Nesting

Database transactions must properly “bracket” any page operation using pin and unpin

A read-only transaction

```
a = pin(pageno)
 ⎡ .
 ⎢ .
 ⎢ .   read data on page at memory address a
 ⎢ .
 ⎣ .
unpin(pageno, false)
```

Proper bracketing useful to keep a count of active users (e.g., transactions) of a page

Pin Implementation

Function pin(*pageno*)

```
if buffer pool already contains pageno then
    f = find frame containing pageno
    f.pinCount = f.pinCount + 1
    return address of frame f
else
    f = select a free frame if buffer is not full or
                a victim frame using the replacement policy
    if f.isDirty then
        write frame f to disk
    read page pageno from disk into frame f
    f.pinCount = 1
    f.isDirty = false
    return address of frame f
```

Invariant:
f.pinCount = 0

Unpin Implementation

Function unpin(*pageno, dirty*)

```
f = find frame containing pageno
f.pinCount = f.pinCount - 1
f.isDirty = f.isDirty || dirty
```

Why don’t we check if *pageno* is in the buffer pool ?

Why don’t we write back to disk during unpin?

Concurrent Writes?

Conflicting page changes are possible

1. The same page *p* is requested by more than one transaction (i.e., pin counter of *p* > 1)
2. Those transactions perform conflicting writes on *p*

Such conflicts are resolved at a higher level (concurrency control) before the page is pinned

Lecture #11

The buffer manager may assume everything is in order whenever it receives an **unpin(p, true)** call

Buffer Replacement Policies

The effectiveness of a buffer pool depends on the used **replacement policy** and **access patterns** in high-level code

Least Recently Used (LRU)
Evict the page whose latest unpin is longest ago

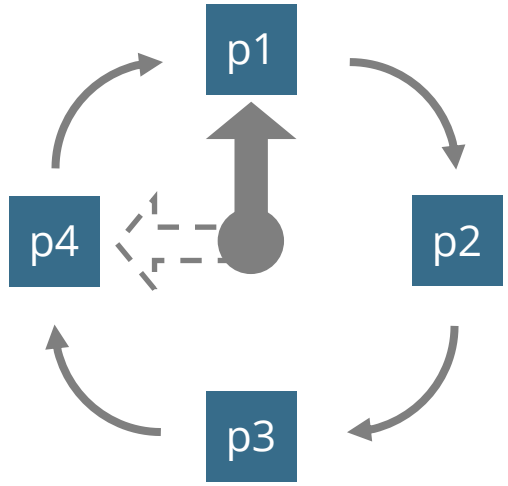
Most Recently Used (MRU)
Evict the page that has been unpinned most recently

Clock
Approximation of LRU without needing a separate timestamp per page

Others: **Random, Toss-Immediate, FIFO, LRU-K**

No single policy handles all possible scenarios well

Clock Replacement Policy



Each frame has a **reference bit**

Set referenced = 1 when pinCount goes to 0

N frames arranged in a circular buffer with a “clock hand”

```
while victim is not found:
    if frames[hand].pinCount == 0 then
        if frames[hand].referenced == 1 then
            frames[hand].referenced = 0
        else
            victim = address of frames[hand]
    hand = (hand + 1) mod N
```

Invoked when the pool is full and we need to evict a page

Replacement Policies Can Fail

LRU and CLOCK are susceptible to **sequential flooding**

Scans pollute the buffer with pages that might not be needed in the near future

For scans the most recently used page is actually the most unneeded page!

Example 1

A buffer pool consists of **10 pages**. A query repeatedly scans relation R.

Case 1: Let the size of relation R be **10 pages**. How many I/O do you expect?

Case 2: Now let the size of relation R be **11 pages**. How many I/O do you expect?

Buffer Management in Practice

Page fixing/hating

Higher-level code may request to **fix** a page if it may be useful in the near future

e.g., nested-loop joins

Likewise, an operator that **hates** a page will not access it any time soon

e.g., table pages in a sequential scan

Page prefetching

Speculative prefetching: assume sequential scan and automatically read ahead

Prefetch lists: higher-level code can instruct the buffer manager which pages to prefetch

Partitioned buffer pools

Separate pools for tables, indexes, logs, etc.

Why Not Use the OS?

Wait! Disk space and buffer management very much look like file management and virtual memory in the OS

But, the DBMS knows the query plan and access patterns of certain operators, and can make better decisions

- Flushing dirty pages to disk in correct order
- Specialized prefetching
- Buffer replacement policy
- Concurrency control

The OS is **not** your friend!

Conclusion

Magnetic Disk and Flash Storage

- Random access much slower than sequential access
- Physical data placement is important

Disk Space Manager

- Hides storage details, introduces pages
- Exposes data as a collection of pages

Buffer Manager

- Mediator between storage and main memory
- The DBMS can manage memory better than the OS