Advanced Database Systems

Spring 2024

Lecture #11:

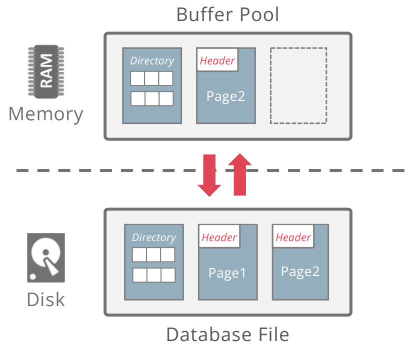
Buffer Management

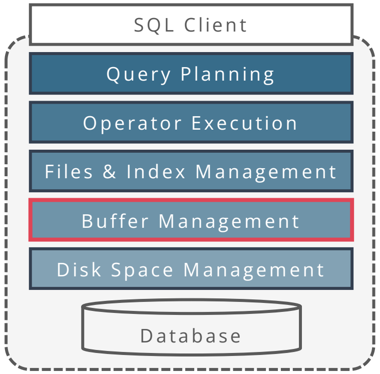
R&G: Chapter 9.4

1

Buffer Management

Transfer data between disk and memory

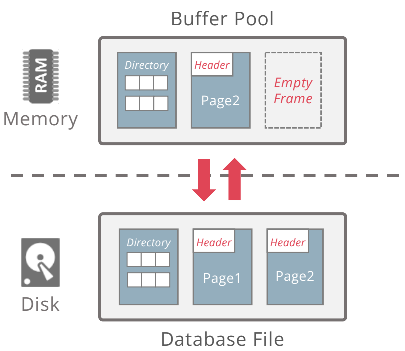




2

Buffer Management

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



Each frame can hold a page

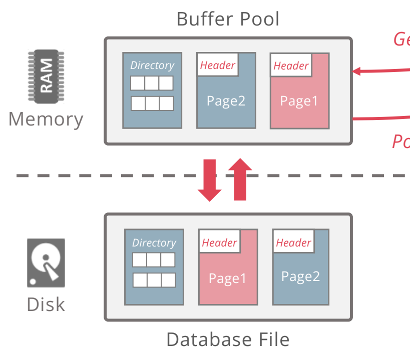
Higher level code can

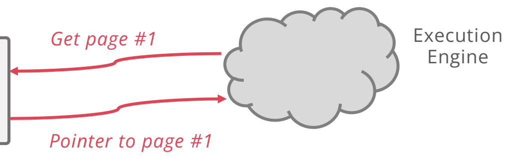
- request (**pin**) a page
- release (**unpin**) a page

3

Buffer Management: Page Request

Ensures requested page is in memory upon return





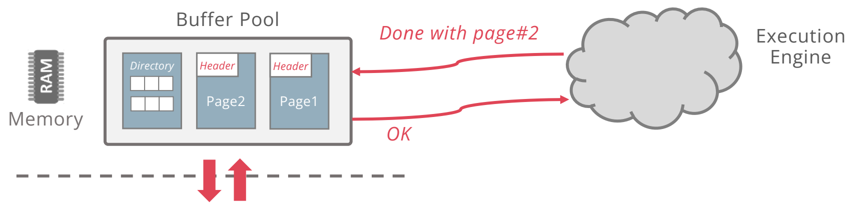
The illusion of operating only in memory

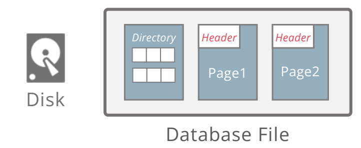
Higher levels need not to worry about whether data is in memory or not

4

Buffer Management: Page Release

Higher levels need to explicitly "release" a page





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?

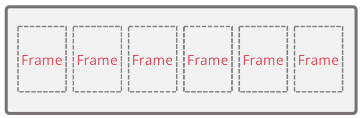
Pin counter per frame: # of concurrent users of the page

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)



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

if unpin is not given then the page is never evicted causing memory leak

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)
```

whether page has changed

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

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 ? atleast one request would use it

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

high possibility of future requests needing the page and writing back to disk is an expensive operation

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

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

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

Next-to-replace frame

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

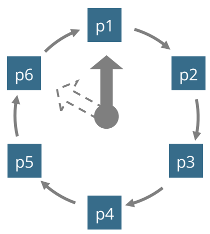
Can be costly. Why?

- 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

log time is not sufficient and would cause bottleneck as eviction is very frequent in dbs

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

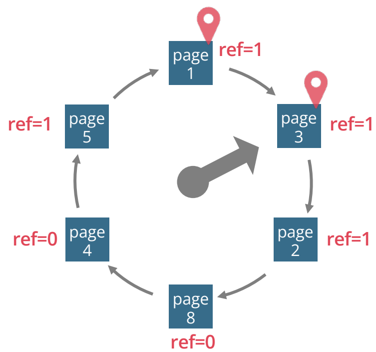
```
while victim is not found:
    if frames[hand].pinCount == 0 then
        if frames[hand].referenced == 1 then
            frames[hand].referenced = 0
        else
            victim = address of frames[hand]
    hand = (hand + 1) mod N
```

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

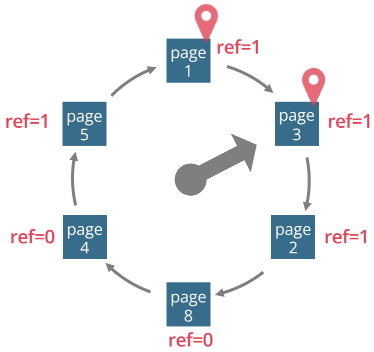
A read request for page 10 arrives

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

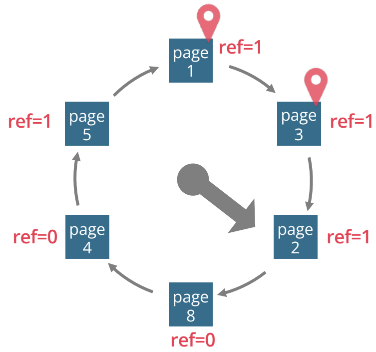


Clock Policy: Read Page 10 (Cont.)

Current frame not pinned

Reference bit set

- Clear reference bit
- Skip

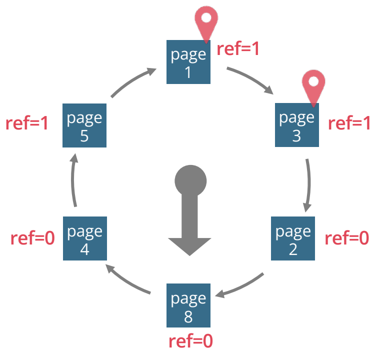


Clock Policy: Read Page 10 (Cont.)

Current frame not pinned

Reference bit unset

- Replace page 8 by page 10

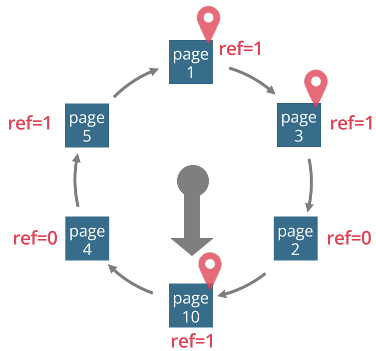


Clock Policy: Read Page 10 (Cont.)

Current frame not pinned

Reference bit unset

- Replace page 8 by page 10
- Set pinned
- Set reference bit
- Advance clock

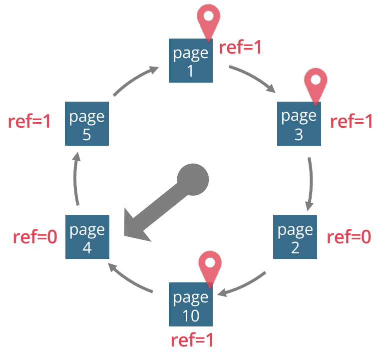


Clock Policy: Read Page 10 (Cont.)

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



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?

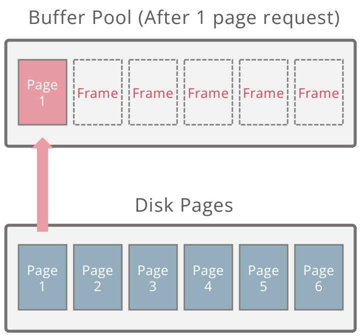
Case 2: Now let the size of relation R be **7 pages**. How many I/O do you expect?

Repeated Scan (LRU)

The buffer pool consists of **6 frames**

- Assume the frames are initially empty

Case 1: R consists of 6 pages



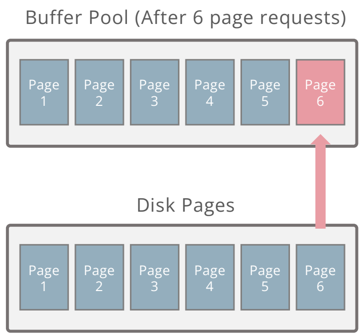
Repeated Scan (LRU)

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!



REPEATED SCAN (LRU)

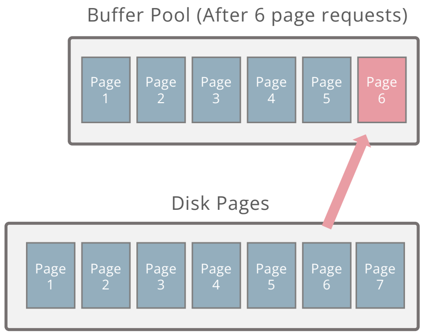
The buffer pool consists of **6 frames**

- 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



REPEATED SCAN (LRU)

The buffer pool consists of **6 frames**

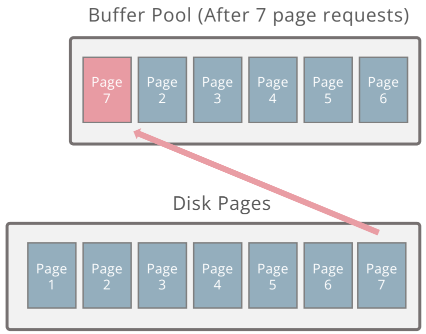
- Assume the frames are initially empty

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



REPEATED SCAN (LRU)

The buffer pool consists of **6 frames**

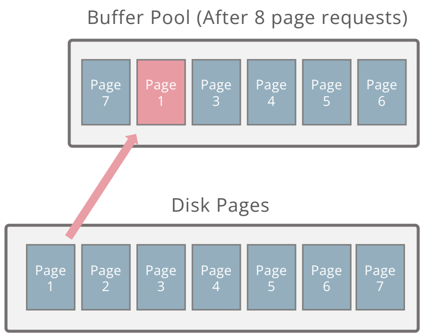
- Assume the frames are initially empty

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

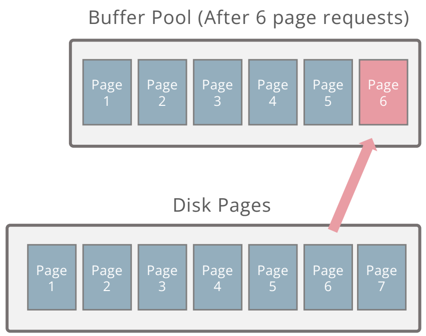
- P7 evicts P1, restart scan,
  P1 evicts P2, P2 evicts P3, etc.
- All subsequent page requests are **misses**!



REPEATED SCAN (MRU)

Most Recently Used (MRU)

- First 6 requests are unavoidable misses

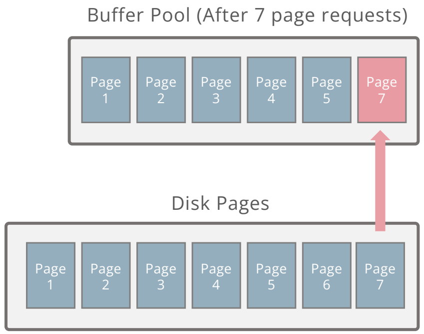


In Case 2, MRU can perform better than Least Recently Used (LRU) in certain scenarios. In this example, all scans are a miss, however P1-P5 request scans are hits in MRU

Repeated Scan (MRU)

Most Recently Used (MRU)

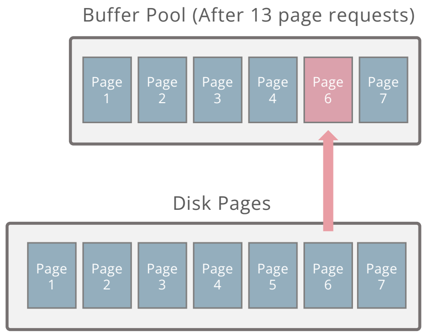
- First 6 requests are unavoidable misses
- Request for P7 evicts P6
- After restart, P1-P5 requests are all **hits**!



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…



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