# Slide 1

THE UNIVERSITY of EDINBURGH
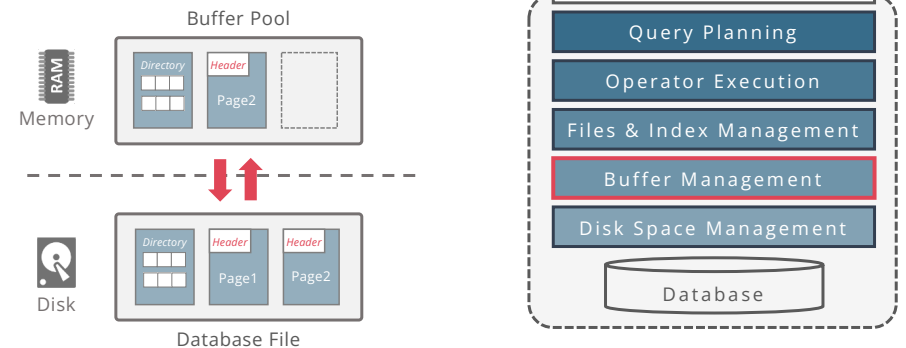
## Advanced Database Systems
Spring 2026

Lecture #05:
## Buffer Management

R&G: Chapter 9.4

1

# Slide 2

## BUFFER MANAGEMENT

**Transfer** data between disk and memory

Buffer Pool

Memory

Directory | Header Page2

Disk

Directory | Header Page1 | Header Page2

Database File

SQL Client
Query Planning
Operator Execution
Files & Index Management
Buffer Management
Disk Space Management
Database

2

# Slide 3

## BUFFER MANAGEMENT

**Buffer pool**: in-memory cache of disk pages, partitioned into **frames**

Buffer Pool

Memory

Directory | Header Page2 | Empty Frame

Disk

Directory | Header Page1 | Header Page2

Database File

Each frame can hold a page

Higher level code can

request (**pin**) a page

release (**unpin**) a page

3

# Slide 4
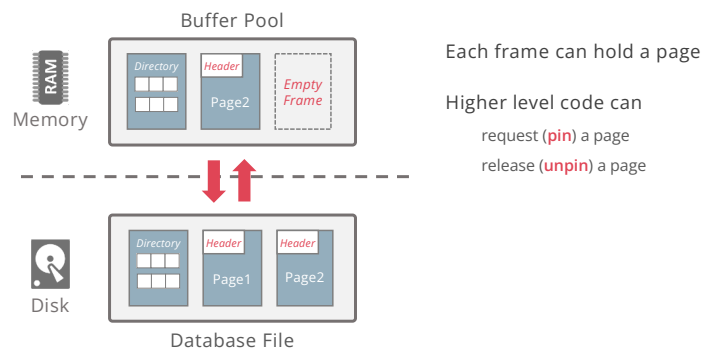
## BUFFER MANAGEMENT: PAGE REQUEST

**Ensures** requested page is in memory upon return

Buffer Pool

Memory

Directory | Header Page2 | Header Page1

Disk

Directory | Header Page1 | Header Page2

Database File

*Get page #1*

*Pointer to page #1*

Execution Engine

The illusion of operating only in memory

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

4

# BUFFER MANAGEMENT: PAGE RELEASE

Higher levels need to explicitly "release" a page



Buffer Pool

Done with page#2

OK

Execution Engine

Memory

Disk

Database File

A page can be simultaneously used by multiple users

If **nobody** is using a page at the moment, only then that page can be removed from the pool

But doesn't have to be removed immediately

---

# OPEN QUESTIONS

What if the buffer pool has no space for a new page?

Use a **replacement policy** to decide which page to evict

What if a page gets modified? How will the buffer manager find out?

**Dirty flag** on page: Is page modified or not, set during release by higher levels

When evicting a dirty page, write it back to disk via disk space manager

How many users are concurrently using a page?

**Pin counter** per frame: # of concurrent users of the page

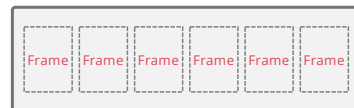If **pin counter = 0**, the page is a candidate for replacement

---

# BUFFER MANAGER STATE

**Buffer pool**

Large range of memory allocated at DBMS server boot time (MBs-GBs)



| Frame | Frame | Frame | Frame | Frame | Frame |

**Buffer manager metadata**:

Smallish array in memory allocated at DBMS server boot time

Page ID lookups need to be fast

Keep an in-memory index (hash table) on PageId

| FrameId | PageId | Dirty? | Pin Count |
|---------|--------|--------|-----------|
| 1 | 1 | N | 0 |
| 2 | 2 | Y | 1 |
| 3 | 8 | N | 0 |
| 4 | 6 | N | 2 |
| 5 | 4 | N | 0 |
| 6 | 5 | N | 0 |

---

# PROPER PIN/UNPIN NESTING

Database users (e.g., 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 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

9

---

# 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?

10

---

# ADVANCED QUESTIONS

Concurrent operations on a page

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*

Solved by **Concurrency Control** module

… before the page is unpinned

Buffer manager may assume everything is in order whenever it gets an **unpin(p, true)** call

What if system crashes before write-back?

Solved by **Recovery** module

More about CC & Recovery later

11

---

# BUFFER REPLACEMENT POLICIES

Page is chosen for replacement by a **replacement policy**:

Least Recently Used (LRU), Clock

Most Recently Used (MRU)

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

Policy can have big impact on #I/Os

Effectiveness depends on the **access patterns** in high-level code

No single policy handles all possible scenarios well

12

# LEAST RECENTLY USED (LRU)

Very common policy: intuitive and simple

Track time each frame was last unpinned (end of use)

Replace the frame which was least recently used (lowest last used time)

Pinned frames are not available to replace

| FrameId | PageId | Dirty? | Pin Count | Last Used |
|---------|--------|--------|-----------|-----------|
| 1 | 1 | N | 0 | 43 |
| 2 | 2 | Y | 1 | 21 |
| 3 | 8 | N | 0 | 22 |
| 4 | 6 | N | 2 | 11 |
| 5 | 4 | N | 0 | 24 |
| 6 | 5 | N | 0 | 15 |

← Pinned frames (rows 2 and 4)

← Next-to-replace frame (row 6)

---

# LEAST RECENTLY USED (LRU)

Good for repeated accesses to popular pages (temporal locality)

Unpopular pages accessed a while ago are more likely to be replaced

Can be costly. Why?

Need to "find min" on the last used attribute

**Naive:** Scan table to find the unpinned frame with the lowest last used time (linear time)

**Better:** Use priority queues to keep frames in sorted order (log time)

Priority queues can still be expensive as page accesses are frequent

Approximate LRU: CLOCK policy

---

# CLOCK REPLACEMENT POLICY

Each frame has a **reference bit**

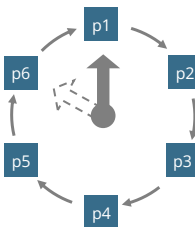Set referenced = 1 when pin count increases

N frames arranged in a circular buffer with a "clock hand"

Clock hand = next page to consider for eviction

```
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
```

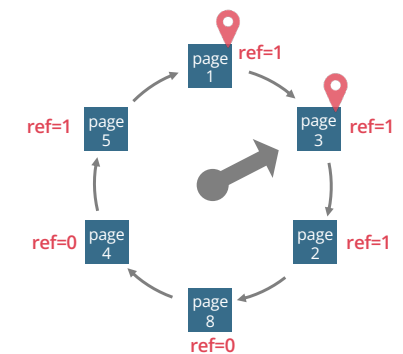Invoked when the pool is full and we need to evict a page



---

# CLOCK POLICY STATE: EXPLICIT & ILLUSTRATED

| FrameId | PageId | Dirty? | Pin Count | Ref Bit |
|---------|--------|--------|-----------|---------|
| 1 | 1 | N | 1 | 1 |
| 2 | 3 | N | 1 | 1 |
| 3 | 2 | N | 0 | 1 |
| 4 | 8 | N | 0 | 0 |
| 5 | 4 | N | 0 | 0 |
| 6 | 5 | N | 0 | 1 |

| Clock Hand |
|------------|
| 2 |

## Clock Policy: Read Page 10
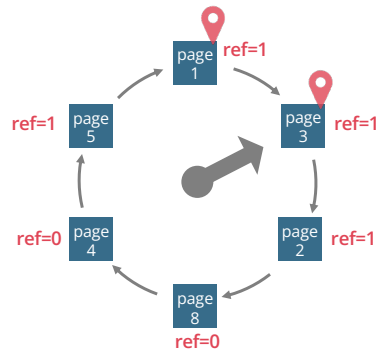
The current buffer state is on the right

A read request for page 10 arrives

The buffer pool has no page 10 and is full

The buffer manager needs to evict one page. Which one?
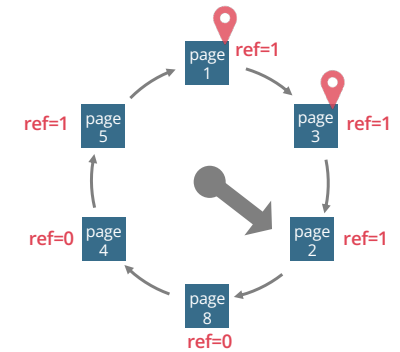
Current frame has pin count > 0

Action: **Skip**



18

---

## Clock Policy: Read Page 10 (Cont.)

Current frame not pinned

Reference bit set
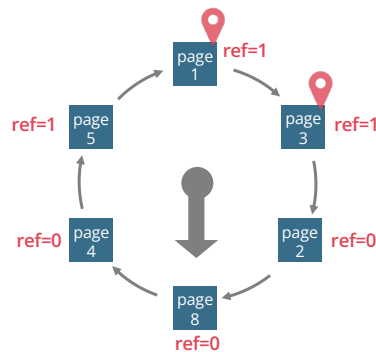
Clear reference bit

Skip



19

---

## Clock Policy: Read Page 10 (Cont.)

Current frame not pinned

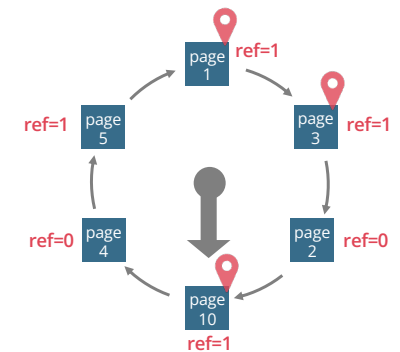Reference bit unset

Replace page 8 by page 10



20

---

## Clock Policy: Read Page 10 (Cont.)

Current frame not pinned

Reference bit unset

Replace page 8 by page 10

Set pinned

Set reference bit

Advance clock



21

# CLOCK POLICY: READ PAGE 10 (CONT.)

Current frame not pinned

Reference bit unset
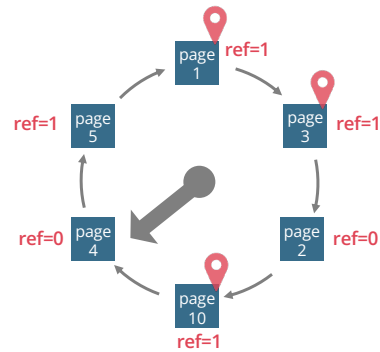
- Replace page 8 by page 10
- Set pinned
- Set reference bit
- Advance clock
- Return pointer to page 10



page 1 ref=1
page 3 ref=1
ref=0 page 2
page 10 ref=1
ref=0 page 4
ref=1 page 5

# REPLACEMENT POLICIES CAN FAIL

LRU and CLOCK are susceptible to sequential flooding

Scans pollute the buffer with pages that might not be needed soon

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

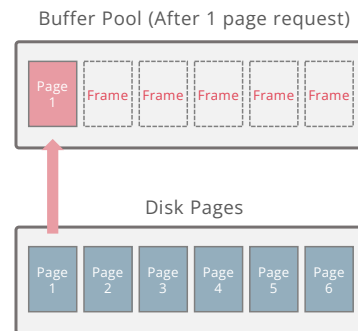| Example 1 |
| --- |
| A buffer pool consists of **6 frames**. A query repeatedly scans relation R. |
| **Case 1:** Let the size of relation R be **6 pages**. How many I/O do you expect? |
| **Case 2:** Now let the size of relation R be **7 pages**. How many I/O do you expect? |

# REPEATED SCAN (LRU)

The buffer pool consists of **6 frames**

Assume the frames are initially empty

Case 1: R consists of 6 pages

Buffer Pool (After 1 page request)



Page 1 | Frame | Frame | Frame | Frame | Frame

Disk Pages

Page 1 | Page 2 | Page 3 | Page 4 | Page 5 | Page 6
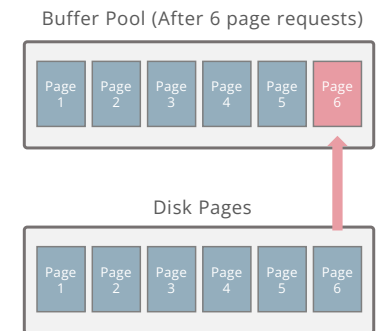
# REPEATED SCAN (LRU)

The buffer pool consists of **6 frames**

Assume the frames are initially empty

Case 1: R consists of 6 pages

First 6 requests are unavoidable misses

Subsequent requests P1-P6 are all **hits**!

Buffer Pool (After 6 page requests)



Page 1 | Page 2 | Page 3 | Page 4 | Page 5 | Page 6

Disk Pages

Page 1 | Page 2 | Page 3 | Page 4 | Page 5 | Page 6

# REPEATED SCAN (LRU)
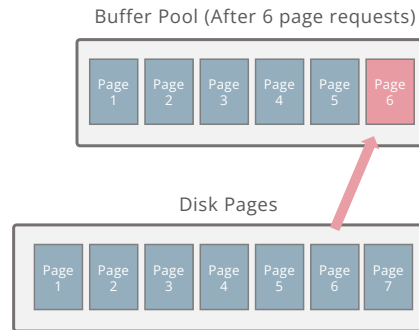
The buffer pool consists of **6 frames**

  Assume the frames are initially empty

Case 1: R consists of 6 pages

  First 6 requests are unavoidable misses

  Subsequent requests P1-P6 are all **hits**!

Case 2: R consists of 7 pages

Buffer Pool (After 6 page requests)

| Page 1 | Page 2 | Page 3 | Page 4 | Page 5 | Page 6 |

Disk Pages

| Page 1 | Page 2 | Page 3 | Page 4 | Page 5 | Page 6 | Page 7 |

---

# REPEATED SCAN (LRU)

The buffer pool consists of **6 frames**

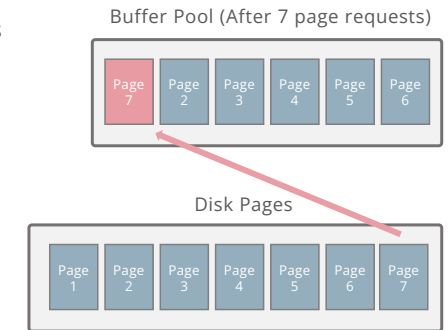  Assume the frames are initially empty

Case 1: R consists of 6 pages

  First 6 requests are unavoidable misses

  Subsequent requests P1-P6 are all **hits**!

Case 2: R consists of 7 pages

  P7 evicts P1, restart scan

Buffer Pool (After 7 page requests)

| Page 7 | Page 2 | Page 3 | Page 4 | Page 5 | Page 6 |

Disk Pages

| Page 1 | Page 2 | Page 3 | Page 4 | Page 5 | Page 6 | Page 7 |

---

# REPEATED SCAN (LRU)

The buffer pool consists of **6 frames**

  Assume the frames are initially empty

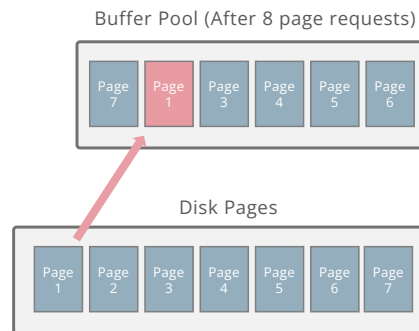Case 1: R consists of 6 pages

  First 6 requests are unavoidable misses

  Subsequent requests P1-P6 are all **hits**!

Case 2: R consists of 7 pages

  P7 evicts P1, restart scan,
  P1 evicts P2, P2 evicts P3, etc.
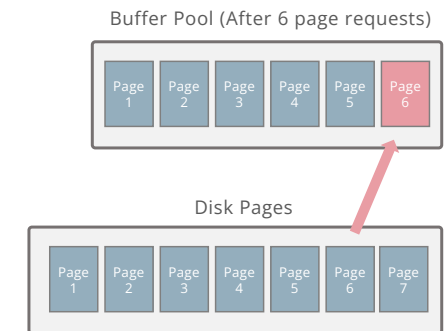
  All subsequent page requests are **misses**!

Buffer Pool (After 8 page requests)

| Page 7 | Page 1 | Page 3 | Page 4 | Page 5 | Page 6 |

Disk Pages

| Page 1 | Page 2 | Page 3 | Page 4 | Page 5 | Page 6 | Page 7 |

---

# REPEATED SCAN (MRU)

Most Recently Used (MRU)

  First 6 requests are unavoidable misses

Buffer Pool (After 6 page requests)

| Page 1 | Page 2 | Page 3 | Page 4 | Page 5 | Page 6 |

Disk Pages

| Page 1 | Page 2 | Page 3 | Page 4 | Page 5 | Page 6 | Page 7 |

# REPEATED SCAN (MRU)
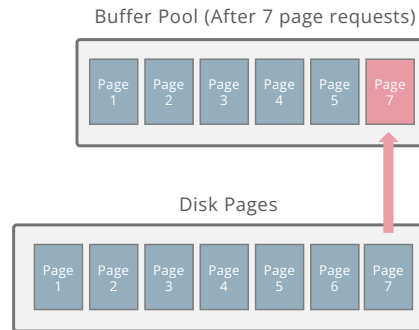
Most Recently Used (MRU)

First 6 requests are unavoidable misses

Request for P7 evicts P6

After restart, P1-P5 requests are all hits!

Buffer Pool (After 7 page requests)

| Page 1 | Page 2 | Page 3 | Page 4 | Page 5 | Page 7 |

Disk Pages

| Page 1 | Page 2 | Page 3 | Page 4 | Page 5 | Page 6 | Page 7 |

---

# REPEATED SCAN (MRU)
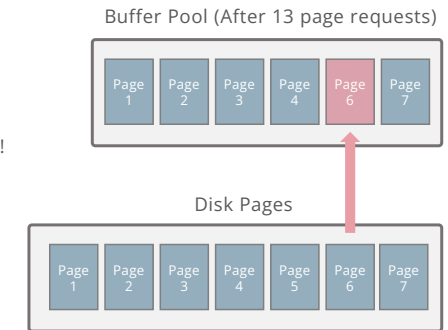
Most Recently Used (MRU)

First 6 requests are unavoidable misses

Request for P7 evicts P6

After restart, P1-P5 requests are all hits!

Request for P6 evicts P5

Next 5 requests are also hits!

and so on…

Buffer Pool (After 13 page requests)

| Page 1 | Page 2 | Page 3 | Page 4 | Page 6 | Page 7 |

Disk Pages

| Page 1 | Page 2 | Page 3 | Page 4 | Page 5 | Page 6 | Page 7 |

---

# BEST REPLACEMENT POLICY?

LRU suffers from sequential flooding

But good for random access (hot vs. cold data)

**LRU-K** variant:

Consider history of the last K references

Evict the page whose K-th most recent access is furthest away in the past

MRU better fit for repeated sequential scans

Repeated scans are very common in database workloads (e.g., nested-loops join)

Hybrids are not uncommon in modern DBMSs

PostgreSQL uses CLOCK but handles sequential scans separately

---

# BUFFER MANAGEMENT IN PRACTICE

**Priority hints**

The DBMS knows the context of each page during query execution

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

Page fixing & hating:

Request to **fix** a page as it may be useful soon (e.g., nested-loop joins)

Request to **hate** a page as it may not be accessed soon (e.g., pages in a sequential scan)

**Partitioned buffer pools**

Separate pools for tables, indexes, logs, etc.

# BUFFER MANAGEMENT IN PRACTICE

**Page Prefetching**

Ask disk space manager for a run of sequential pages

E.g., on request for Page 1, ask for Pages 2-5

Why does this help?

Amortise random I/O overhead

Allow computation while I/O continues in background
(disk and CPU are "parallel devices")

# WHY NOT USE THE OS?

**Wait!** Doesn't the filesystem (OS) manage buffers and pages too?

**Yes, but:**

DBMS requires ability to force flushing pages to disk in correct order

Required for recovery, as discussed later

DBMS has more information about query plans and access patterns of operators

Affects both page replacement and prefetching

Portability: different filesystem, different behaviour

The OS is **not** your friend!

# SUMMARY

Buffer Manager

Mediator between storage and main memory

Maps disk page IDs to RAM addresses

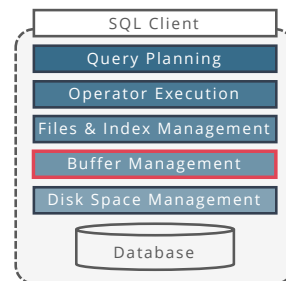Ensures each requested page is "pinned" in RAM

To be (briefly) manipulated in-memory

And then unpinned by the caller!

Attempts to minimize "cache misses"

By replacing pages unlikely to be referenced

By prefetching pages likely to be referenced

| SQL Client |
| --- |
| Query Planning |
| Operator Execution |
| Files & Index Management |
| Buffer Management |
| Disk Space Management |
| Database |