CS150A Database

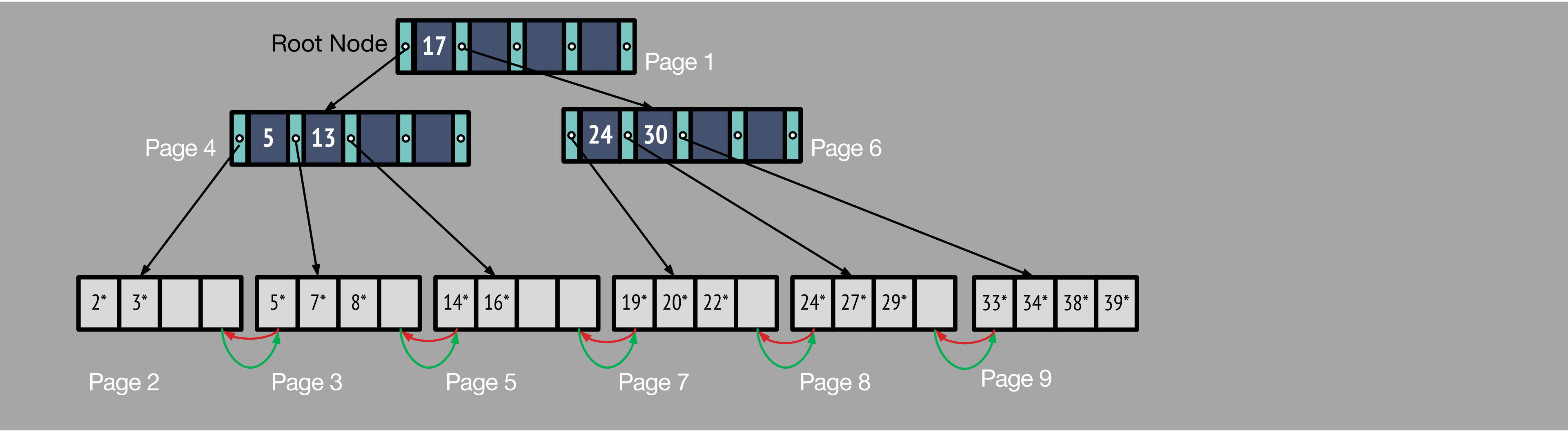
Wenjie Wang
School of Information Science and Technology
ShanghaiTech University
Oct. 14, 2024

Today:
- Buffer Manager:
  - Dirty Pages Handling
  - Page Replacement Policies

Readings:
- Database Management Systems (DBMS), Chapter 9.4

Acknowledgement: Joe Hellerstein@UCBerkeley's scourse notes

Review



- Occupancy Invariant
  - Each interior node is at least partially full:
    - **d <= #entries <= 2d**
    - **d: order of the tree (max fan-out = 2d + 1)**
- Data pages at bottom need not be stored in logical order
  - Next and prev pointers

Review

- ISAM is a static structure
  - **Only leaf pages modified**; overflow pages needed
  - Overflow **chains can degrade performance** unless size of data set and data distribution stay constant

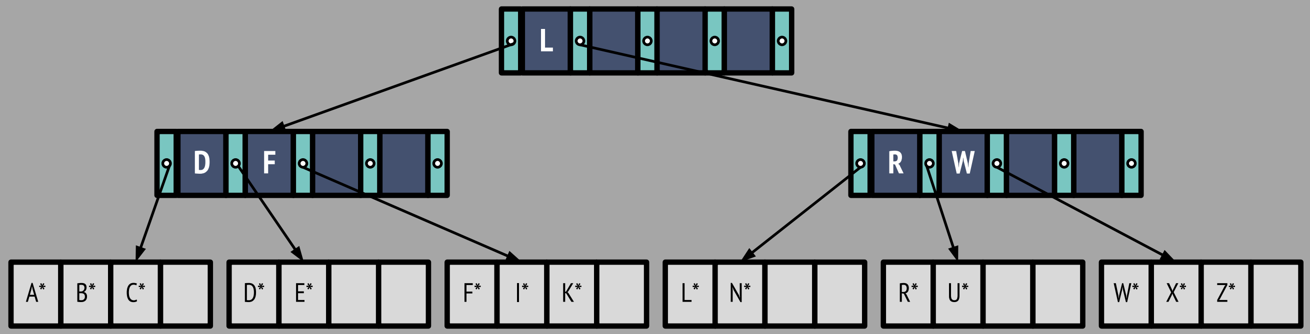
- **B+ Tree is a dynamic structure**
  - Inserts/deletes leave tree height-balanced; $\log_F N$ cost
  - High fanout (F) means depth rarely more than 3 or 4.
  - Almost always better than maintaining a sorted file.
  - Typically, 67% occupancy on average
  - Usually preferable to ISAM; adjusts to growth gracefully.

BULK LOADING B+-TREES

Bulk Loading of B+ Tree Part 1

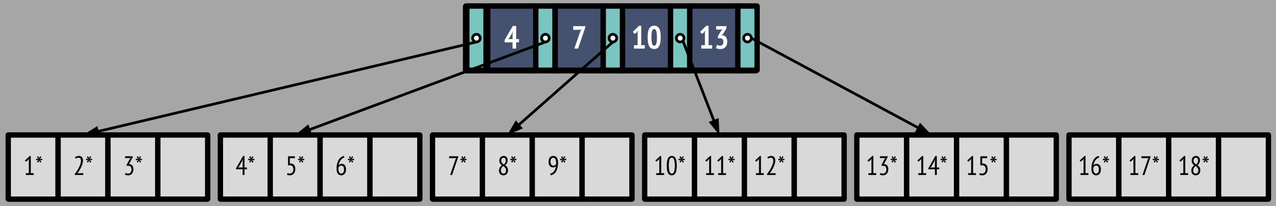
- Suppose we want to build an index on a large table
- Would it be efficient to just call insert repeatedly
  - No … Why not?
  - Random Order: CLZARNDXEKFWIUB. Order 2.
  - Try it: Interactive demo

Bulk Loading of B+ Tree Part 2



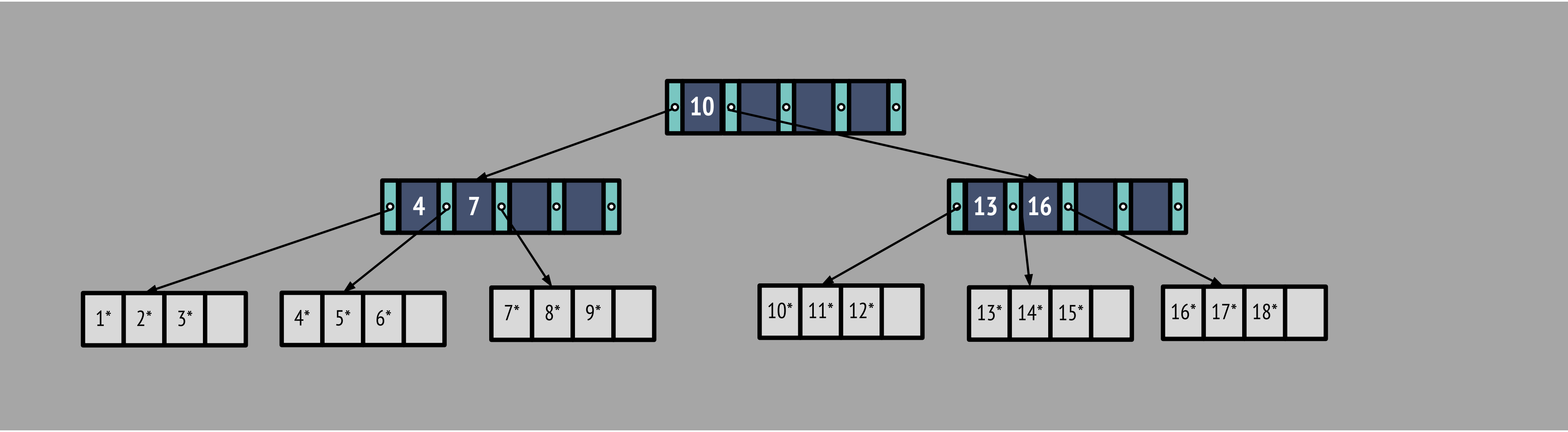
- Constantly need to search from root
- Leaves and internal nodes mostly half-empty
- **Modifying random pages:**
  poor cache efficiency

Smarter Bulk Loading a B+ Tree



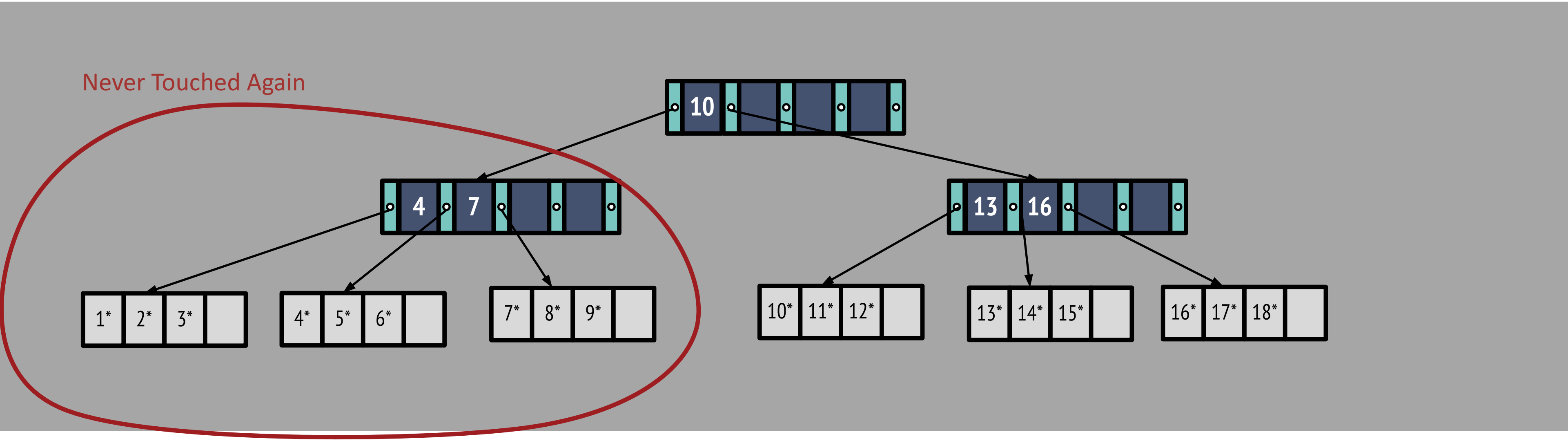
- Sort the input records by key:
  - 1*, 2*, 3*, 4*, …
  - We'll learn a good disk-based sort algorithm soon!
- Fill leaf pages to some fill factor (e.g. ¾)
  - Updating parent until full

Smarter Bulk Loading a B+ Tree Part 2



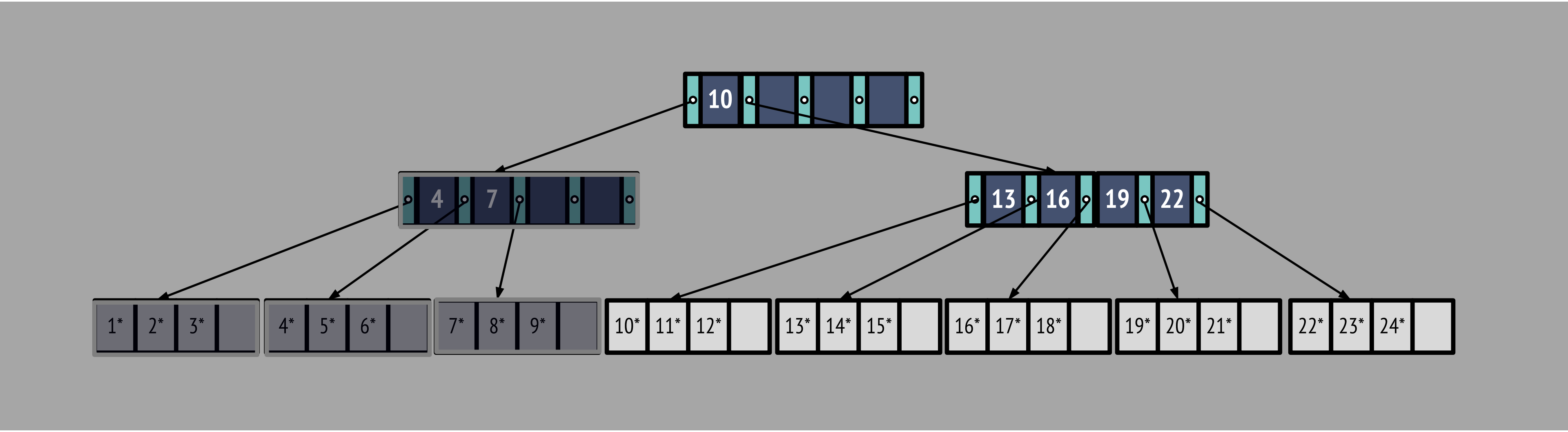
- Sort the input records by key:
  - 1*, 2*, 3*, 4*, …
- Fill leaf pages to some fill factor (e.g. ¾)
  - Update parent until full
  - Then split parent (50/50) and copy to sibling

Smarter Bulk Loading a B+ Tree Part 3



- Lower left part of the tree is never touched again
- Occupancy invariant maintained

Smarter Bulk Loading a B+ Tree Part 4



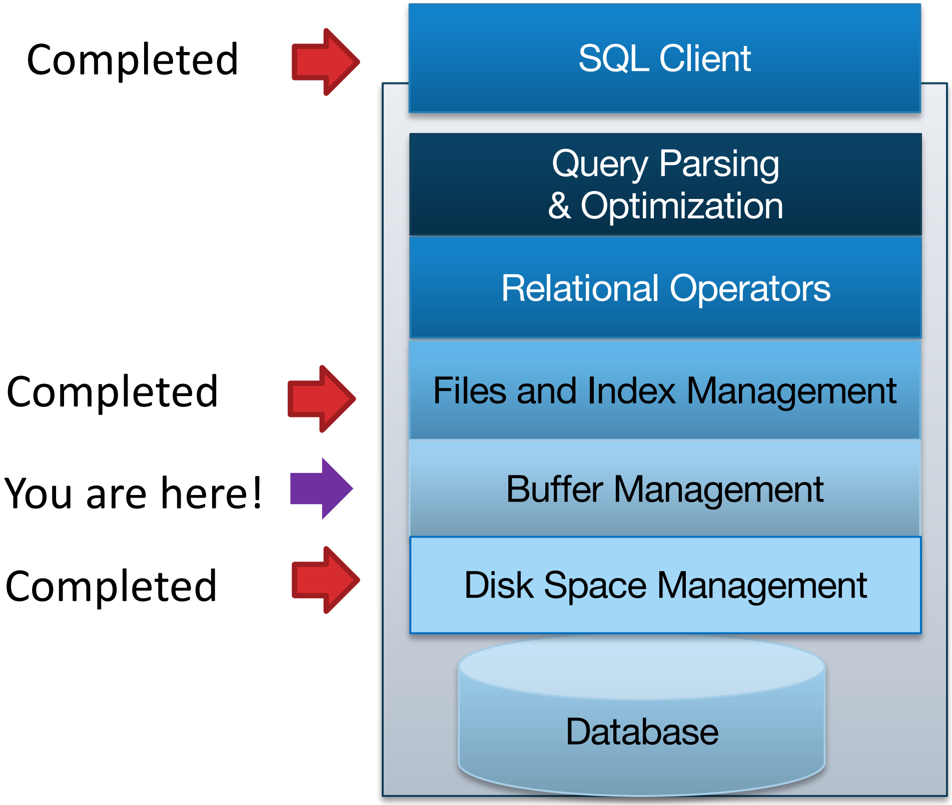
- Sort the input records by key:
  - 1*, 2*, 3*, 4*, …
- Fill leaf pages to some fill factor (e.g. ¾)
  - Update parent until full
  - Then split parent

Summary of Bulk Loading

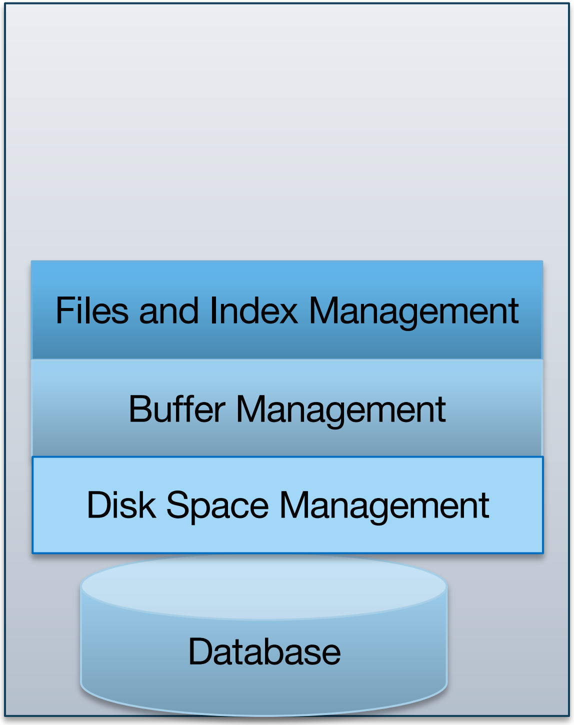
- Option 1: Multiple inserts
  - **Slow**
  - Does not give sequential storage of leaves

- Option 2: Bulk Loading
  - Leaves will be stored sequentially (and linked, of course)
  - Can control “fill factor” on pages.
  - Fewer I/Os during build. (Why?)

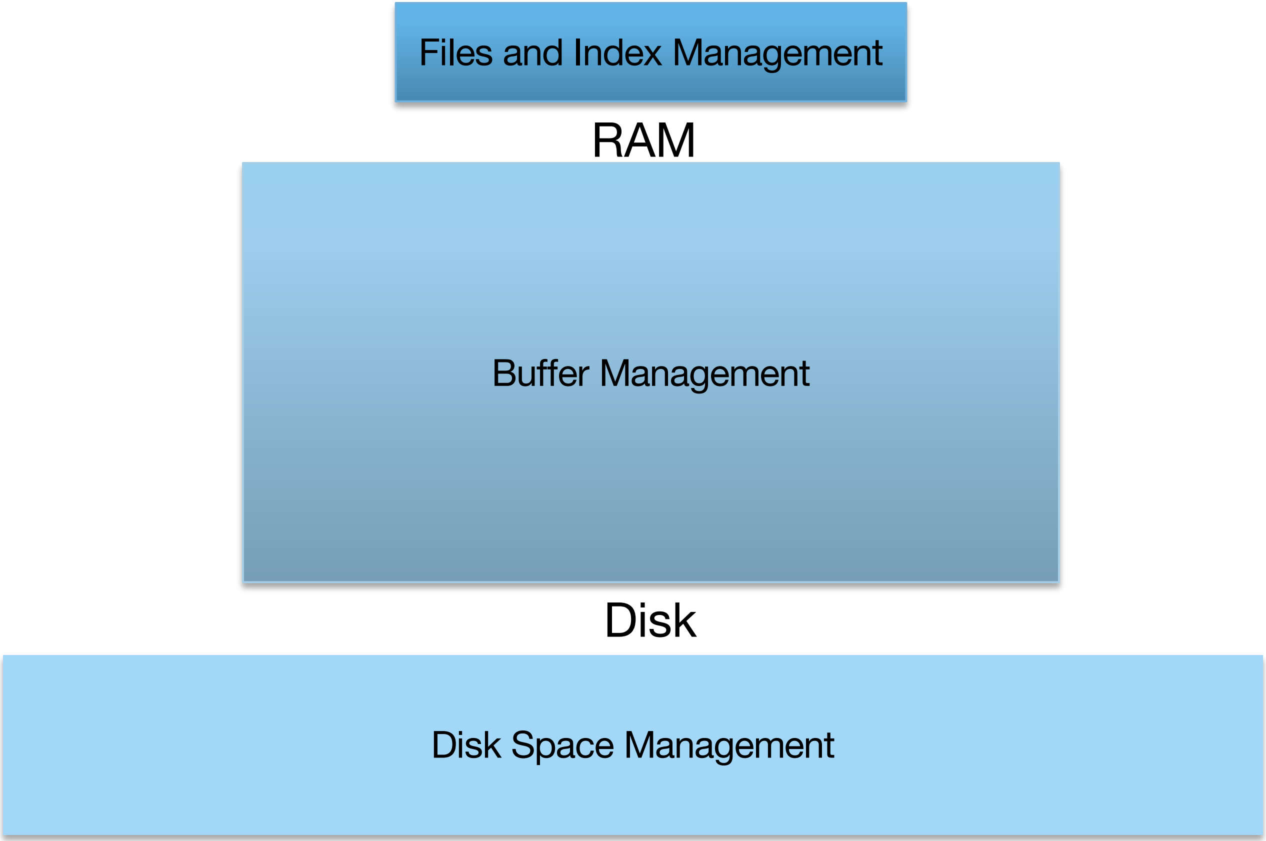
Architecture of a DBMS: What we've learned



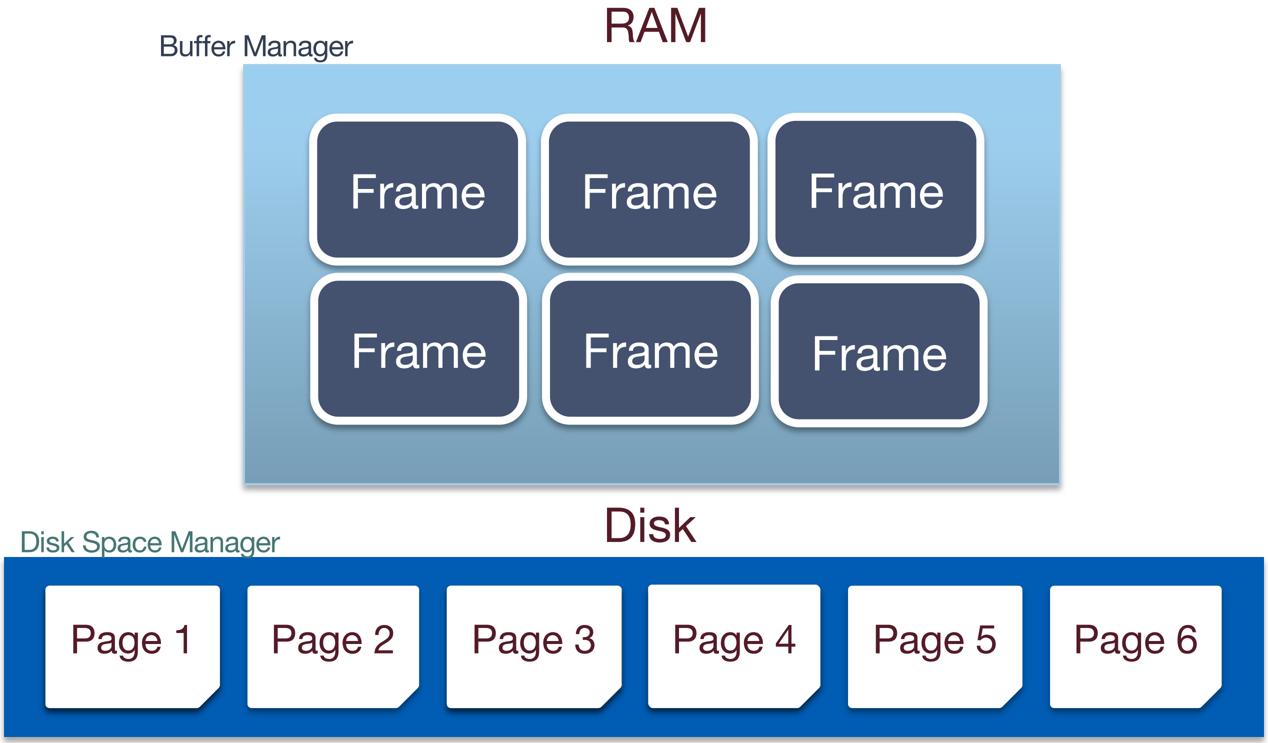
Lower Architecture of a DBMS



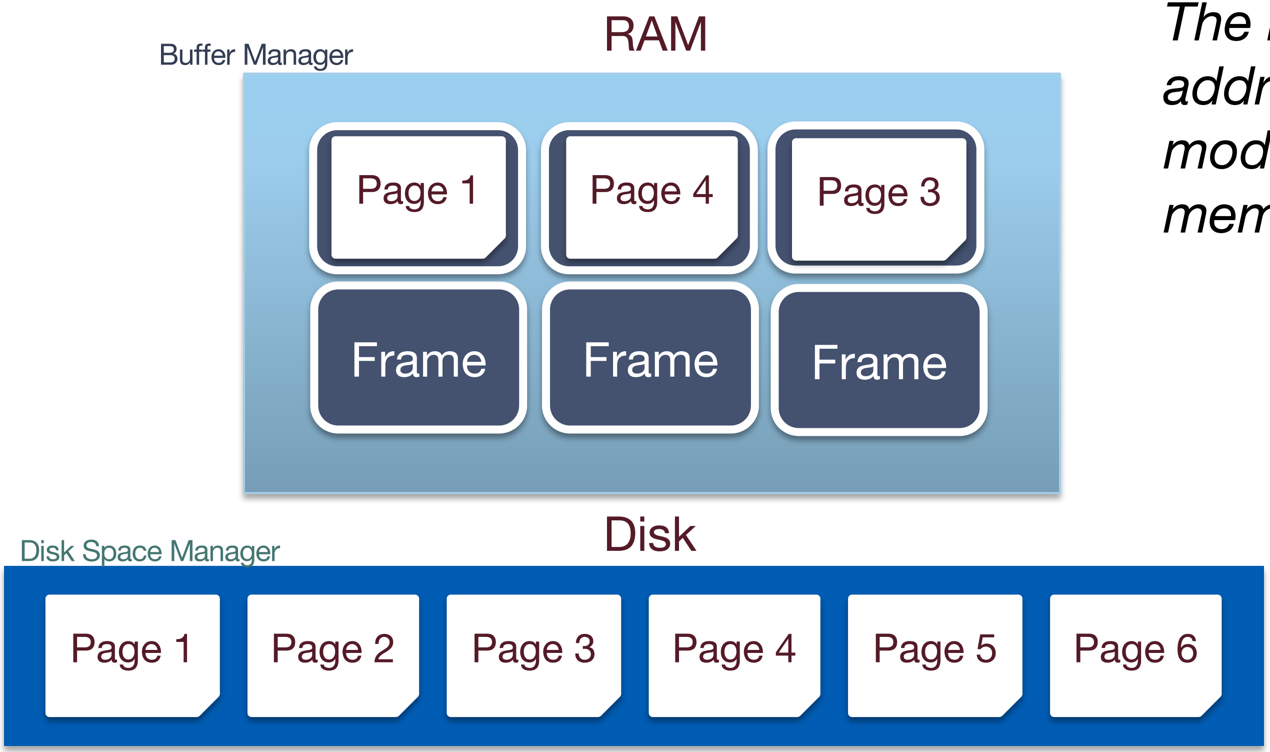
Buffer Management Levels of Abstraction



Buffer Management, cont

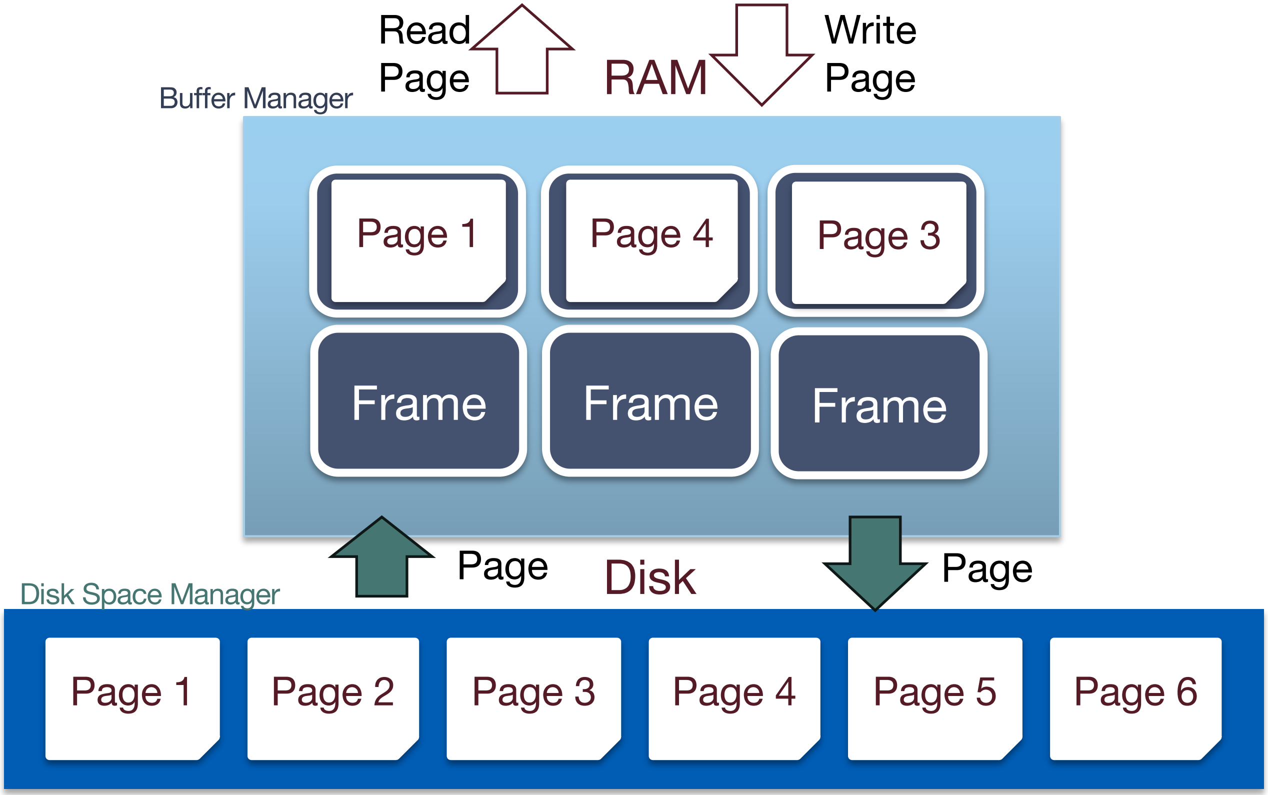


Buffer Management Read

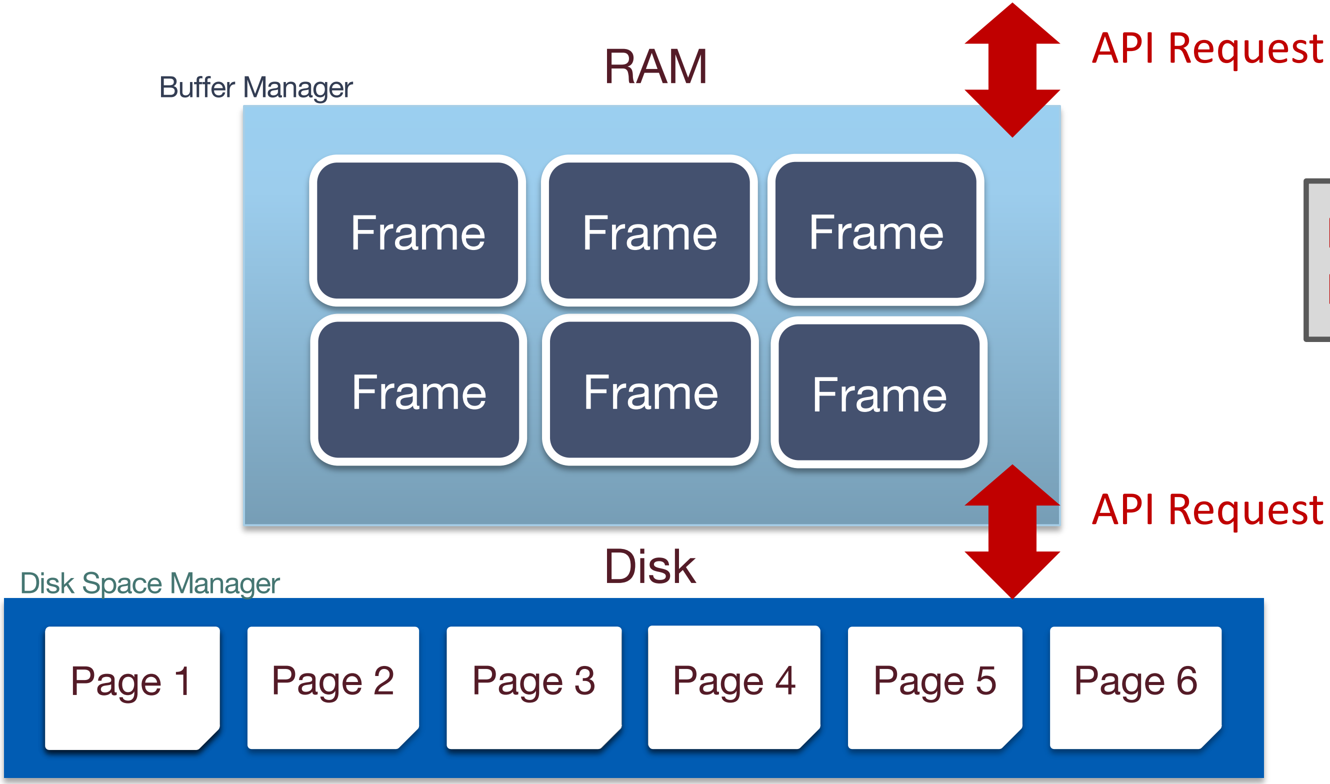


The illusion of addressing and modifying disk pages in memory.

APIs

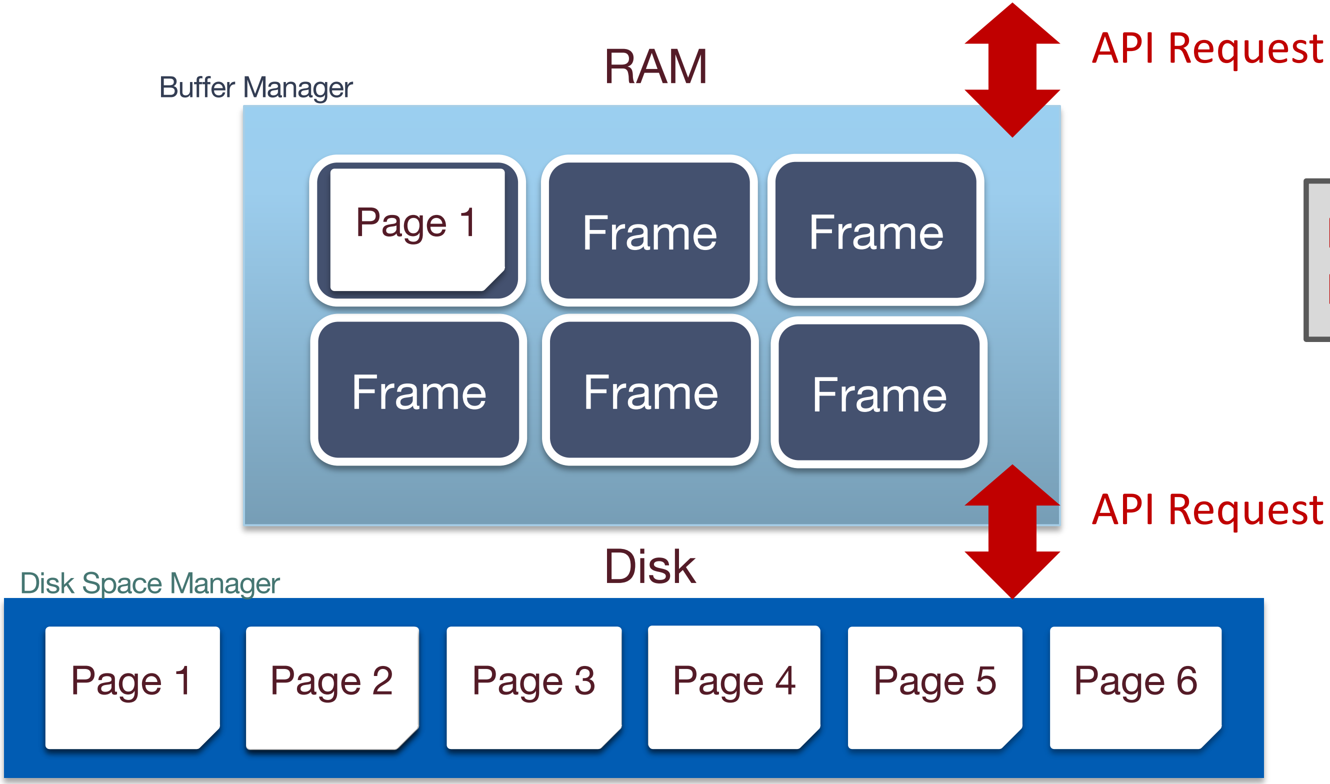

Read Page
RAM
Write Page
Buffer Manager
Page 1
Page 4
Page 3
Frame
Frame
Frame
Page
Disk
Page
Disk Space Manager
Page 1
Page 2
Page 3
Page 4
Page 5
Page 6

Mapping Pages Into Memory, Pt 1



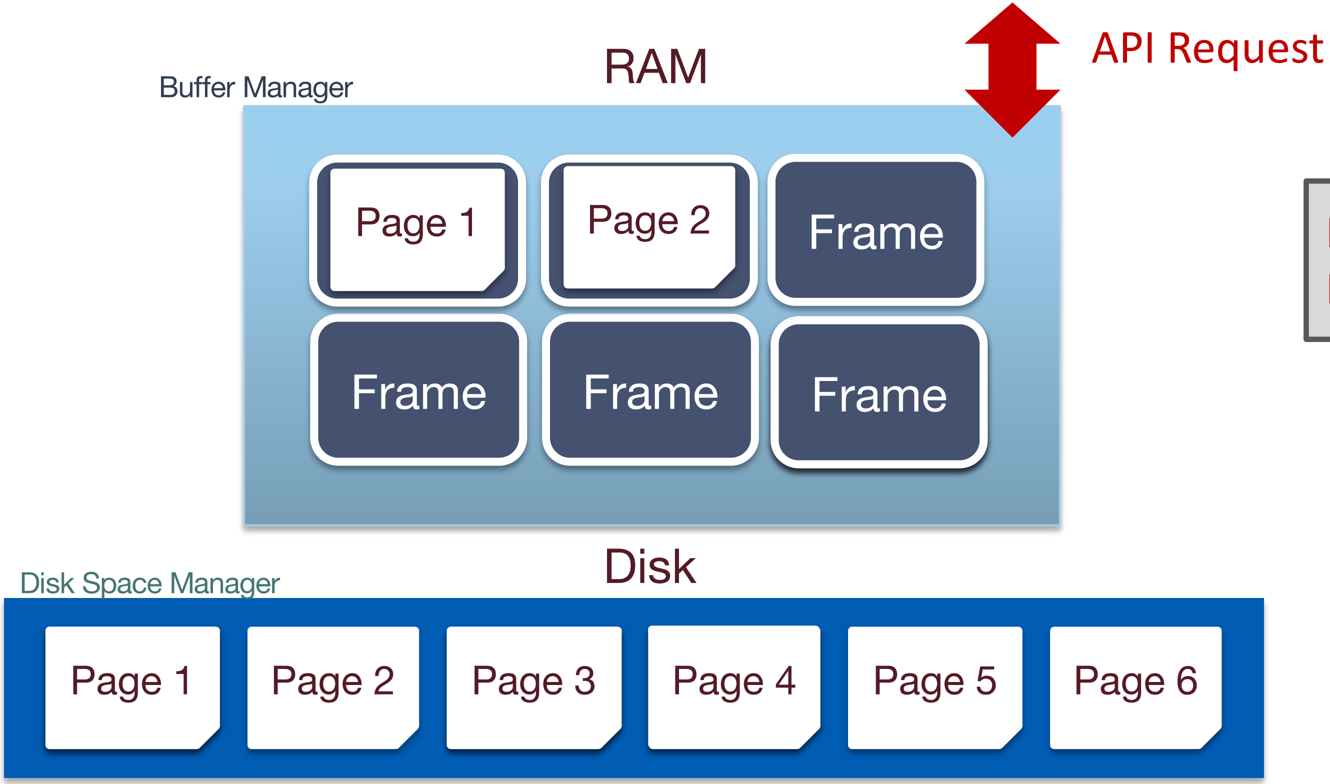
Message:
Request: Read page 1

Mapping Pages Into Memory, Pt 2



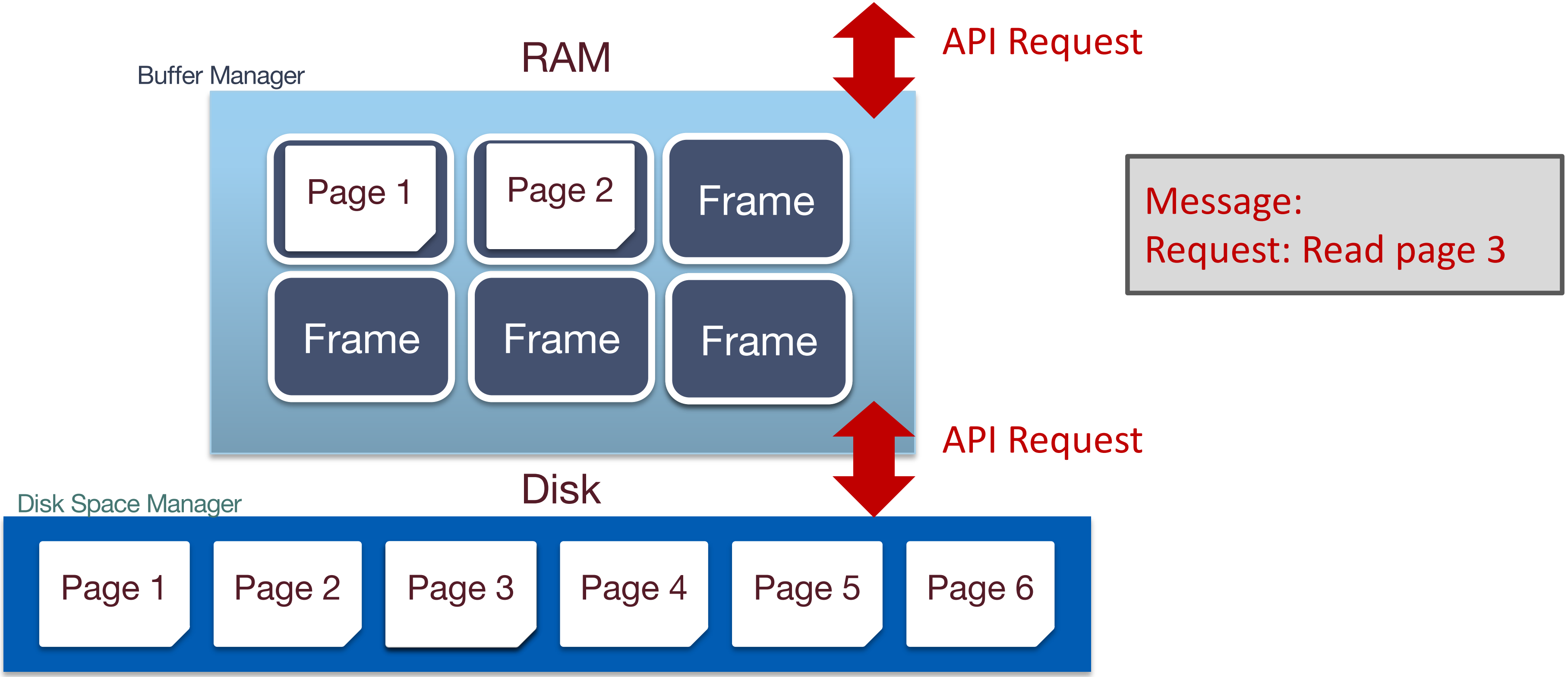
Message:
Request: Read page 2

Mapping Pages Into Memory, Pt 3

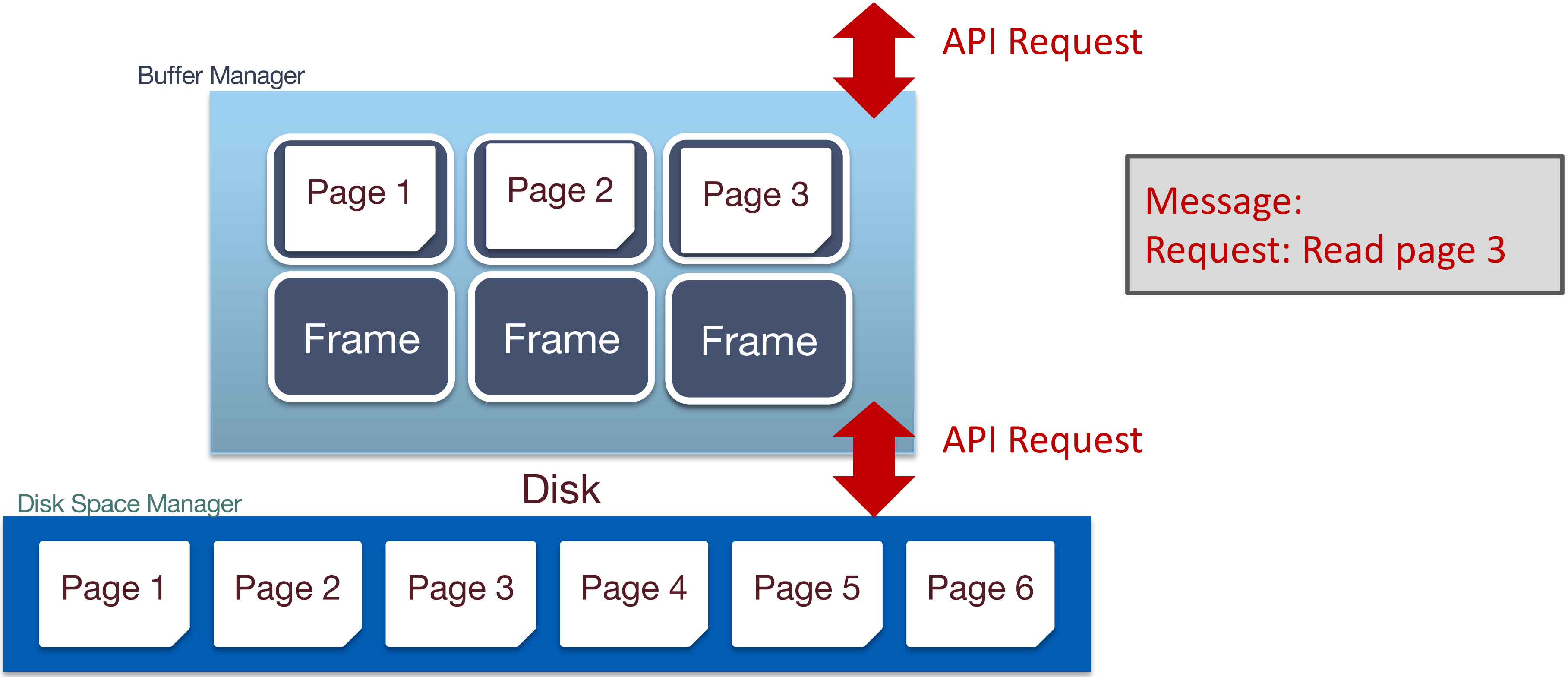


Message:
Request: Read page 2

Mapping Pages Into Memory, Pt 4



Mapping Pages Into Memory

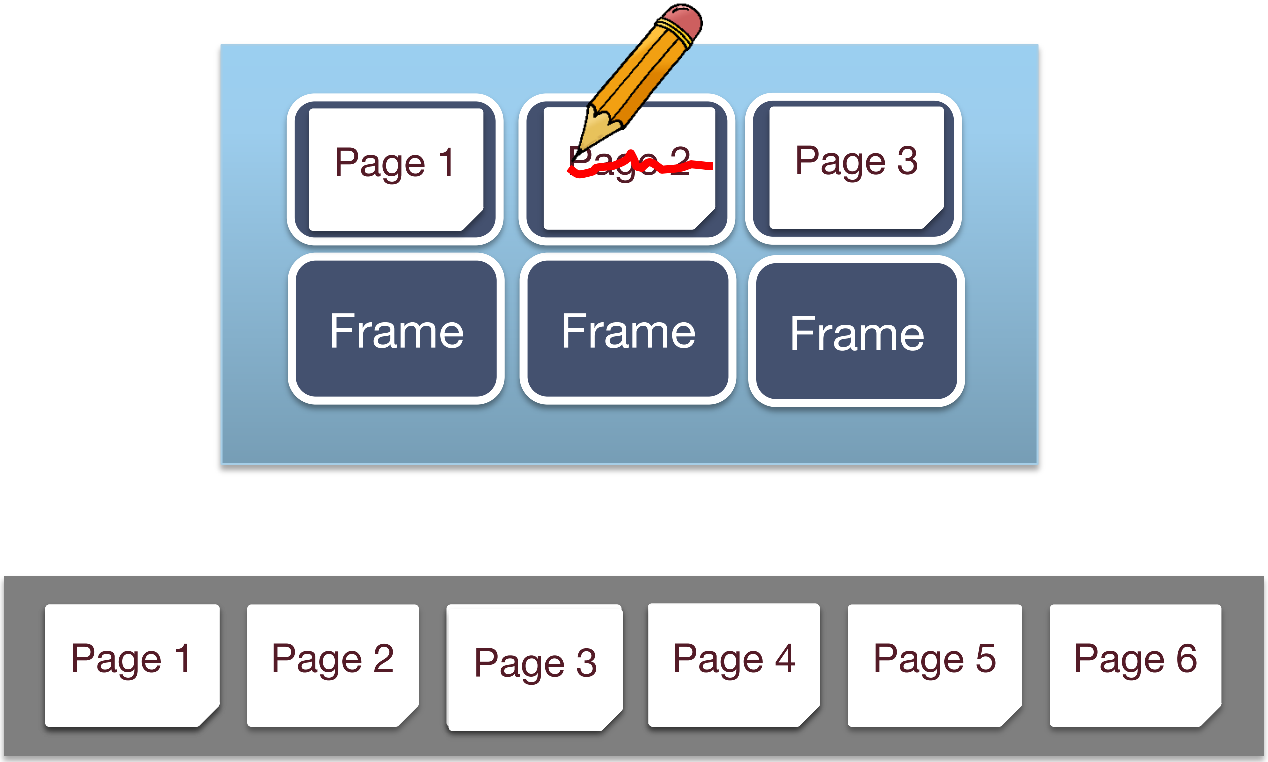


Questions We Need to Answer

1. Handling dirty pages

2. Page Replacement

Q1: Dirty Pages?

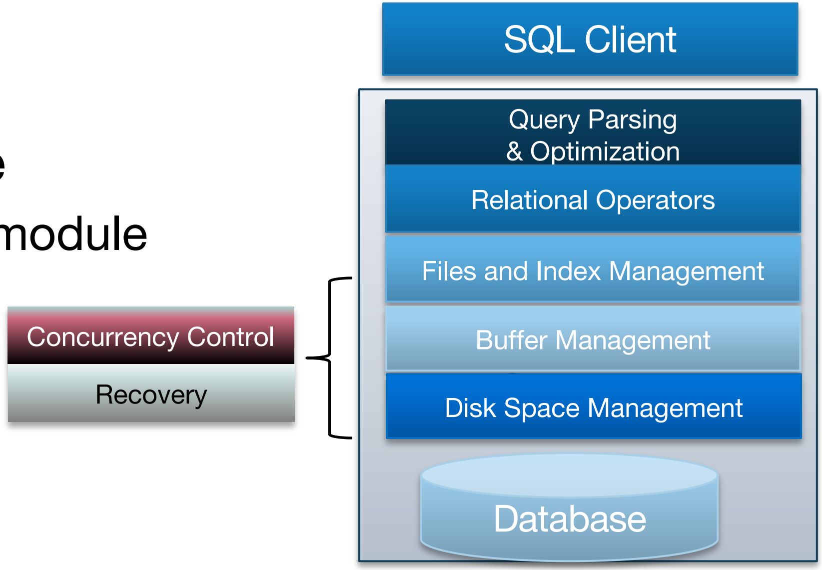


Handling Dirty Pages

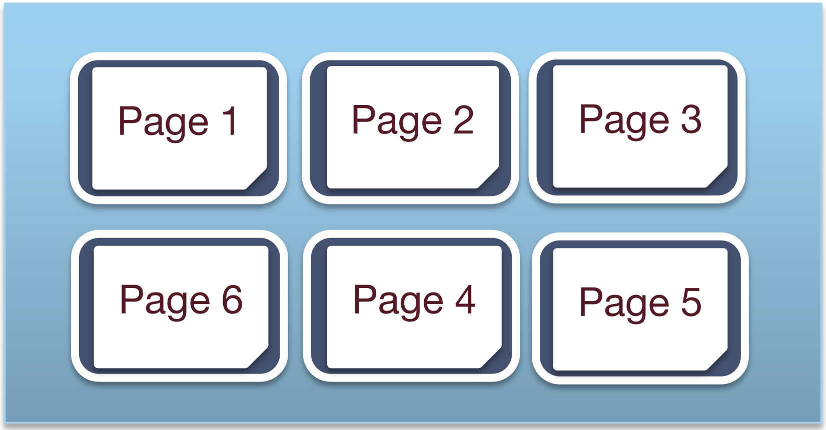
- Handling dirty pages
  - How will the buffer manager find out?
    - Dirty bit on page
    - have a bit associated with each frame, it is going to say that the page in that frame is dirty
  - What to do with a dirty page?
    - Write back via disk manager

Advanced Questions

- Concurrent operations on a page
  - Solved by Concurrency Control module
- System Crash before write-back
  - Solved by Recovery module

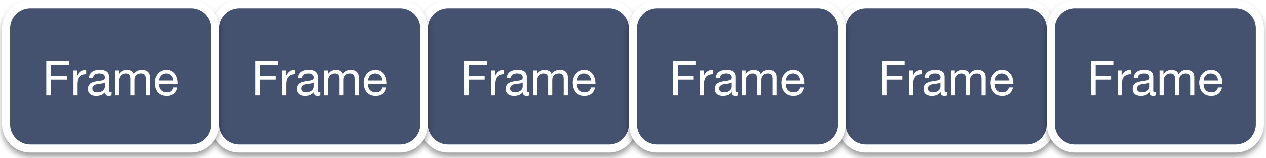


BufMgr State



BufMgr State: Explicit

Buffer pool: Large range of memory, malloc'ed at DBMS server boot time (MBs-GBs)



FrameId	PageId	Dirty?	Pin Count
1			
2			
3			
4			
5			
6			

BufMgr State: Explicit Pt 2

Buffer pool: Large range of memory, malloc'ed at DBMS server boot time (MBs-GBs)

Frame	Frame	Frame	Frame	Frame	Frame

Buffer Manager metadata: Small array in memory, malloc'ed at DBMS server boot time

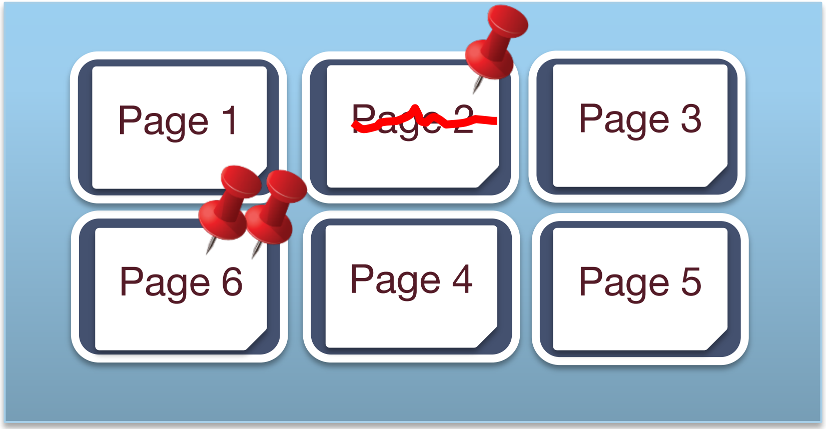
FrameId	PageId	Dirty?	Pin Count
1	1	N	0
2	2	Y	1
3	3	N	0
4	6	N	2
5	4	N	0
6	5	N	0

- keep this data structure indexed by the page id
- where is page 6 ? page 6 is in frame id 4.
- how we go find the physical location at RAM, of the page 6 from the disk

BufMgr State: Illustrated

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

BufMgr State: Illustrated 2



Page Replacement Terminology Review

- How will the buffer manager know if a page is “in use”?
  - **Page pin count**
- If buffer manager is full, what page should be replaced?
  - **Page replacement policy**

When a Page is Requested …

1. If requested page is not in pool:
    a. Choose an **un-pinned** (pin_count = 0) frame.
       where we put the new page and replace whatever page is in there now

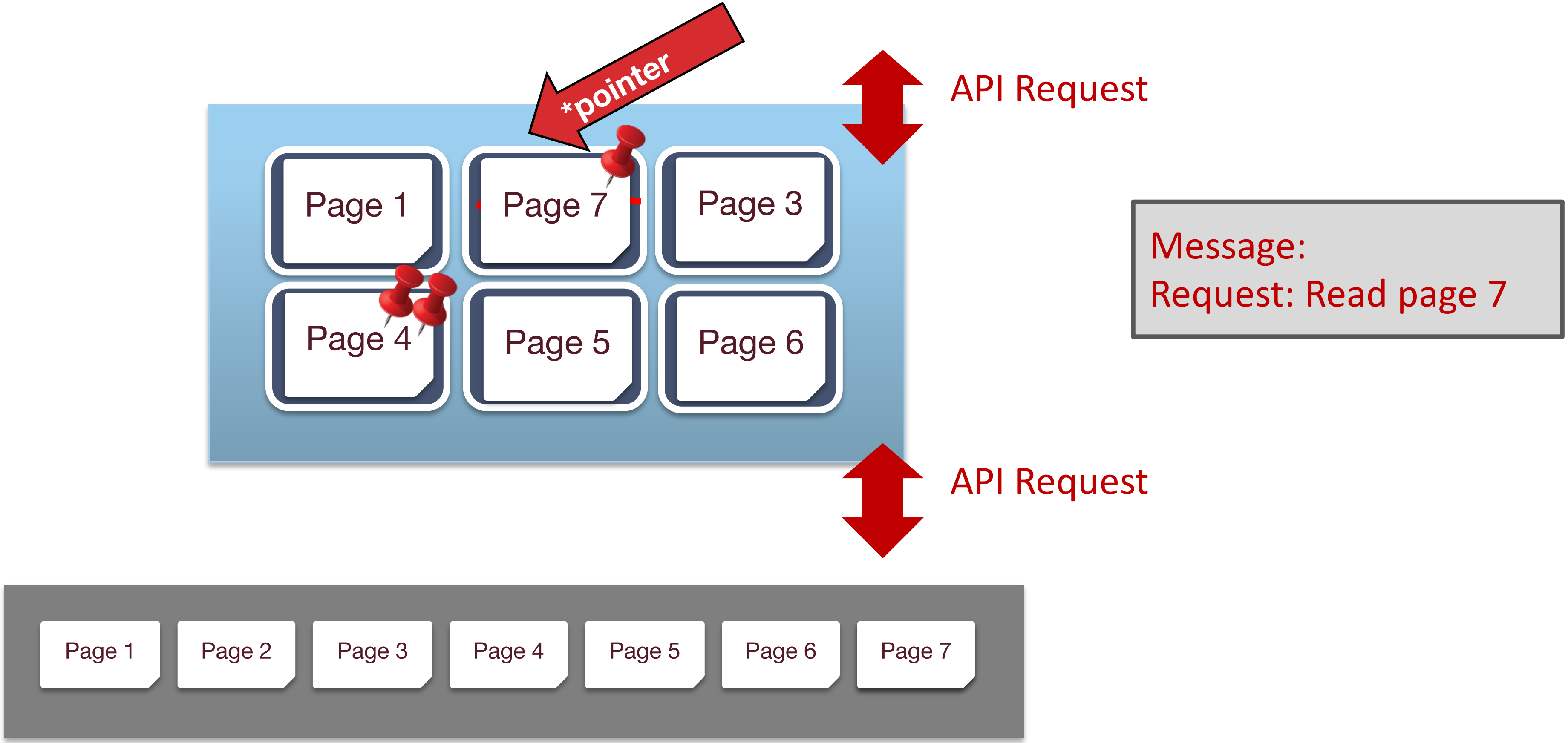
    b. For the page going to be replaced:
       If frame "dirty", write current page to disk, mark "clean"
    c. Read requested page into frame

2. Pin the page and return its address to the requester

If requests can be predicted (e.g., sequential scans) pages can be pre-fetched

- several pages at a time!

Q2: Page Replacement



After Requestor Finishes

1. Requestor of page must:
   - set dirty bit if page was modified
   - unpin the page (preferably soon!)
     - Why does requestor unpin?
     - What happens if they don't do it soon?

2. Page in pool may be requested many times
   - a pin count is used.
   - To pin a page: pin_count++
   - A page is a candidate for replacement iff
     - pin_count == 0 ("unpinned")

3. CC & recovery may do additional I/Os upon replacement
   - Write Ahead Log protocol; more later!

Answers to Our Previous Questions

1. **Handling dirty pages**
   - How will the buffer manager find out?
     - Dirty bit on page
   - What to do with a dirty page?
     - Write back via disk manager

2. **Page Replacement**
   - How will the buffer mgr know if a page is “in use”?
     - Page pin count
   - **If buffer manager is full, which page should be replaced?**
     - **Page replacement policy**

Page Replacement Policy Intro

- Page is chosen for replacement by a **replacement policy:**
  - Least-recently-used (LRU), Clock
  - Most-recently-used (MRU)

- Policy can have big impact on #I/Os
  - the choice of the best policy depends on the work that your system has to support, on the access patterns of your queries,

LRU Replacement Policy

- Least Recently Used (LRU)
  - Pinned Frame: not available to replace
  - Track time each frame last unpinned (end of use)
    - That's why unpin soon after use
  - Replace the frame which was least recently used

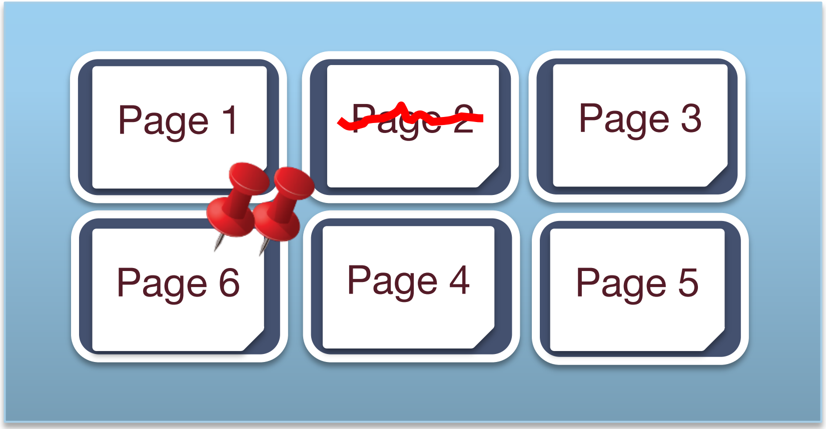
FrameId	PageId	Dirty?	Pin Count	Last Used
1	1	N	0	43
2	2	Y	1	21
3	3	N	0	22
4	6	N	2	11
5	4	N	0	24
6	5	N	0	15

LRU Replacement Policy, Pt 2

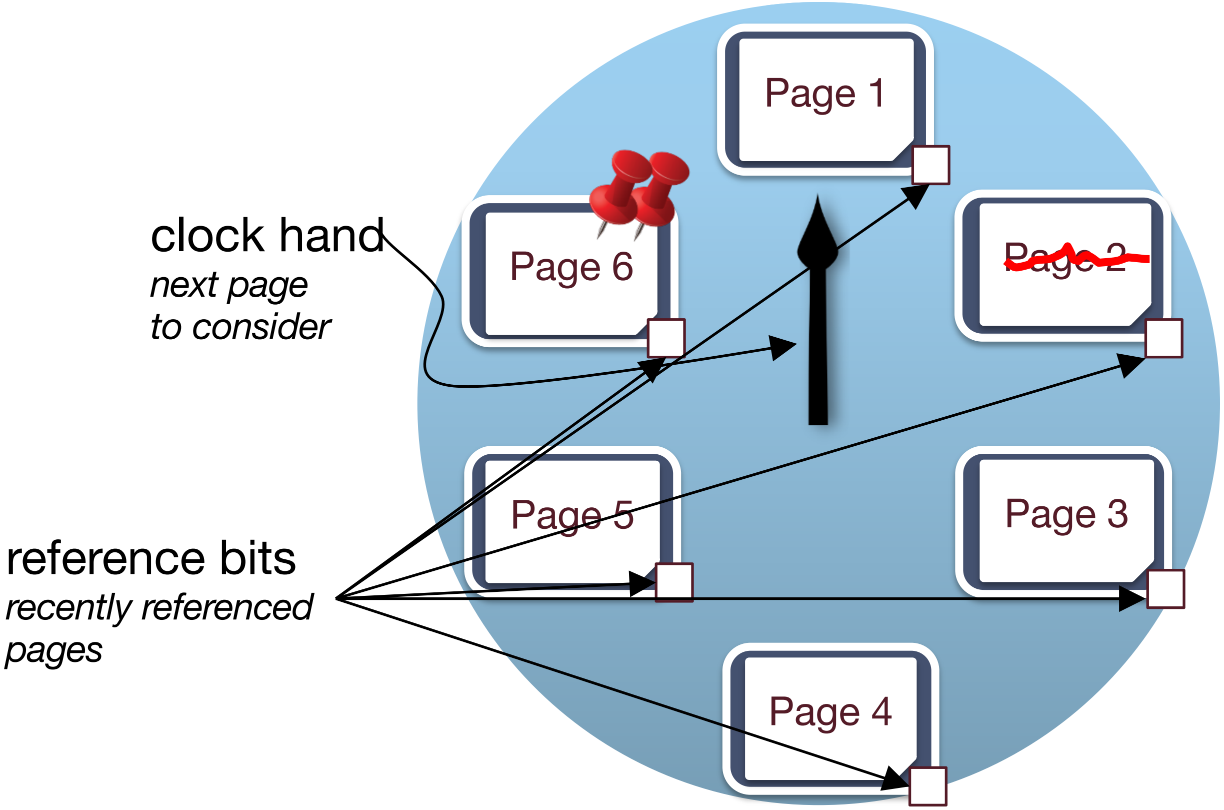
- Very common policy: intuitive and simple
  - Good for repeated accesses to popular pages (temporal locality)
  - Can be costly. Why?
    - Need to “find min” on the last used attribute (priority heap data structure)
- Approximate LRU: CLOCK policy

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

BufMgr State: Illustrated



Clock Policy State: Illustrated



Clock Policy State: Explicit

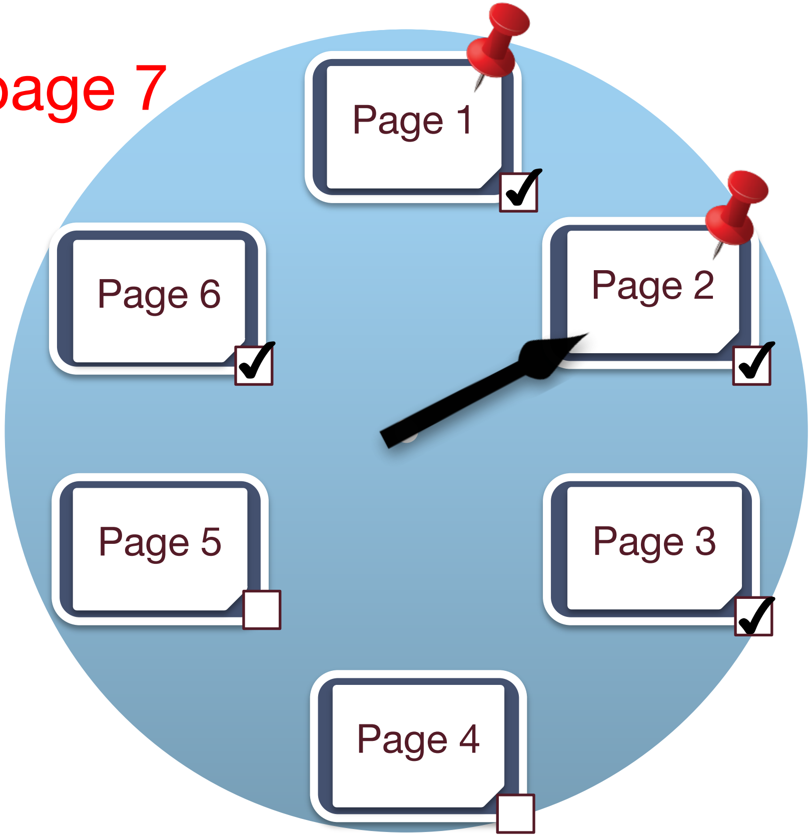
FrameId	PageId	Dirty?	Pin Count	Ref Bit
1	1	N	1	1
2	2	N	1	1
3	3	N	0	1
4	4	N	0	0
5	5	N	0	0
6	6	N	0	1

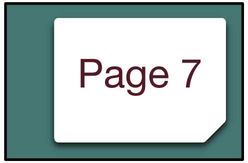
Clock Hand
1

Clock Policy State: Illustrated Part 1

Request: Read page 7

Current frame has pin-count > 0:
Skip





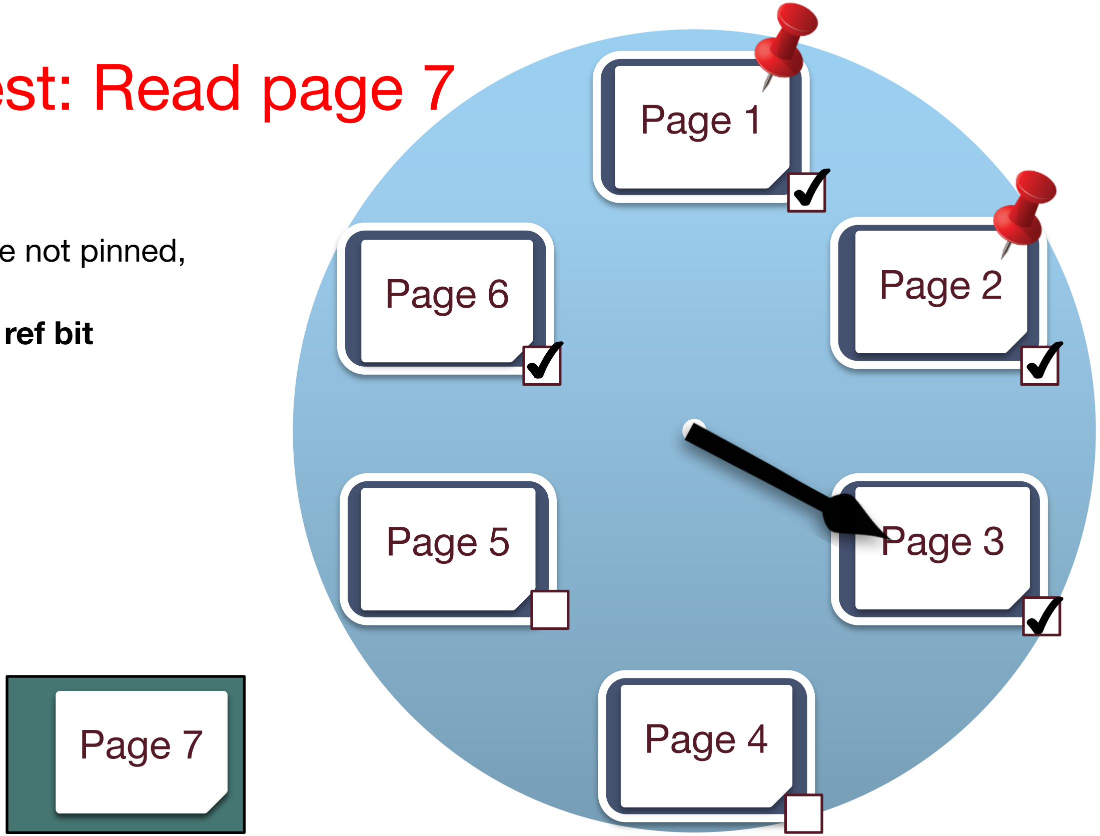
Clock Policy State: Illustrated, Part 2

Request: Read page 7

Current frame not pinned,
Ref bit set:

Clear ref bit

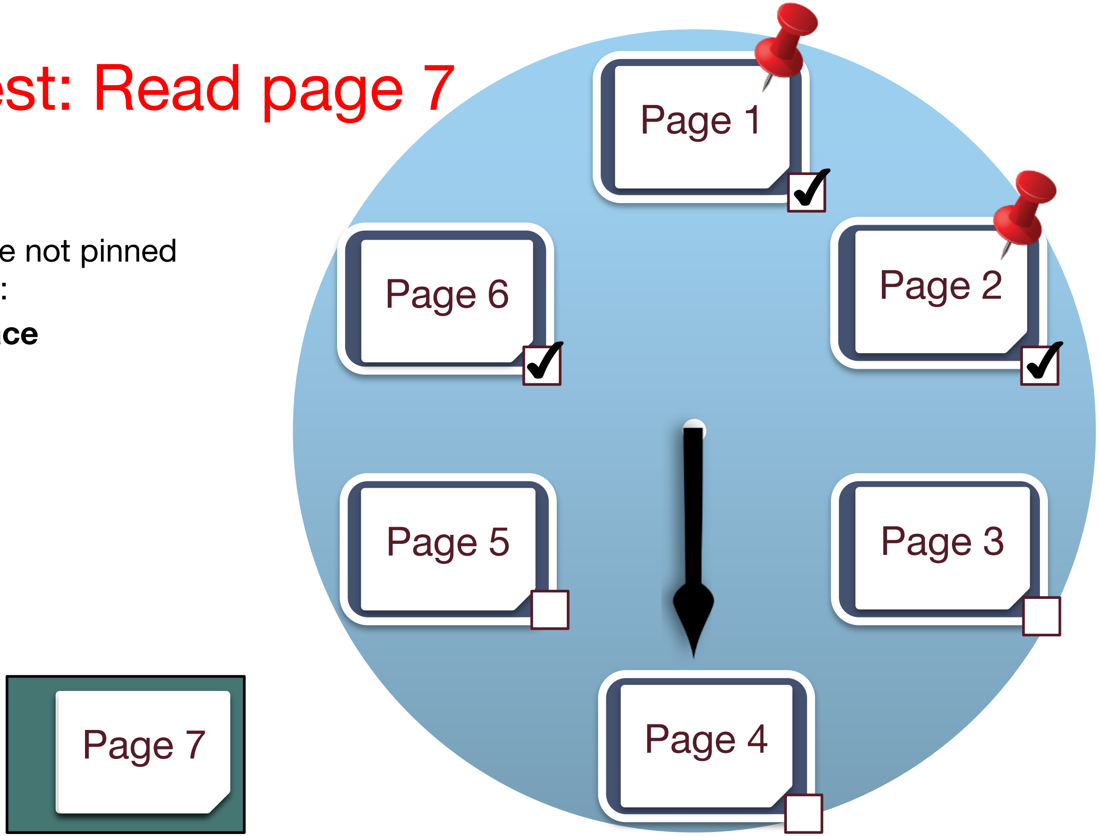
Skip



Clock Policy State: Illustrated, Pt 3

Request: Read page 7

Current frame not pinned
Ref bit unset:
Replace



Clock Policy State: Illustrated, Pt 4

Request: Read page 7

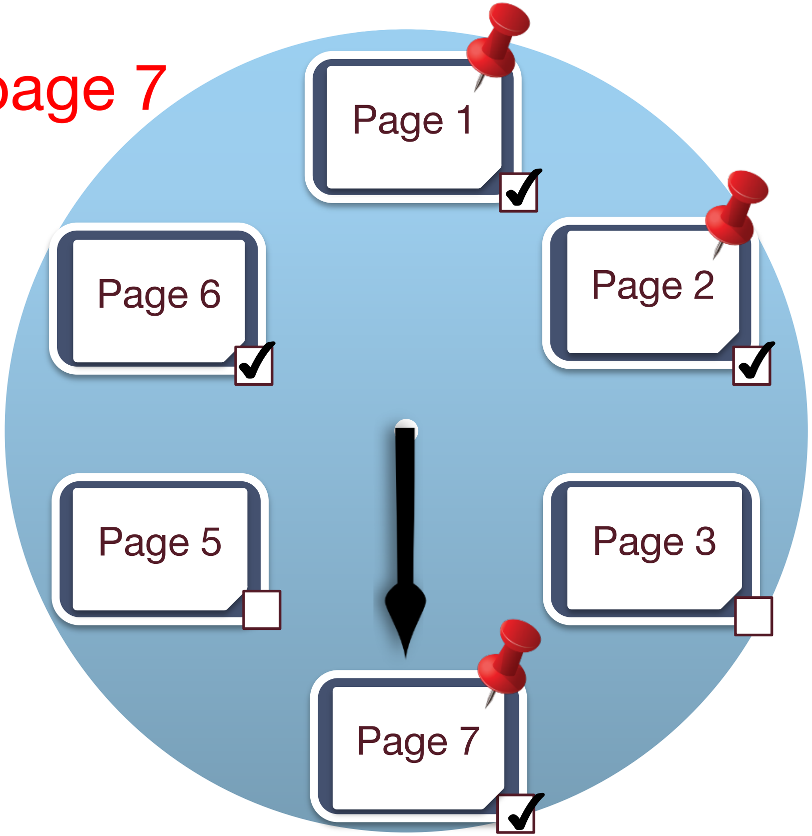
Current frame not pinned
Ref bit unset:

Replace

Set pinned

Set ref bit

Advance clock



Clock Policy State: Illustrated, Pt 5

Request: Read page 7

Current frame not pinned
Ref bit unset:

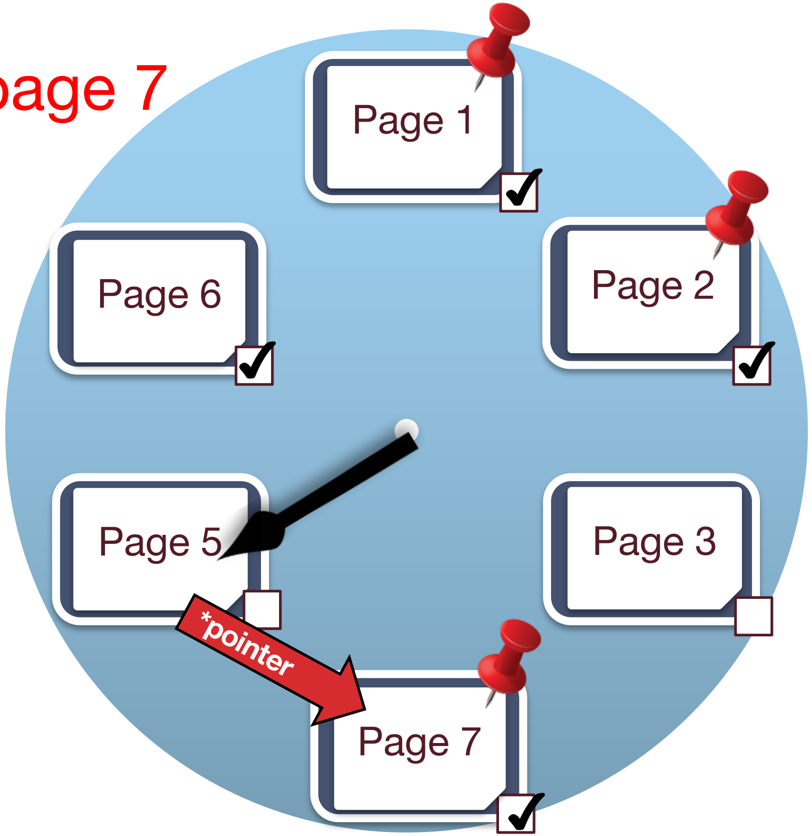
Replace

Set pinned

Set ref bit

Advance clock

Return pointer



Clock Policy Pseudocode

```
page *clock_request_page(int &clk_hand, int pg_num) {
  retval = NULL;
  while (retval == NULL) {
    current = frame_table[clk_hand];
    // the happy case: replace current page
    if (current.pin_count == 0 && current.refbit == 0) {
      if (current.dirty == 1)
        write_page(fi.page, frames[clk_hand]);
      read_page(pg_num, frames[clk_hand]);
      retval = frames[clk_hand];
      current.dirty = 0;
      current.pin_count = 1;
      current.refbit = 1; // referenced!
    }
    // second chance: unset reference bit
    else if (current.pin_count == 0 && current.refbit == 1) {
      current.refbit == 0;
    }
    // else pin_count > 1, so skip

    clk_hand += (clk_hand + 1) % MAX_FRAME; // advance clock hand
  }
  return retval;
}
```

Clock Policy Pseudocode, Pt 2

```
page *clock_request_page(int &clk_hand, int pg_num) {
  retval = NULL;
  while (retval == NULL) {
    current = frame_table[clk_hand];
    // the happy case: replace current page
    if (current.pin_count == 0 && current.refbit == 0) {
      if (current.dirty == 1)
        write_page(fi.page, frames[clk_hand]);
      read_page(pg_num, frames[clk_hand]);
      retval = frames[clk_hand];
      current.dirty = 0;
      current.pin_count = 1;
      current.refbit = 1; // referenced!
    }
    // second chance: unset reference bit
    else if (current.pin_count == 0 && current.refbit == 1) {
      current.refbit == 0;
    }
    // else pin_count > 1, so skip

    clk_hand += (clk_hand + 1) % MAX_FRAME; // advance clock hand
  }
  return retval;
}
```

Clock Policy Pseudocode, Pt 3

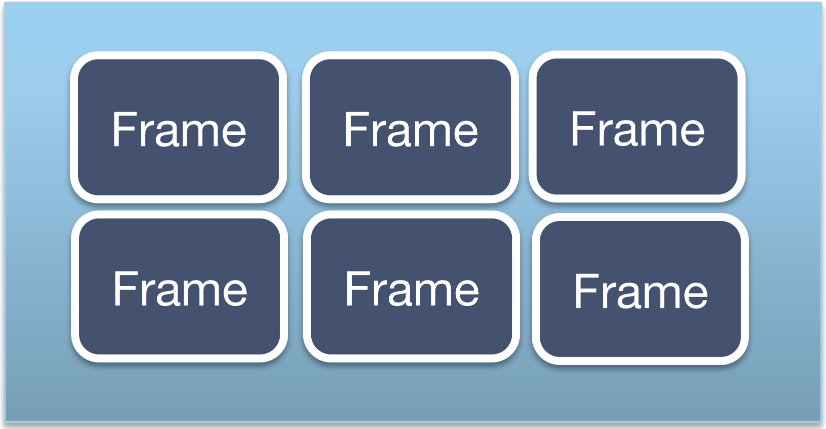
```
page *clock_request_page(int &clk_hand, int pg_num) {
  retval = NULL;
  while (retval == NULL) {
    current = frame_table[clk_hand];
    // the happy case: replace current page
    if (current.pin_count == 0 && current.refbit == 0) {
      if (current.dirty == 1)
        write_page(fi.page, frames[clk_hand]);
      read_page(pg_num, frames[clk_hand]);
      retval = frames[clk_hand];
      current.dirty = 0;
      current.pin_count = 1;
      current.refbit = 1; // referenced!
    }
    // second chance: unset reference bit
    else if (current.pin_count == 0 && current.refbit == 1) {
      current.refbit == 0;
    }
    // else pin_count > 1, so skip

    clk_hand += (clk_hand + 1) % MAX_FRAME; // advance clock hand
  }
  return retval;
}
```

Is LRU/Clock Always Best?

- Very common policy: intuitive and simple
- Works well for repeated accesses to popular pages
  - Temporal locality
- LRU can be costly → Clock policy is cheap
  - Quite similar
  - If you like, try to find cases where they differ.
- When might they perform poorly
  - What about repeated scans of big files?

Repeated Scan (LRU)

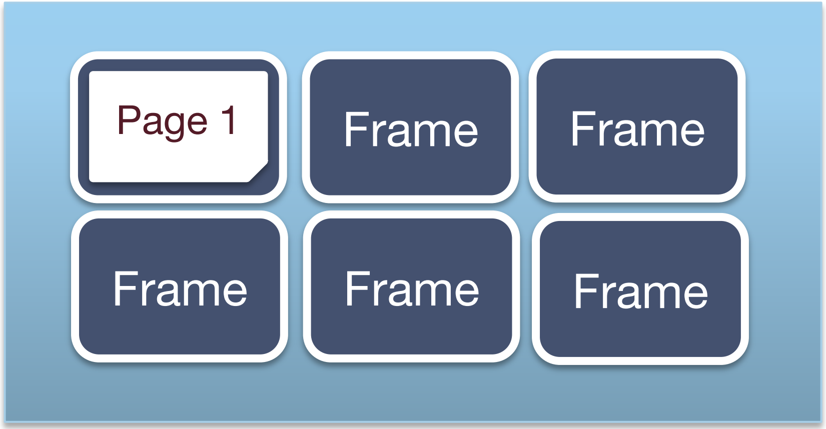
- Cache Hits: 0
- Attempts: 0



cache hit : we see a page, you get a page request, and we find that page already in the buffer pool.

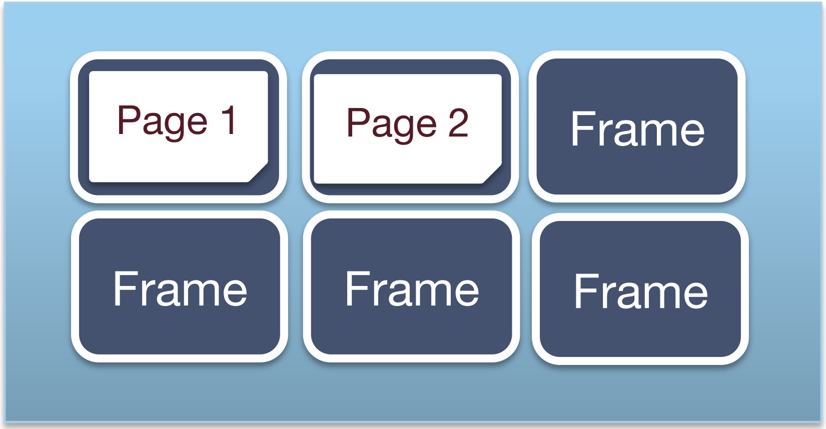
Repeated Scan (LRU): Read Page 1

- Cache Hits: 0
- Attempts: 1



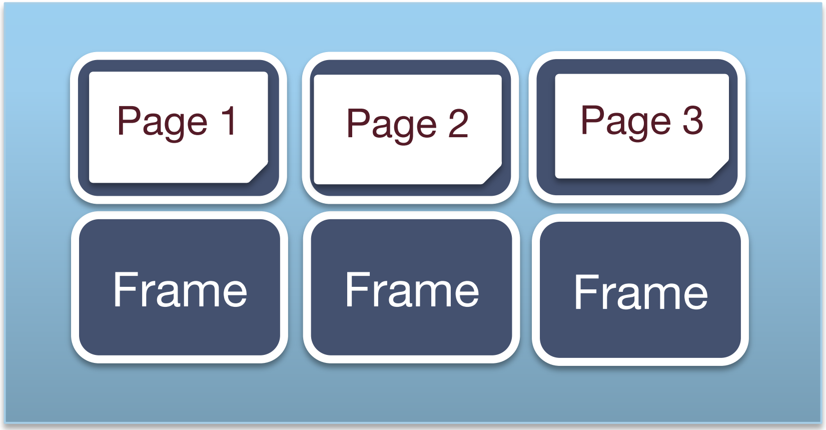
Repeated Scan (LRU): Read Page 2

- Cache Hits: 0
- Attempts: 2

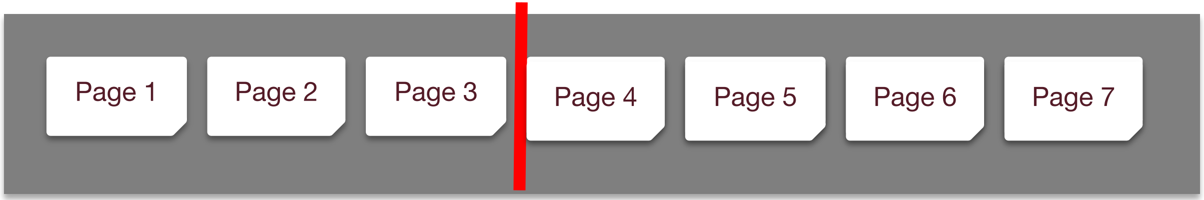


Repeated Scan (LRU): Read Page 3

- Cache Hits: 0
- Attempts 3:

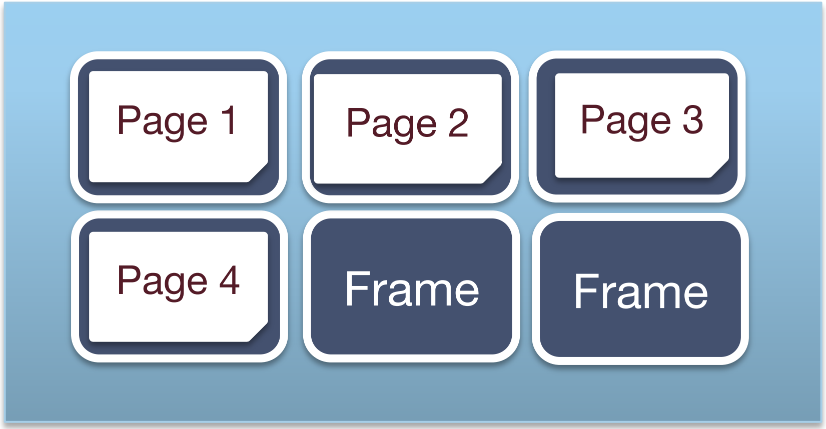


Disk Space Manager

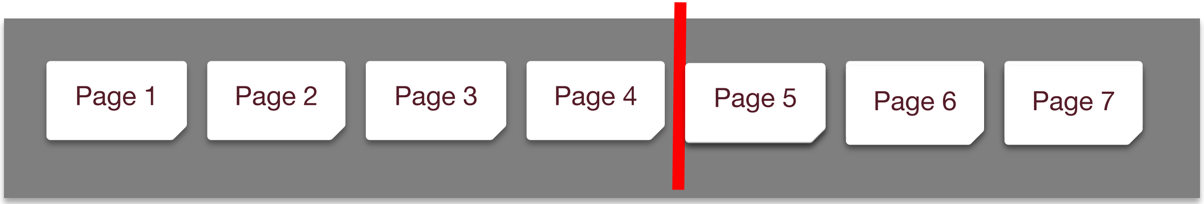


Repeated Scan (LRU): Read Page 4

- Cache Hits 0:
- Attempts: 4

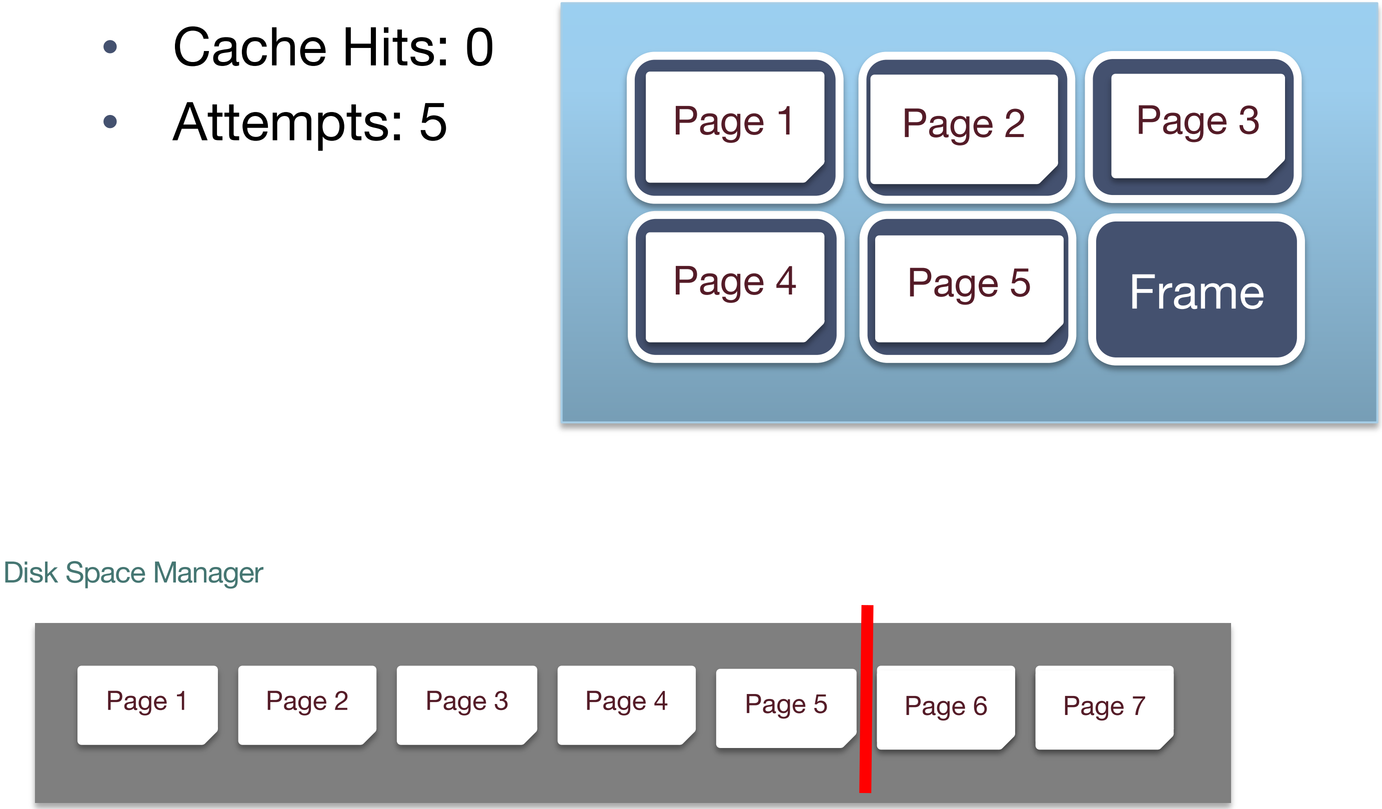


Disk Space Manager



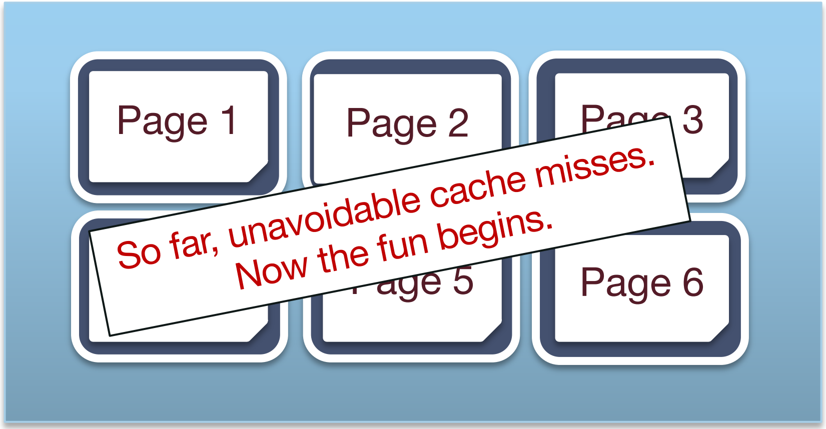
Repeated Scan (LRU): Read Page 5

- Cache Hits: 0
- Attempts: 5

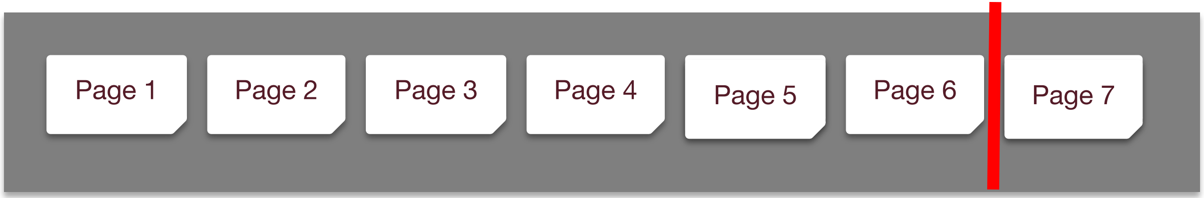


Repeated Scan (LRU): Read Page 6

- Cache Hits: 0
- Attempts 6

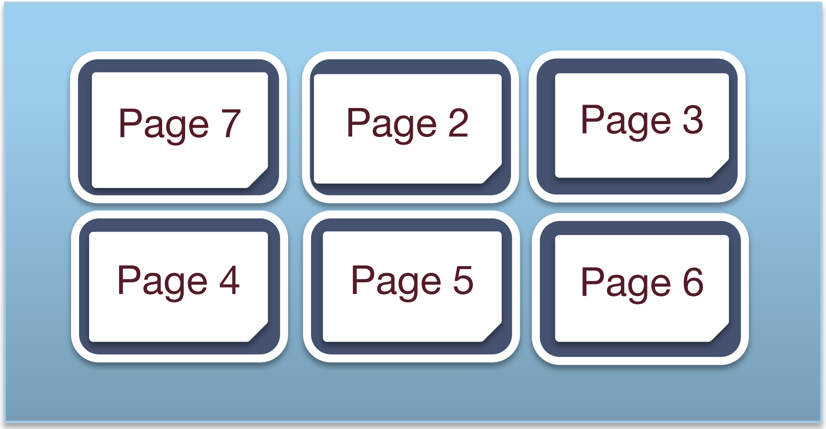


Disk Space Manager

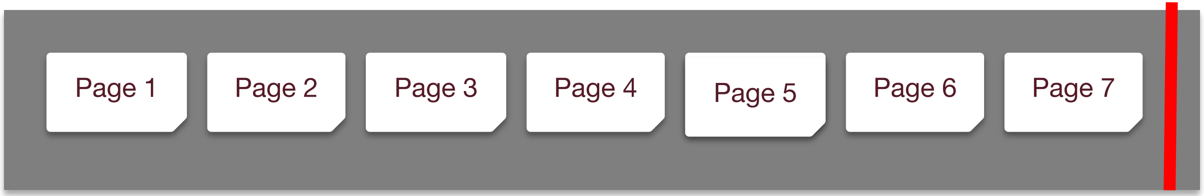


Repeated Scan (LRU): Read Page 7

- Cache Hits: 0
- Attempts: 7

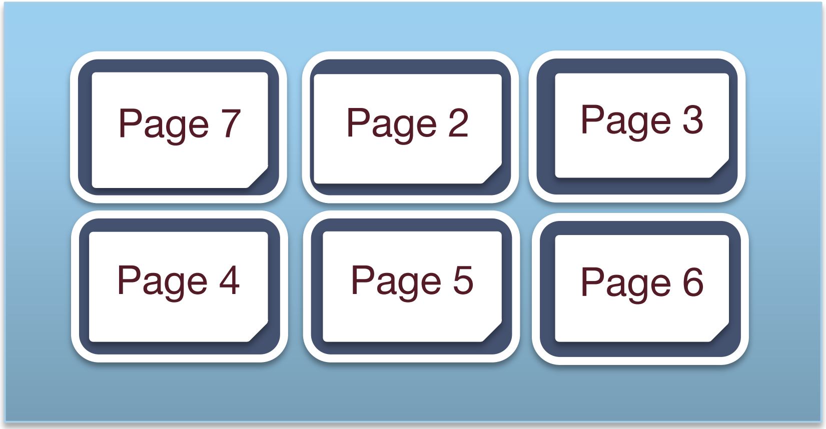


Disk Space Manager



Repeated Scan (LRU): Reset to beginning

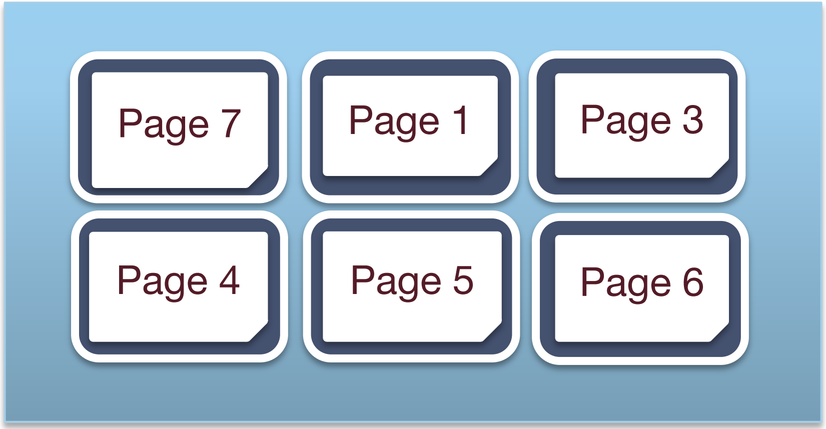
- Cache Hits: 0
- Attempts: 7



Disk Space Manager

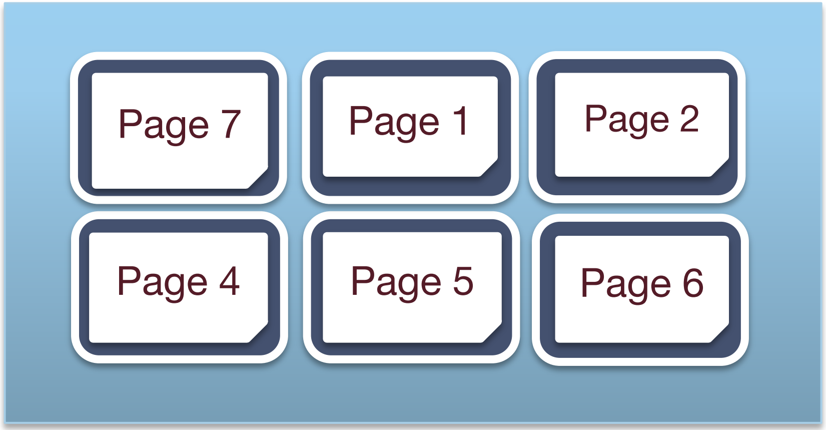
Repeated Scan (LRU): Read Page 1 (again)

- Cache Hits: 0
- Attempts: 8

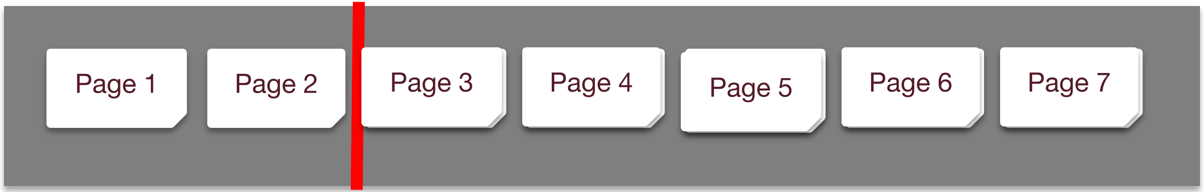


Repeated Scan (LRU): Read Page 2 (again)

- Cache Hits: 0
- Attempts: 9

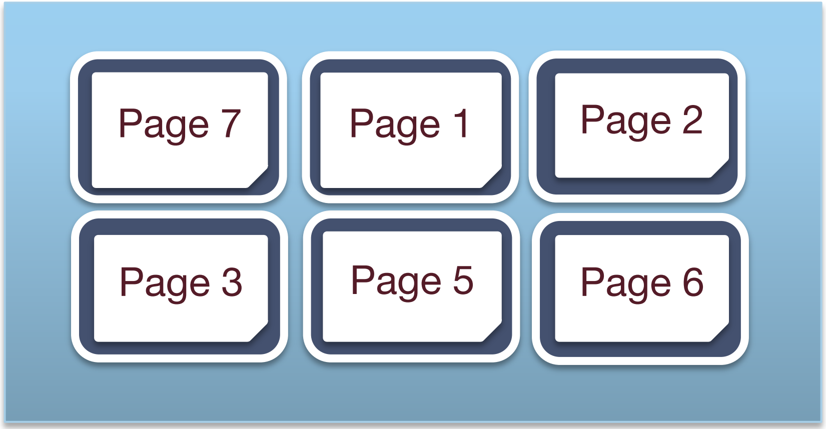


Disk Space Manager



Repeated Scan (LRU): Read Page 3 (again)

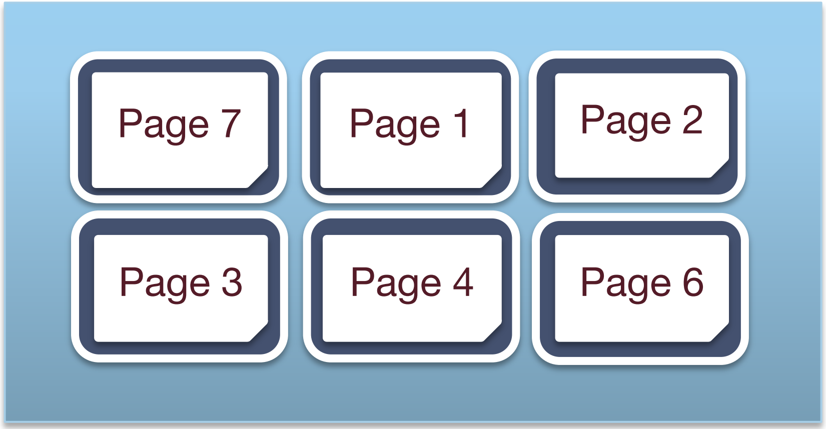
- Cache Hits: 0
- Attempts: 10



Disk Space Manager

Repeated Scan (LRU): Page 4 (again)

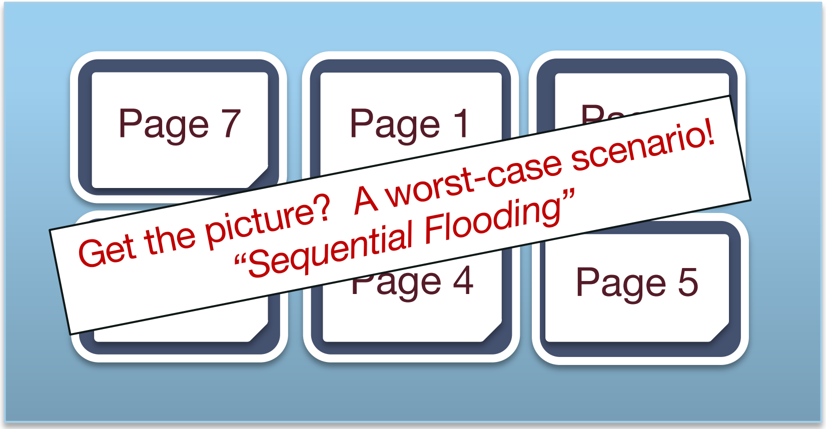
- Cache Hits: 0
- Attempts: 11



Disk Space Manager

Repeated Scan (LRU): Read Page 5, cont

- Cache Hits: 0
- Attempts: 12



Disk Space Manager

Sequential Scan + LRU

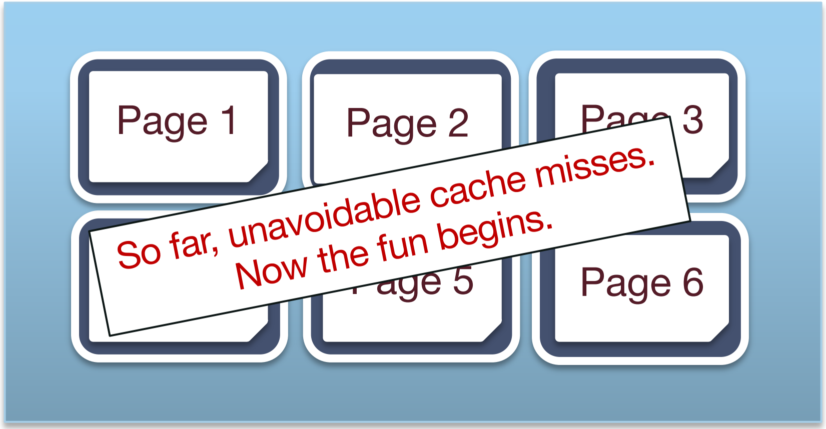
- Sequential flooding
- 0% hit rate in cache!

- Repeated sequential scan very common in database workloads
  - We will see it in nested-loops join

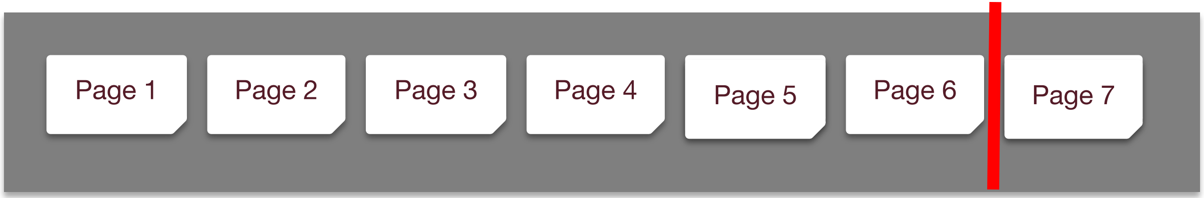
- What could be better?

Repeated Scan (MRU)

- Cache Hits: 0
- Attempts: 6

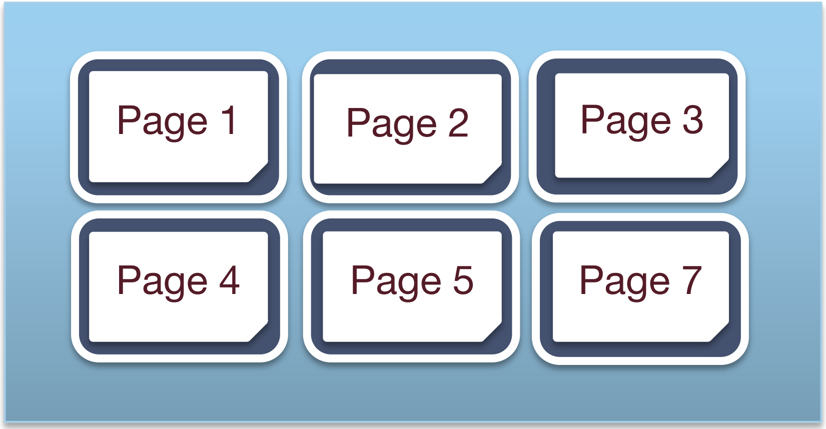


Disk Space Manager



Repeated Scan (MRU): Read Page 7

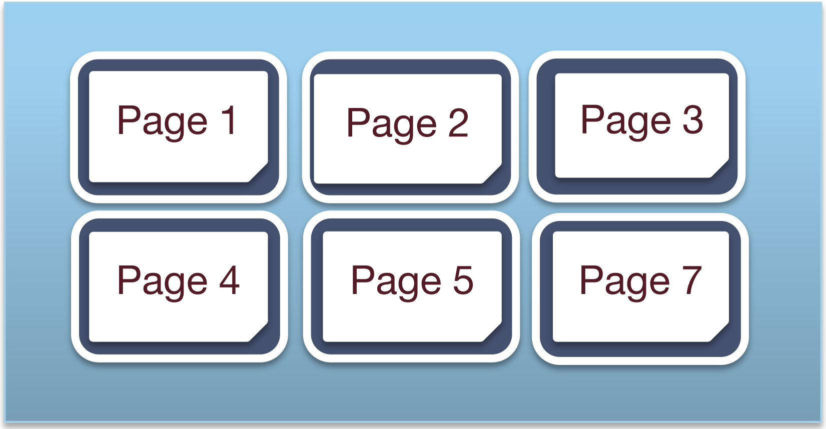
- Cache Hits: 0
- Attempts: 7



Disk Space Manager

Repeated Scan (MRU): Reset

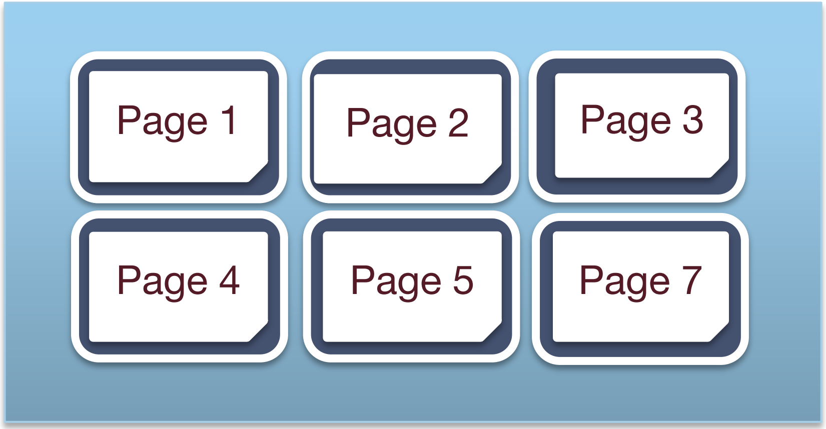
- Cache Hits: 0
- Attempts: 7



Disk Space Manager

Repeated Scan (MRU): Read Page 1 (again)

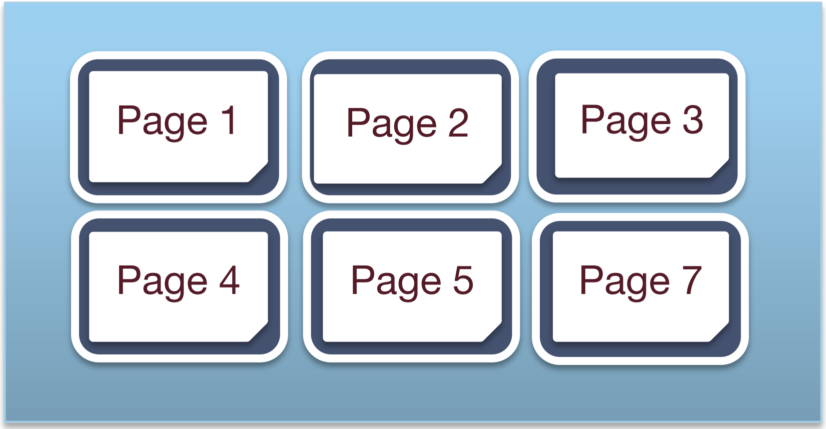
- Cache Hits: 1
- Attempts: 8



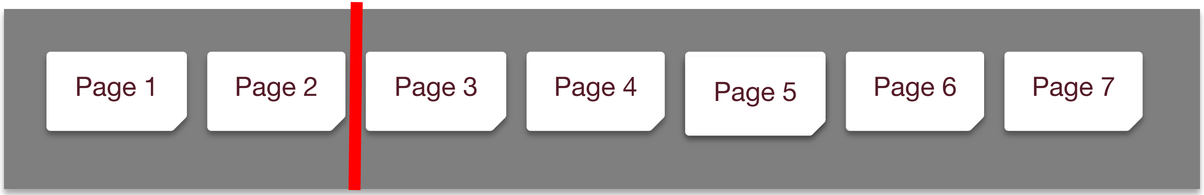
Disk Space Manager

Repeated Scan (MRU): Read Page 2 (again)

- Cache Hits: 2
- Attempts: 9

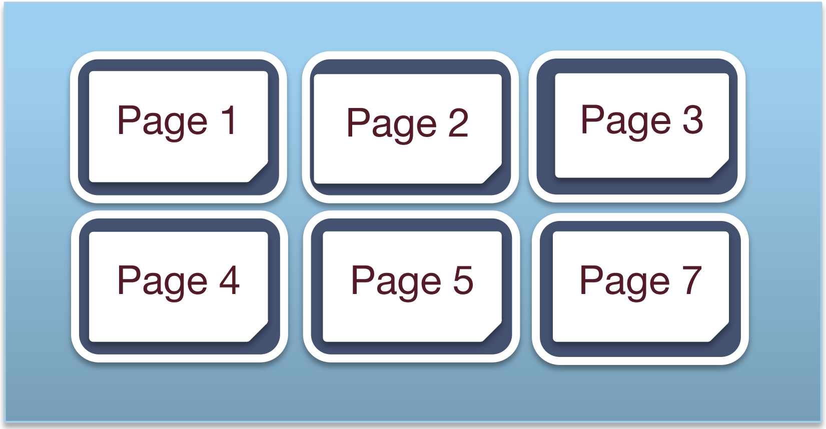


Disk Space Manager



Repeated Scan (MRU): Read Page 3 (again)

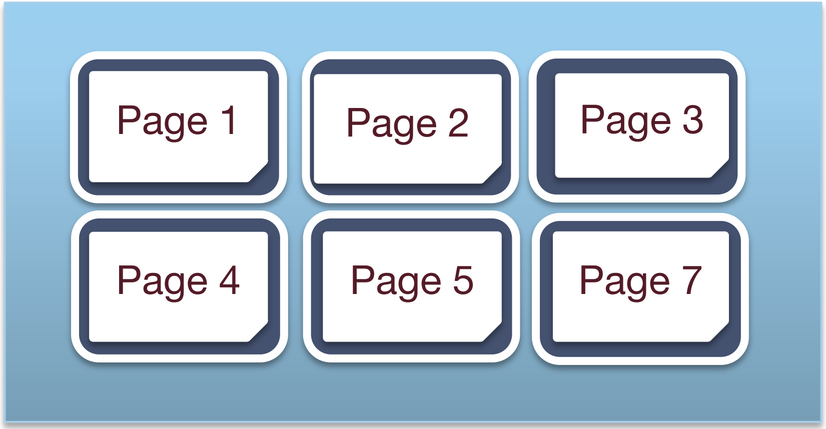
- Cache Hits: 3
- Attempts: 10



Disk Space Manager

Repeated Scan (MRU): Read Page 4 (again)

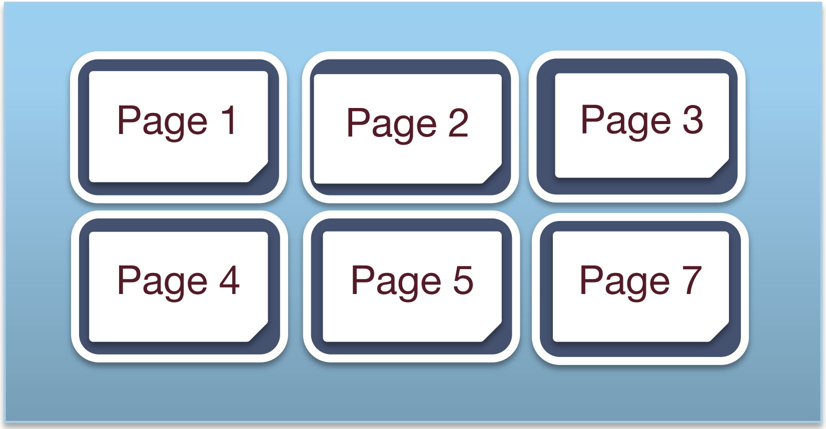
- Cache Hits: 4
- Attempts: 11



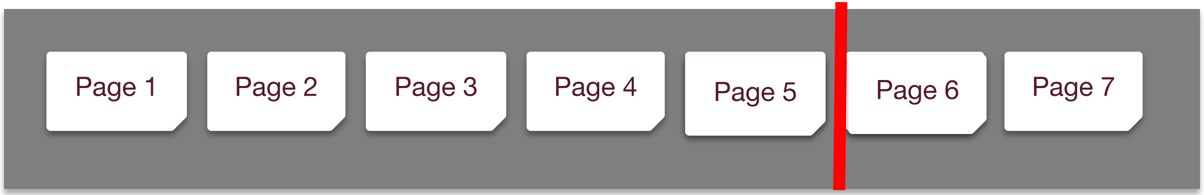
Disk Space Manager

Repeated Scan (MRU): Read Page 5 (again)

- Cache Hits: 5
- Attempts: 12

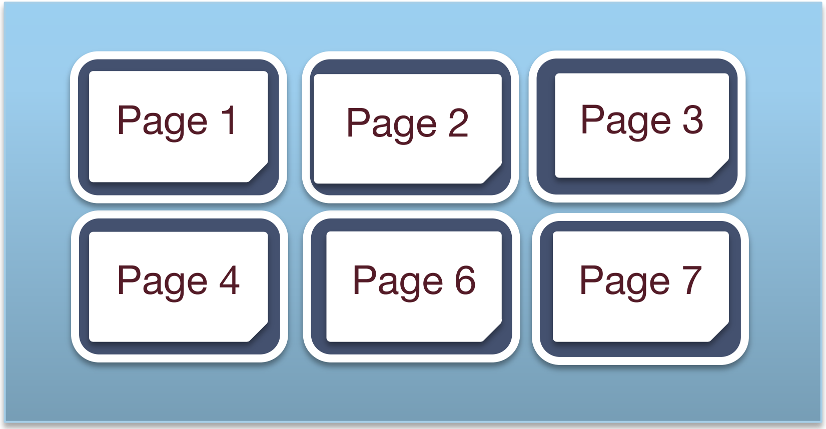


Disk Space Manager

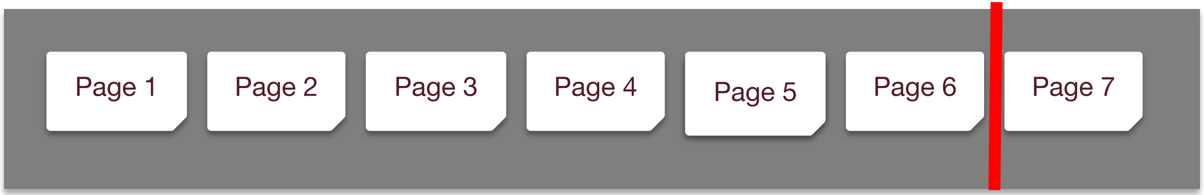


Repeated Scan (MRU): Read Page 6 (again)

- Cache Hits: 5
- Attempts: 13

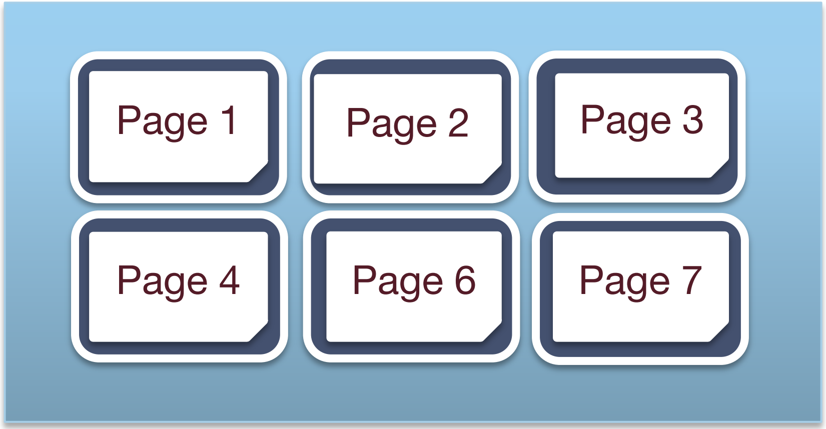


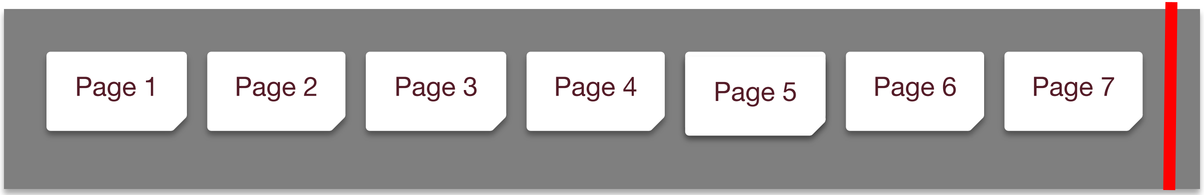
Disk Space Manager



Repeated Scan (MRU): Read Page 7 (again)

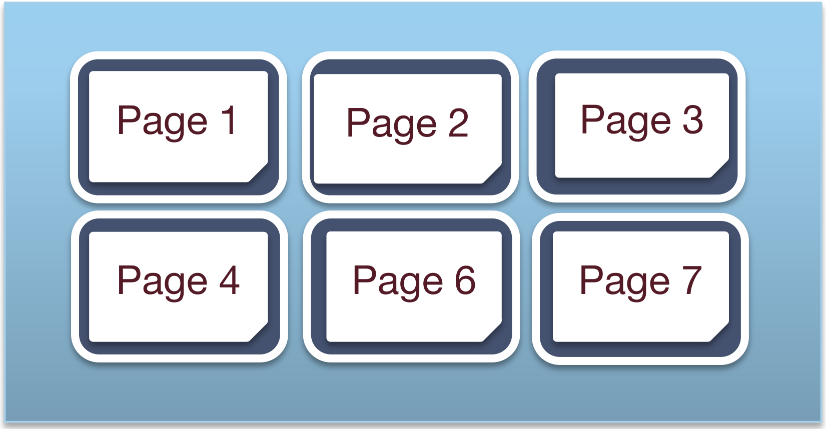
- Cache Hits: 6
- Attempts: 14

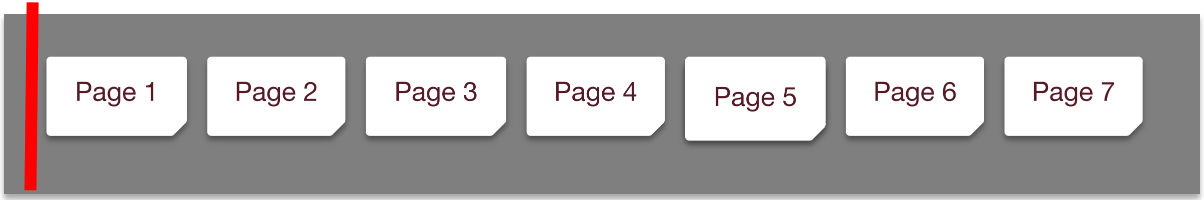




Repeated Scan (MRU): Reset (again)

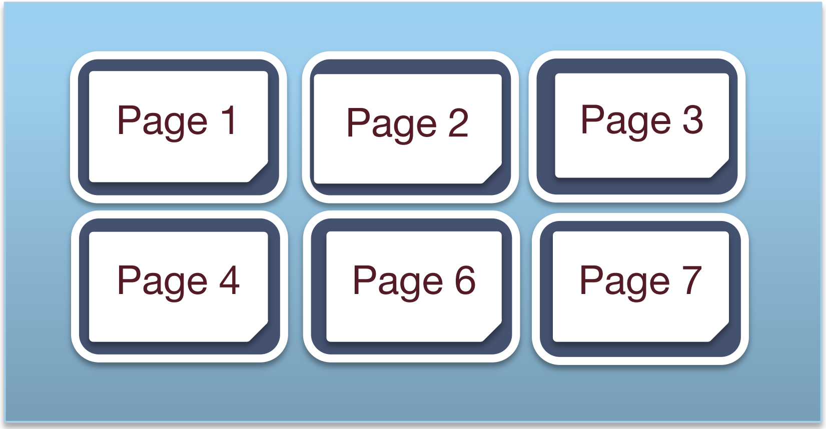
- Cache Hits: 6
- Attempts: 14





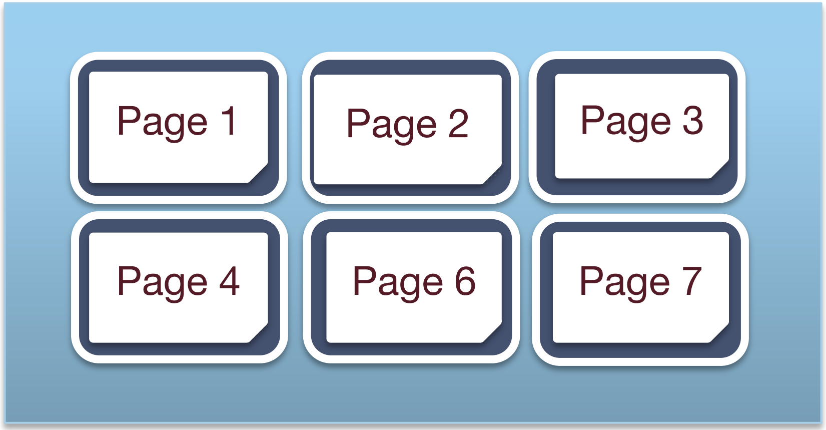
Repeated Scan (MRU): Read Page 1 (again x2)

- Cache Hits: 7
- Attempts: 15



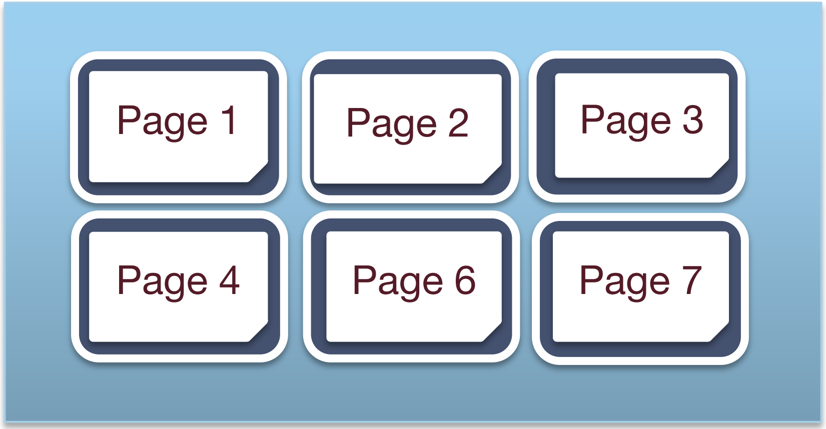
Repeated Scan (MRU): Read Page 2 (again x2)

- Cache Hits: 8
- Attempts: 16



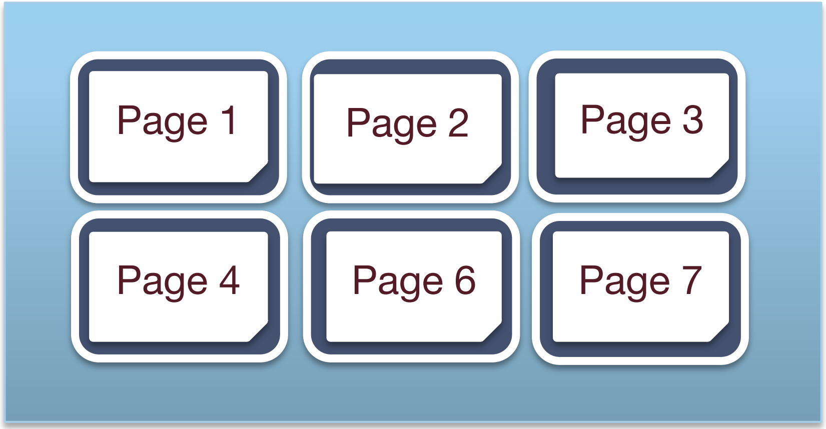
Repeated Scan (MRU): Read Page 3 (again x2)

- Cache Hits: 9
- Attempts: 17



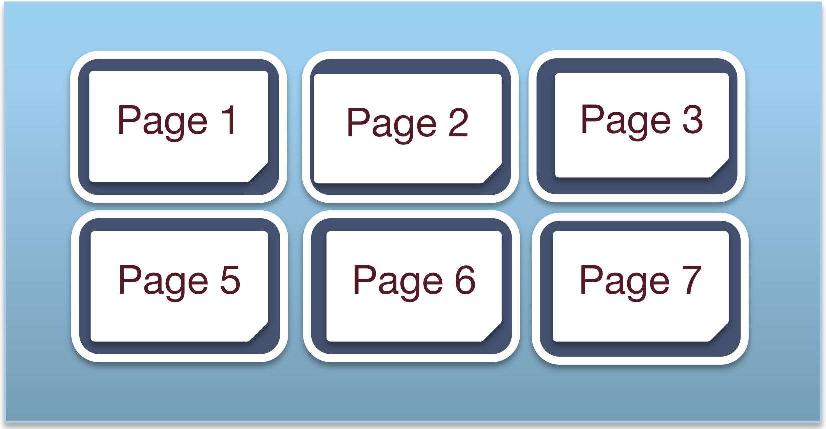
Repeated Scan (MRU): Read Page 4 (again x2)

- Cache Hits: 10
- Attempts: 18



Repeated Scan (MRU): Read Page 5 (again x2)

- Cache Hits: 10
- Attempts: 19



2	3	4
5	6	7

General Case: SeqScan + MRU

B buffers

$N > B$ pages in file

First pass (N attempts): 0 hits

The next (B– 1) passes have B hits each

The next (N – B) passes have (B – 1) hits each

The next (B– 1) passes have B hits each

…

In limit: $(B(B-1) + (B-1)(N-B)) / (N(N-1)) = (B-1)/(N-1)$ hit rate

Improvement for sequential scan: prefetch

- Prefetch: 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?
  - Amortize random I/O overhead
  - Allow computation while I/O continues in background
    - Disk and CPU are “parallel devices”

We seem to need a hybrid!

- LRU wins for random access (hot vs. cold)
    - When might we see that behavior?
- MRU wins for repeated sequential
    - E.g. for certain joins

Two General Approaches

- Use DBMS information to hint to BufferManager
  - For big queries: we can predict I/O patterns from the handful of query processing algorithms we'll learn shortly
  - For simple lookups: LRU often does well

- There are also policies that themselves trying to achieve the best of LRU and MRU, without any information about the workload
  - E.g. 2Q, LRU-2, ARC.
  - See Page Replacement Algorithm on Wikipedia but beware the OS-centric history

- Hybrids are not uncommon in modern DBMSs

Summing Up

- Buffer Manager provides a level of indirection
  - Connects RAM and Disk
  - Maps disk page Ids to RAM addresses
- Ensures that 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

Make Sure You Know

- Pin Counts and Dirty Bits:
  - When do they get set/unset?
  - By what layer of the system?
- LRU, MRU and Clock
  - Be able to run each by hand
  - For Clock:
    - What pages are eligible for replacement
    - When is reference bit set/unset
    - What is the point of the reference bit?
- Sequential flooding
  - And how it behaves for LRU (Clock), MRU