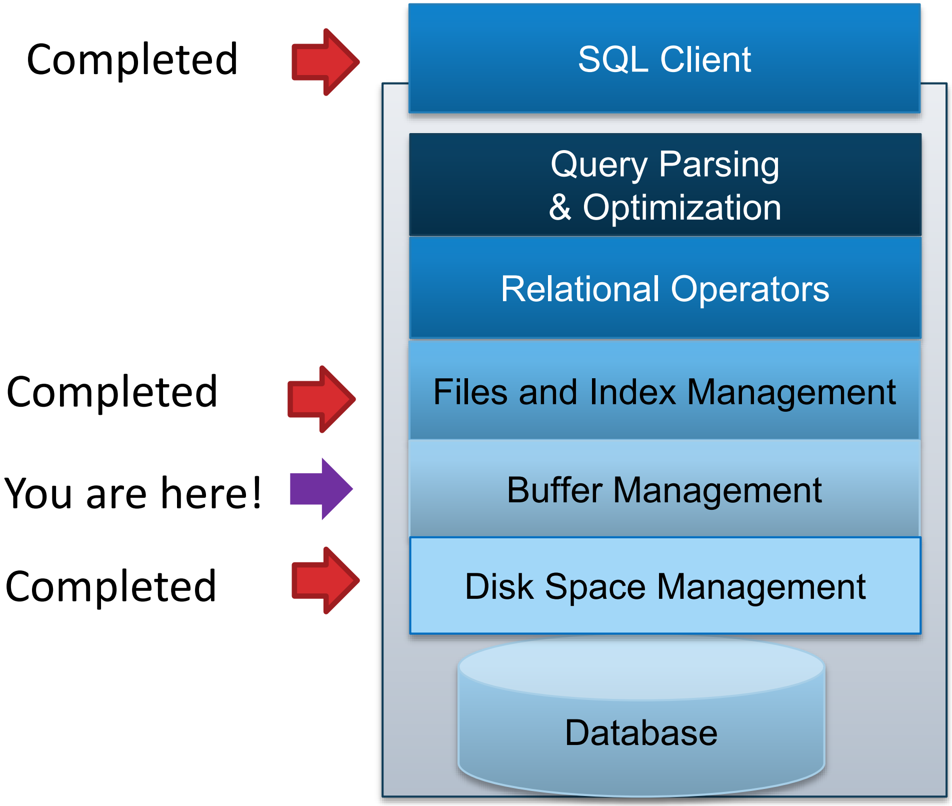
Buffer Management

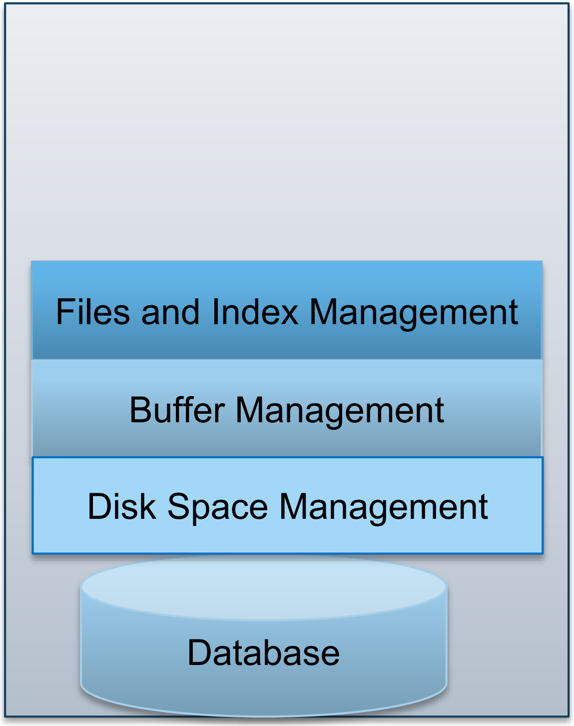
R & G - Chapter 9.4

Berkeley
cs186

Architecture of a DBMS: What we've learned



Lower Architecture of a DBMS



Buffer Management Levels of Abstraction

Files and Index Management

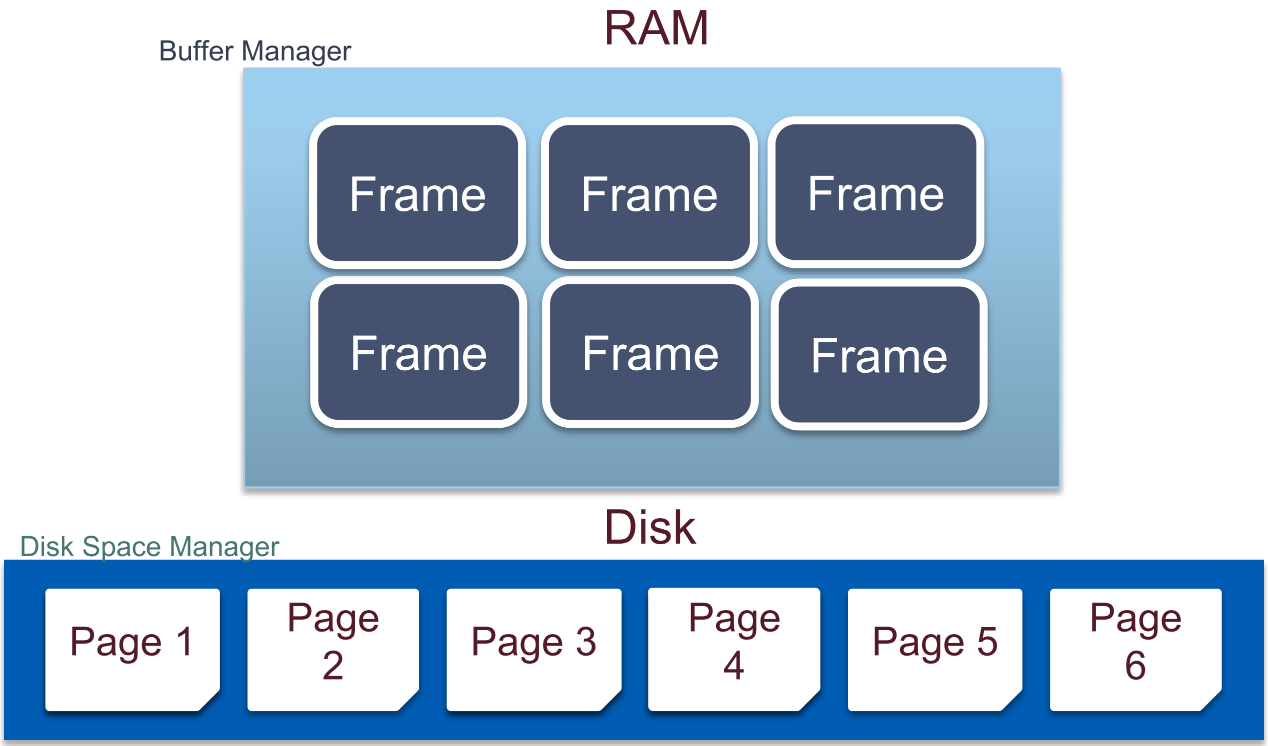
RAM

Buffer Management

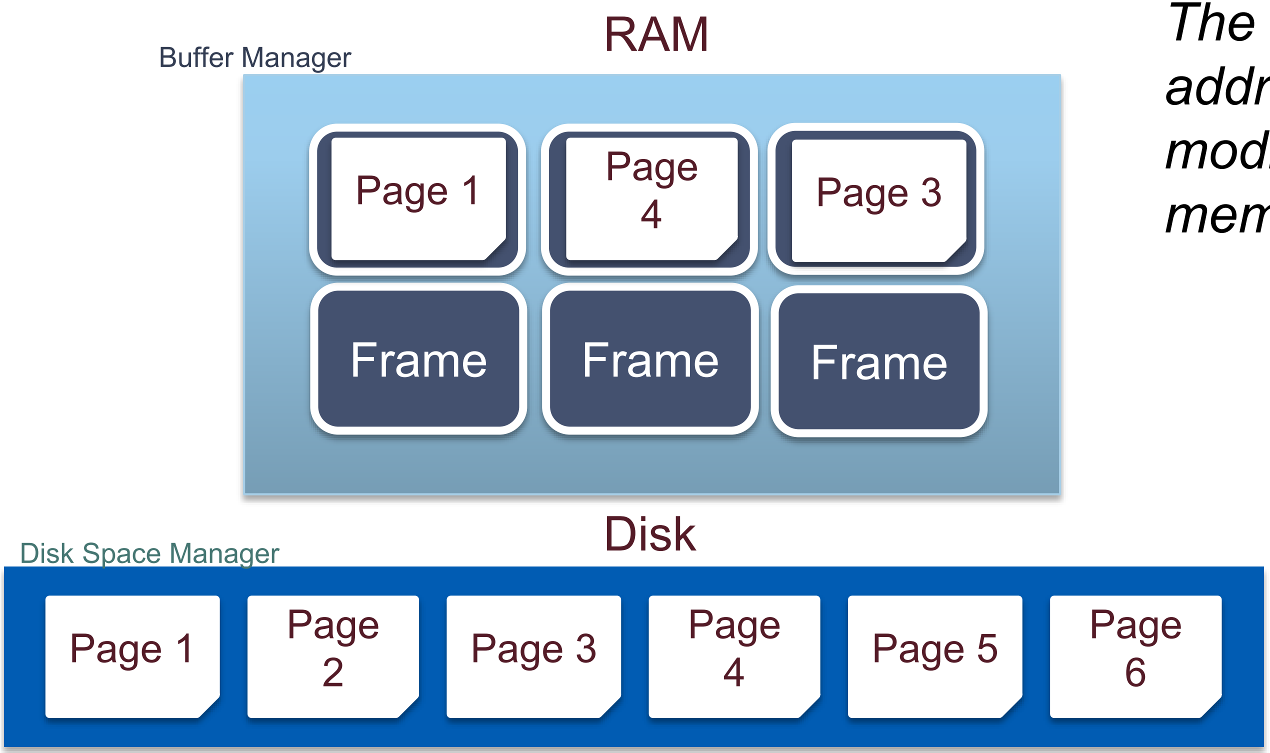
Disk

Disk Space Management

Buffer Management, cont

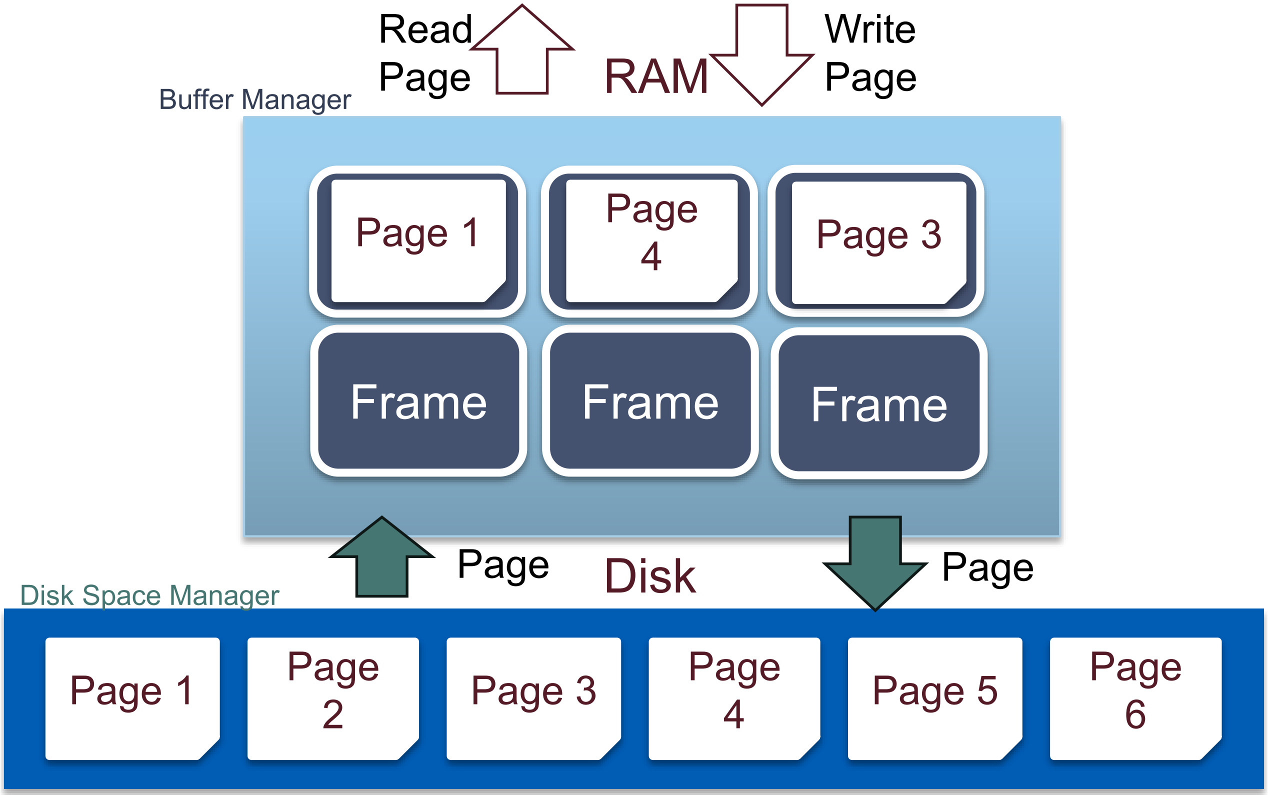


Buffer Management Read

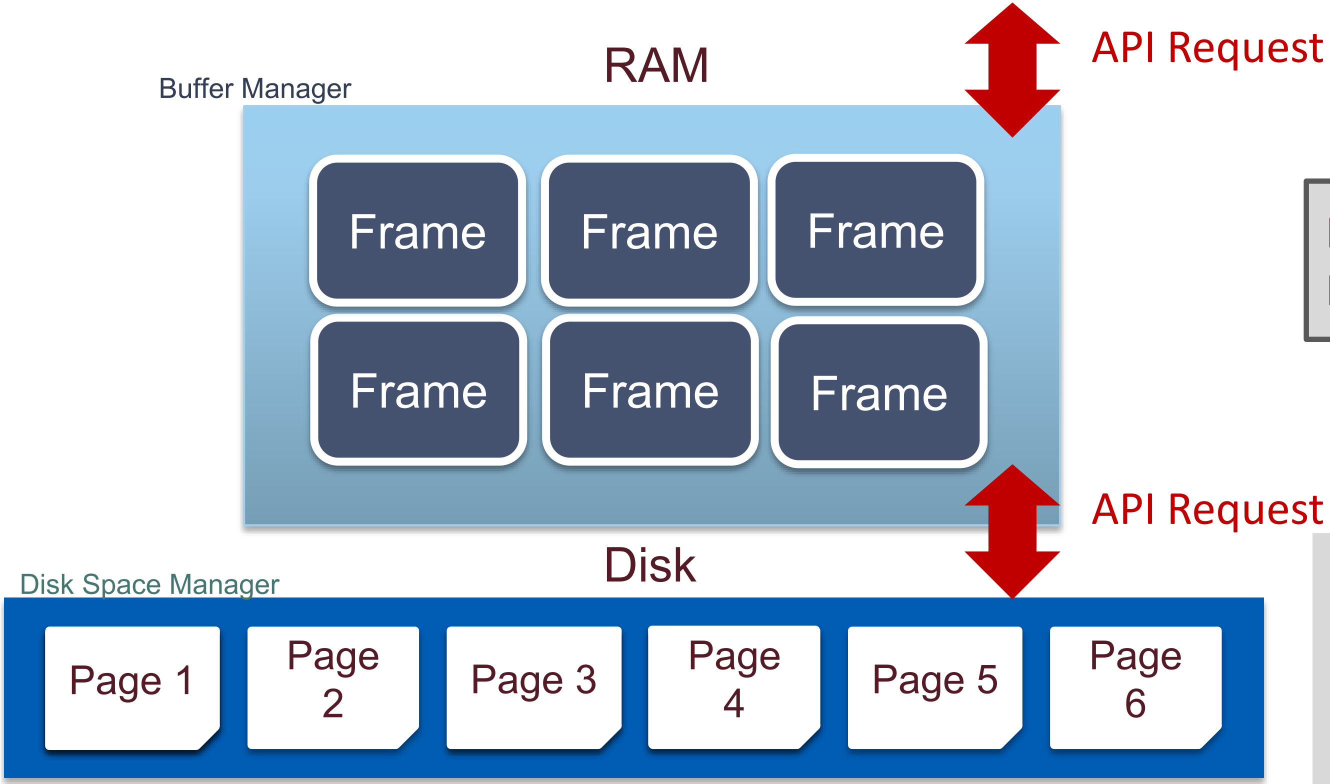


The illusion of addressing and modifying disk pages in memory.

APIs

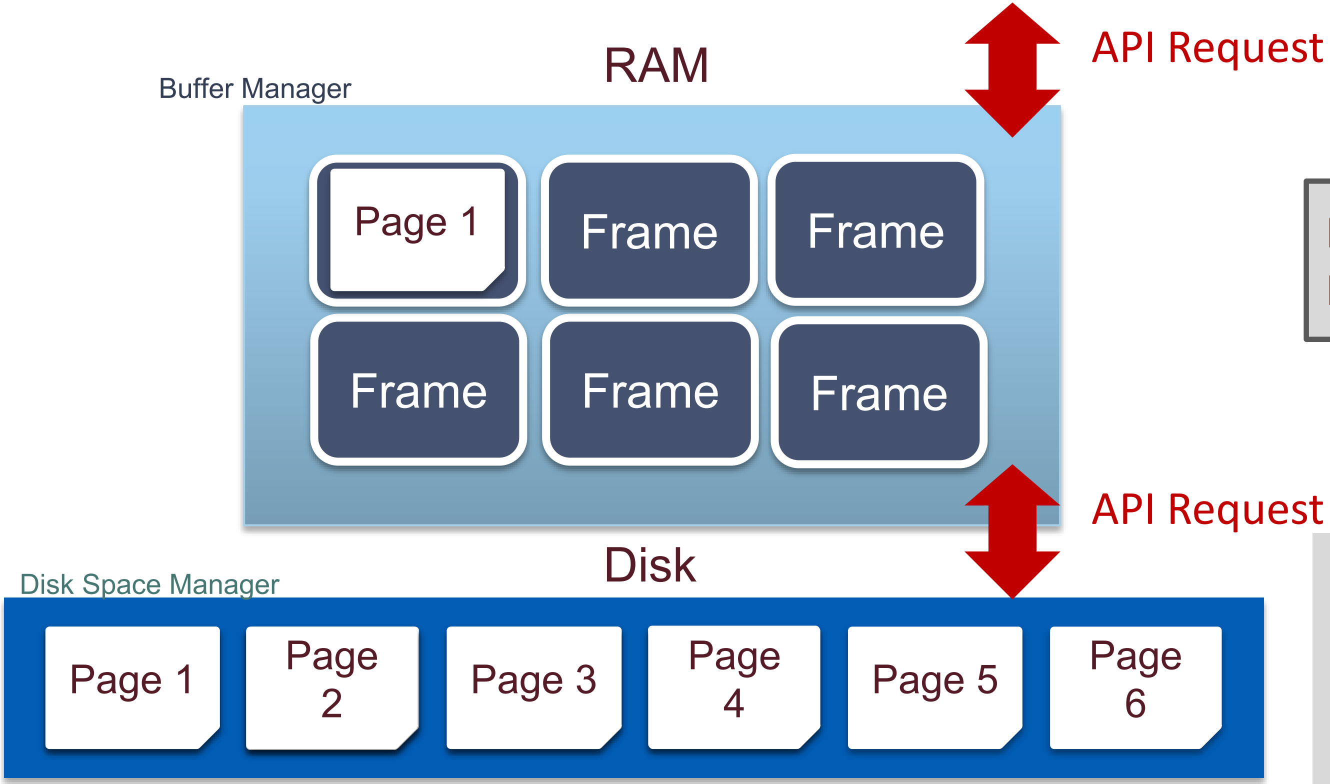

Read Page
RAM
Write Page
Buffer Manager
Page 1
Page 4
Page 3
Frame
Frame
Frame
Page
Disk
Page
Disk Space Manager
Page 1
Page 2
Page 3
Page 4
Page 5
Page 6

Mapping Pages Into Memory, Pt 1



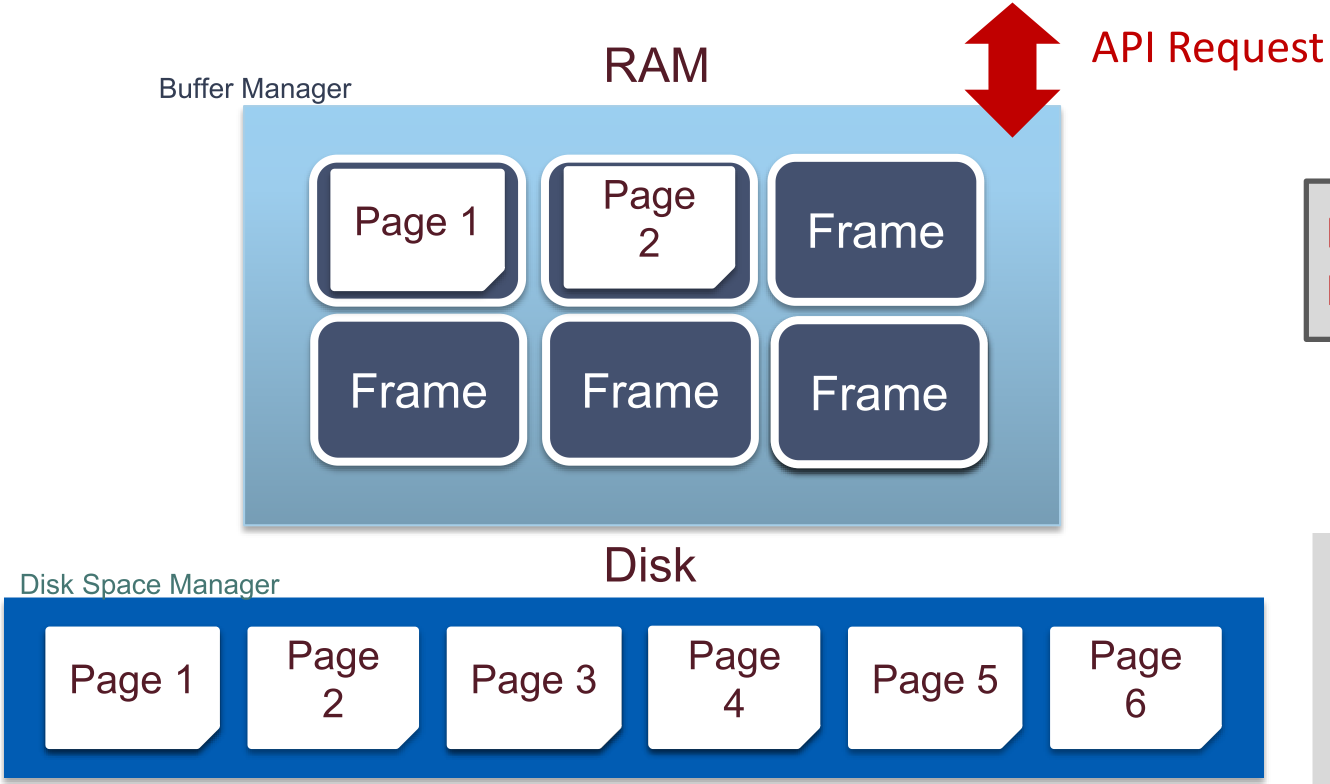
Message:
Request: Read page 1

Mapping Pages Into Memory, Pt 2



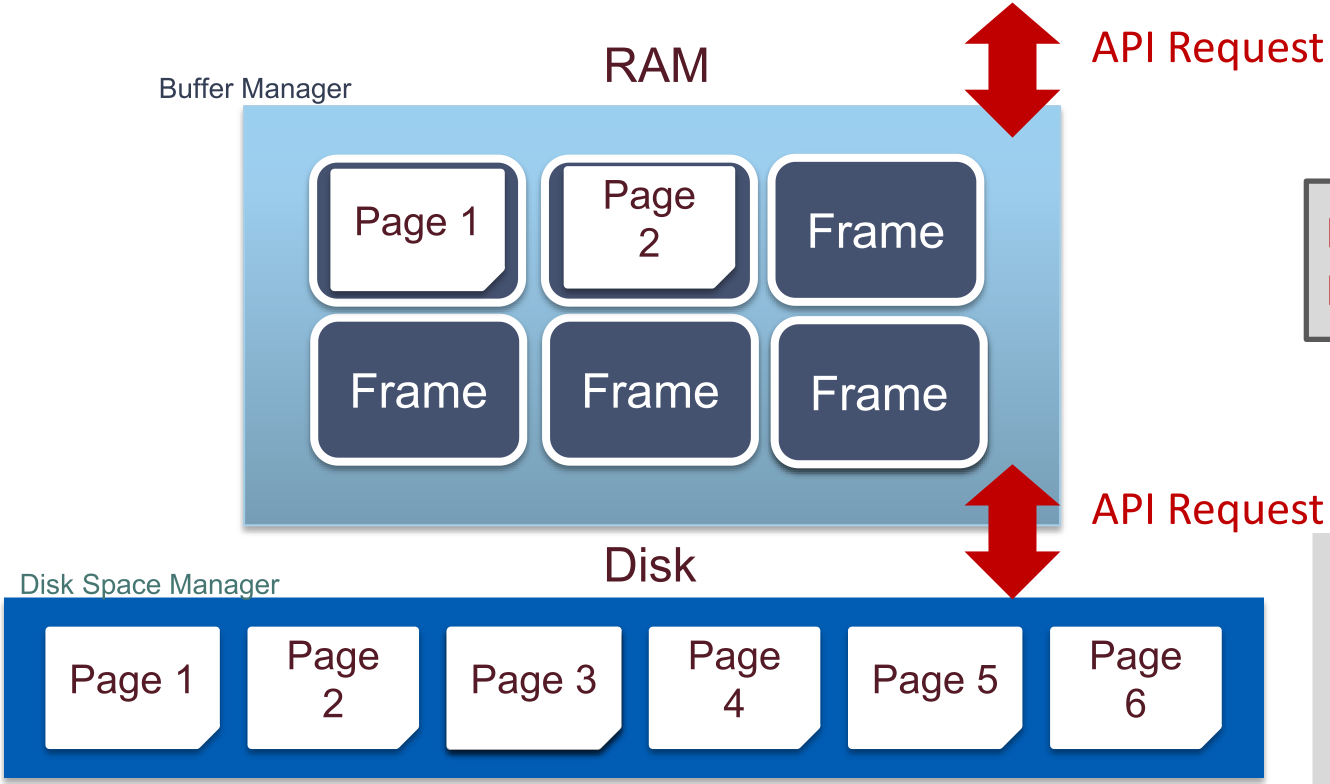
Message:
Request: Read page 2

Mapping Pages Into Memory, Pt 3



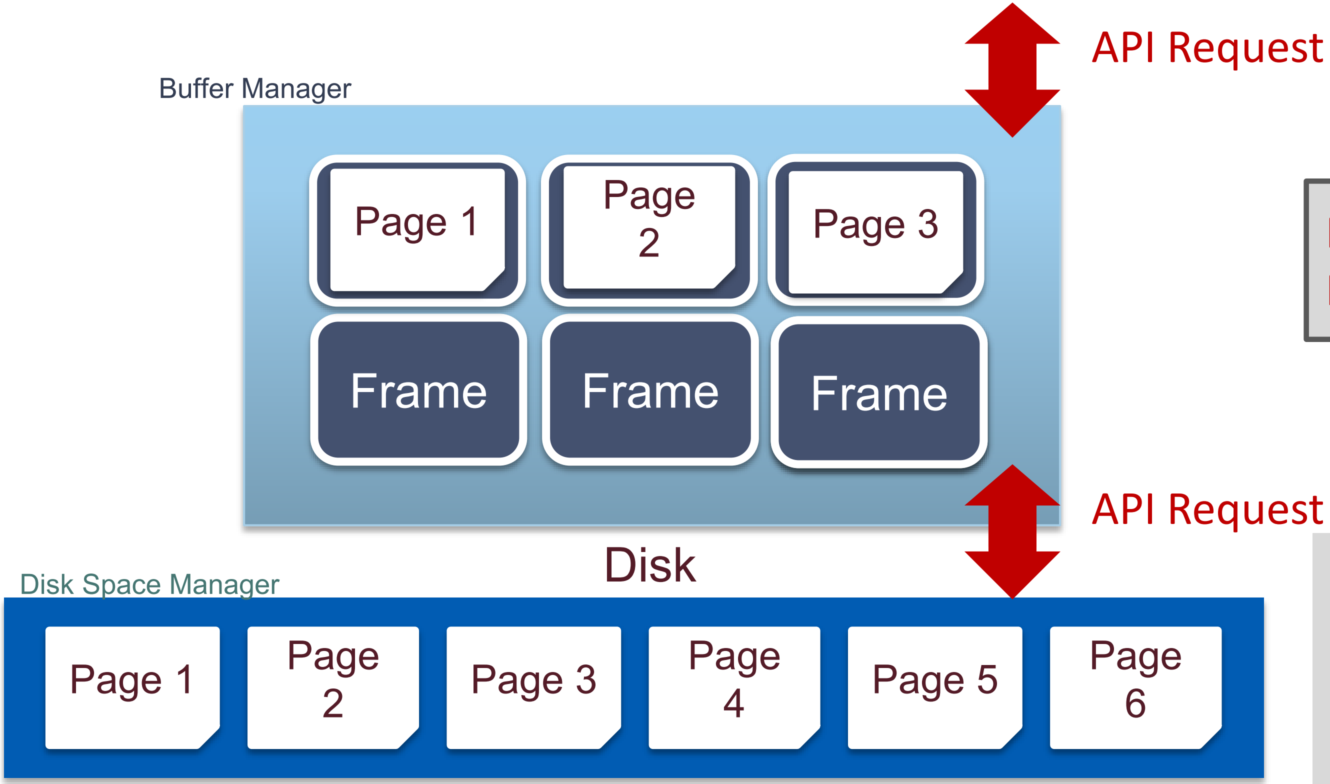
Message:
Request: Read page 2

Mapping Pages Into Memory, Pt 4



Message:
Request: Read page 3

Mapping Pages Into Memory



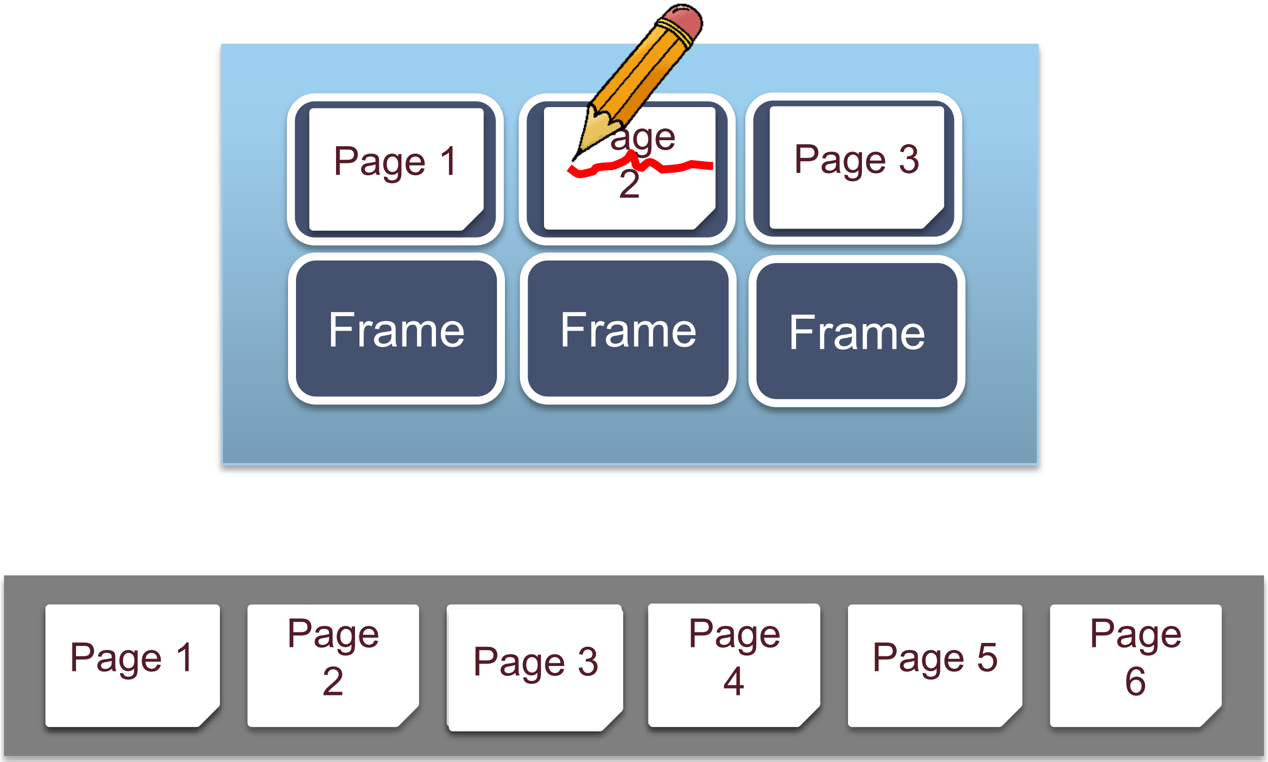
Message:
Request: Read page 3

Questions We Need to Answer

1. Handling dirty pages

2. Page Replacement

Q1: Dirty Pages?

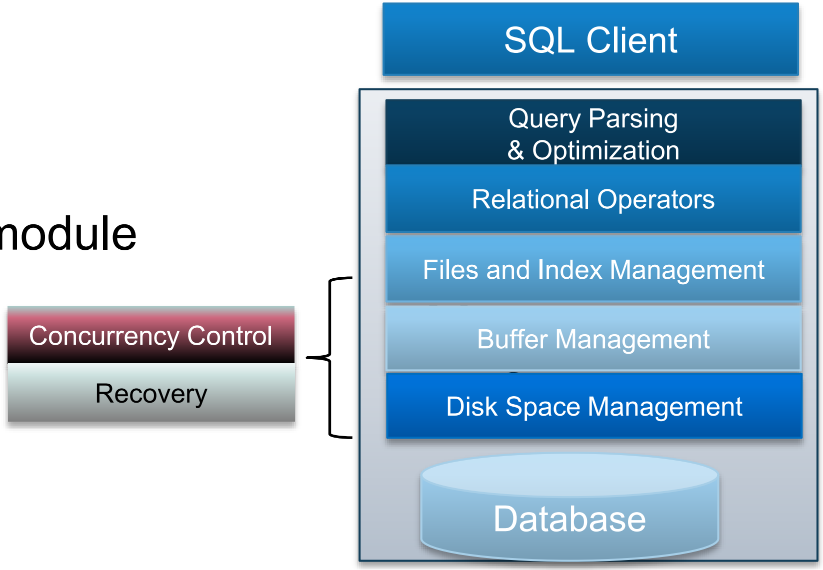


Handling Dirty Pages

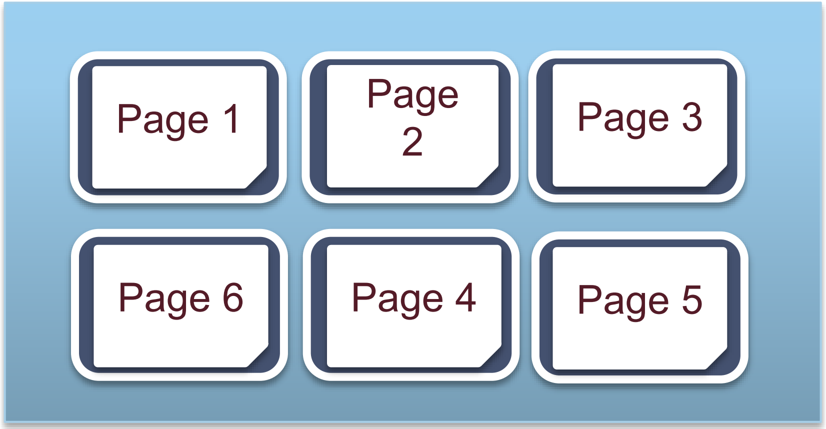
- Handling dirty pages
  - How will the buffer manager find out?
    - Dirty bit on page
  - What to do with a dirty page?
    - Write back via disk manager

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)

Frame	Frame	Frame	Frame	Frame	Frame

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: Smallish array in memory, malloc'ed at DBMS server boot time

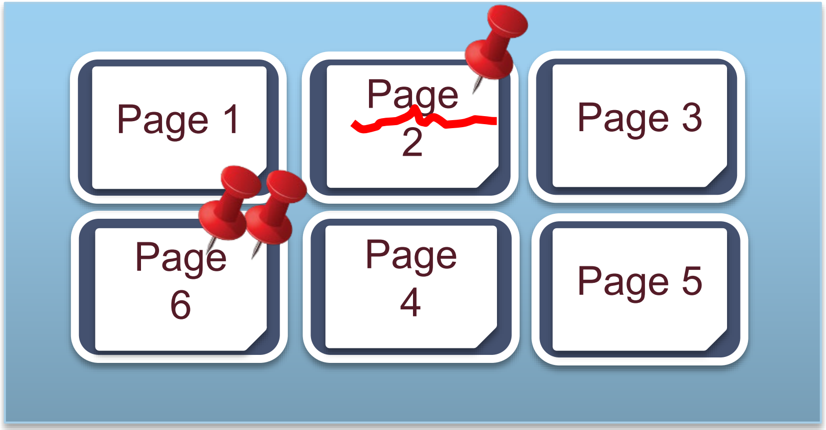
FrameId	PageId	Dirty?	Pin Count
1	1	N	0
2	2	Y	1
3	3	N	0
4	6	N	2
5	4	N	0
6	5	N	0

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

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

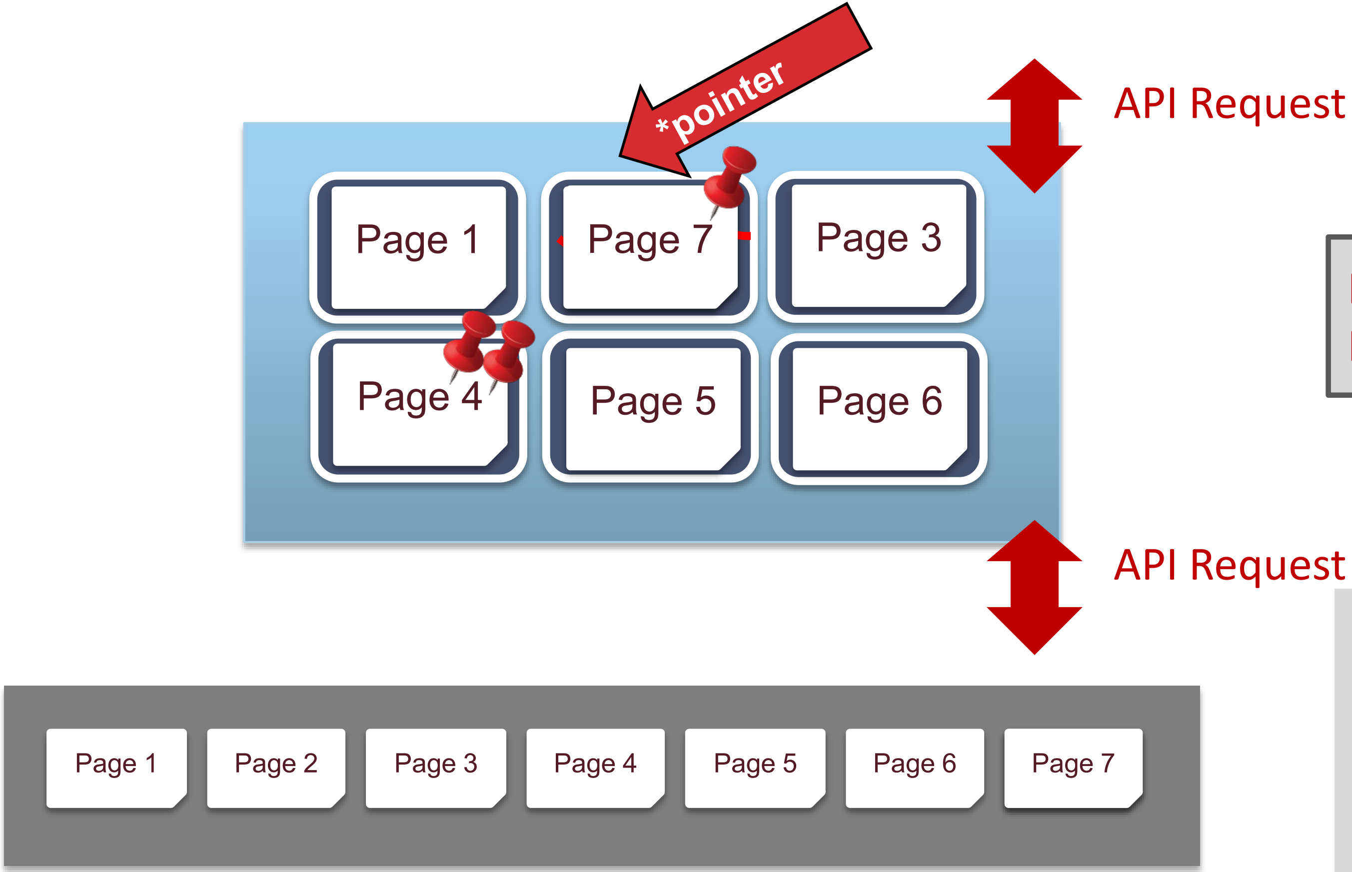
When a Page is Requested …

1. If requested page is not in pool:
   a. Choose an **un-pinned** (pin_count = 0) frame for replacement.
   b. If frame "dirty", write current page to disk, mark "clean"
   c. Read requested page into frame
2. Pin the page and return its address

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

- several pages at a time!

Q2: Page Replacement



Message:
Request: Read page 7

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
  - Depends on the **access pattern.**

LRU Replacement Policy

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

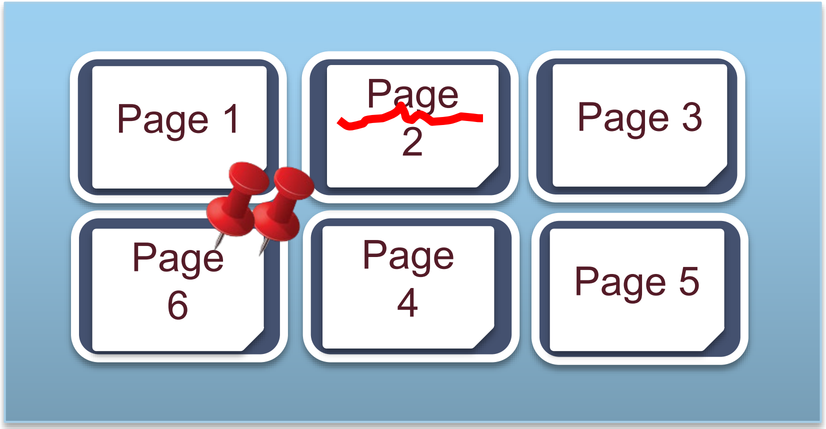
FrameId	PageId	Dirty?	Pin Count	Last Used
1	1	N	0	43
2	2	Y	1	21
3	3	N	0	22
4	6	N	2	11
5	4	N	0	24
6	5	N	0	15

LRU Replacement Policy, Pt 2

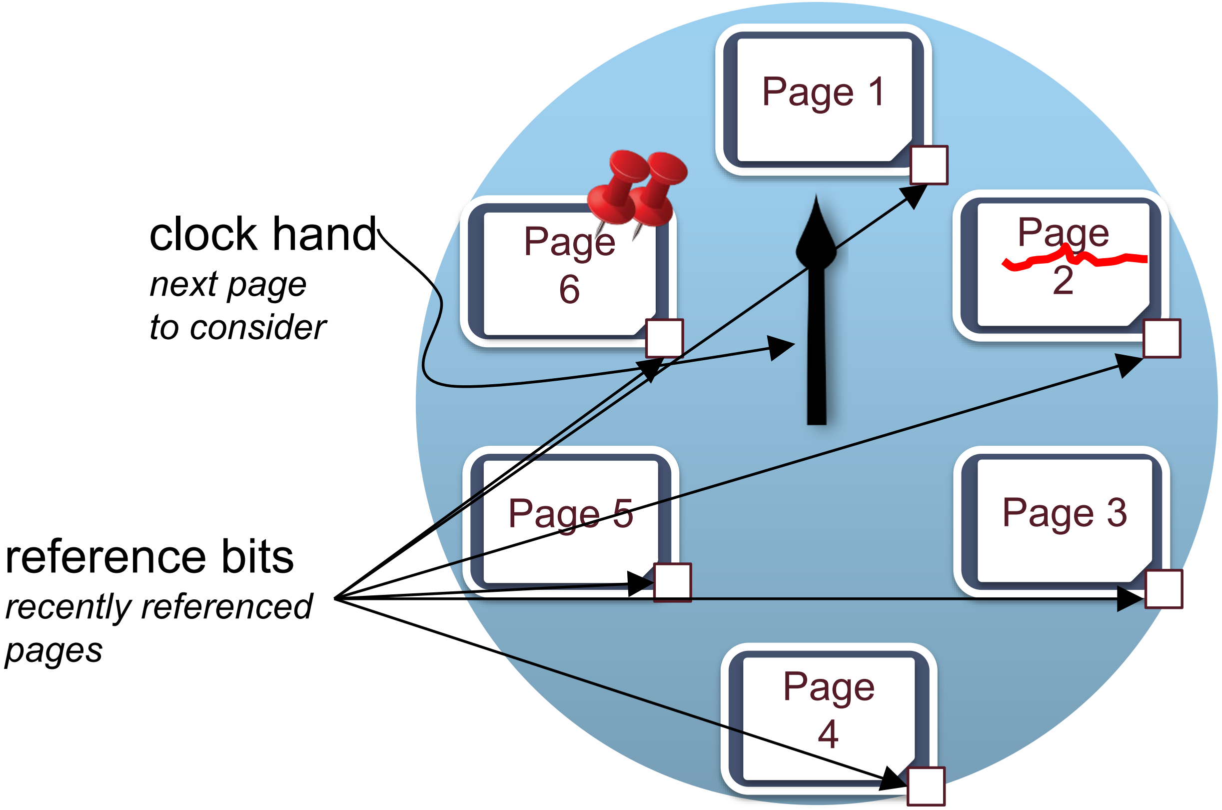
- Very common policy: intuitive and simple
  - Good for repeated accesses to popular pages (temporal locality)
  - Can be costly. Why?
    - Need to "find min" on the last used attribute (priority heap data structure)
- Approximate LRU: CLOCK policy

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

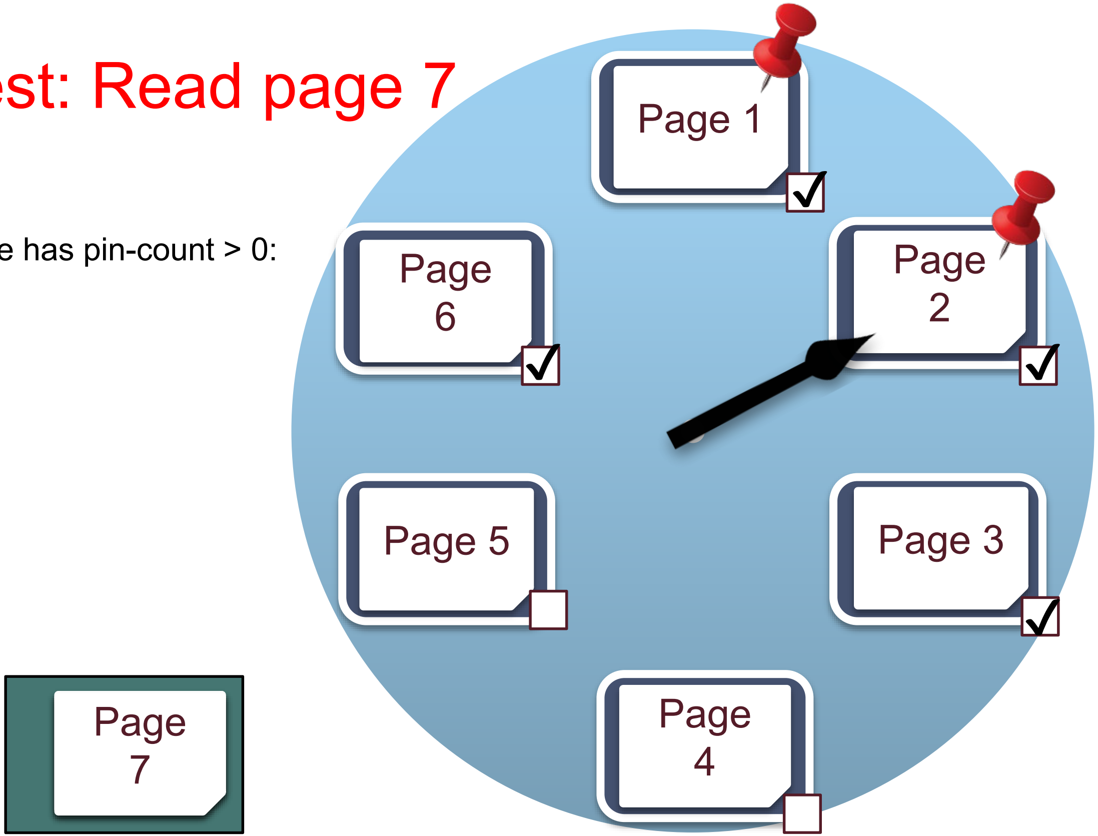
FrameId	PageId	Dirty?	Pin Count	Ref Bit
1	1	N	1	1
2	2	N	1	1
3	3	N	0	1
4	4	N	0	0
5	5	N	0	0
6	6	N	0	1

Clock Hand
1

Clock Policy State: Illustrated Part 1

Request: Read page 7

Current frame has pin-count > 0:
Skip



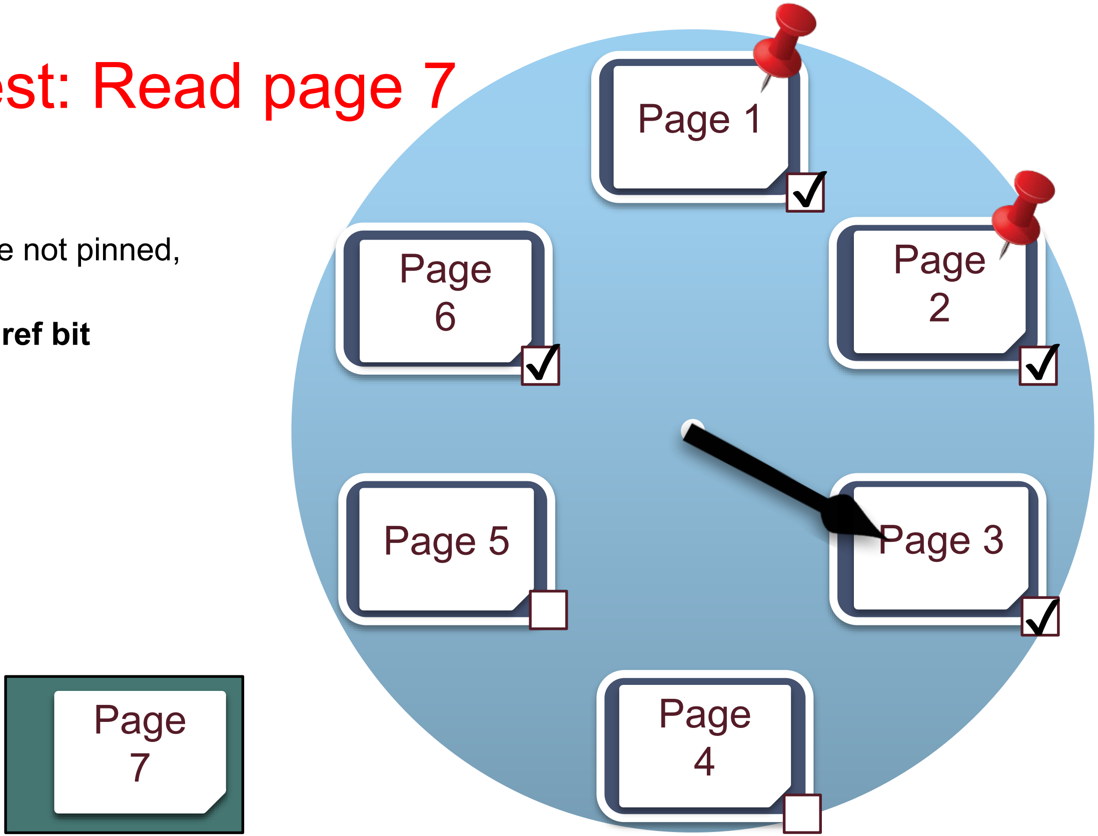
Clock Policy State: Illustrated, Part 2

Request: Read page 7

Current frame not pinned,
Ref bit set:

Clear ref bit

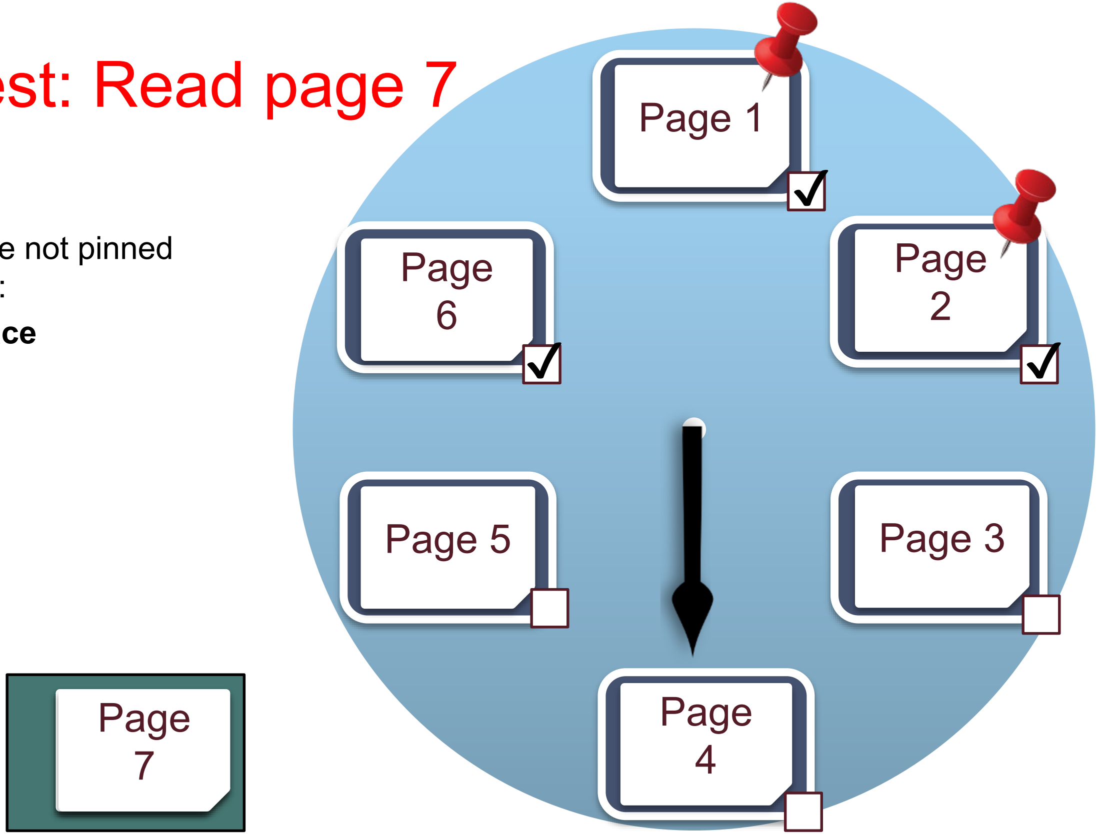
Skip



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

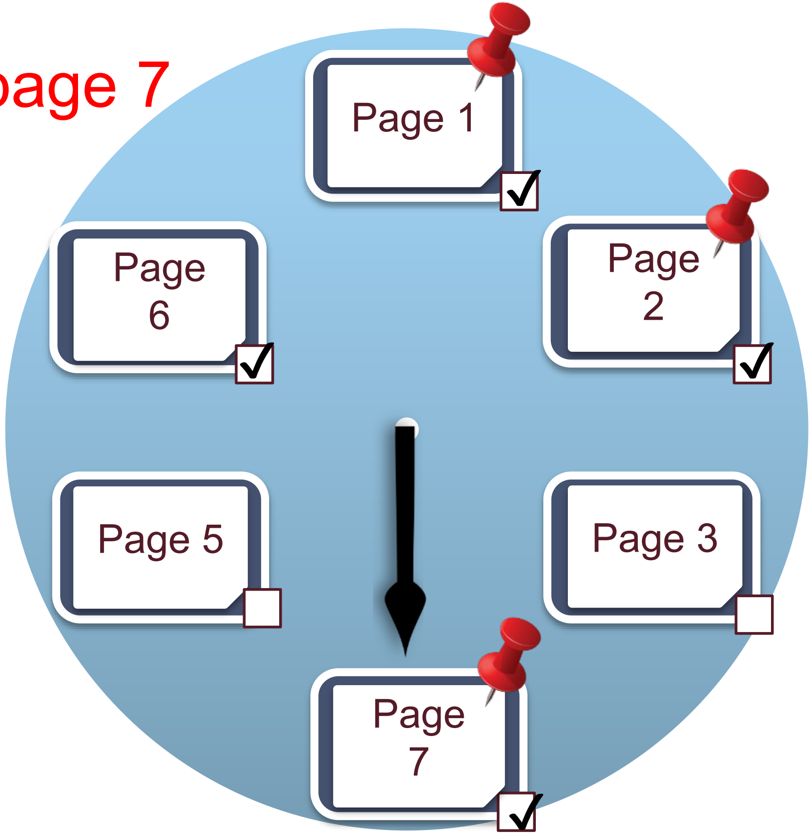
Current frame not pinned
Ref bit unset:

Replace

Set pinned

Set ref bit

Advance clock

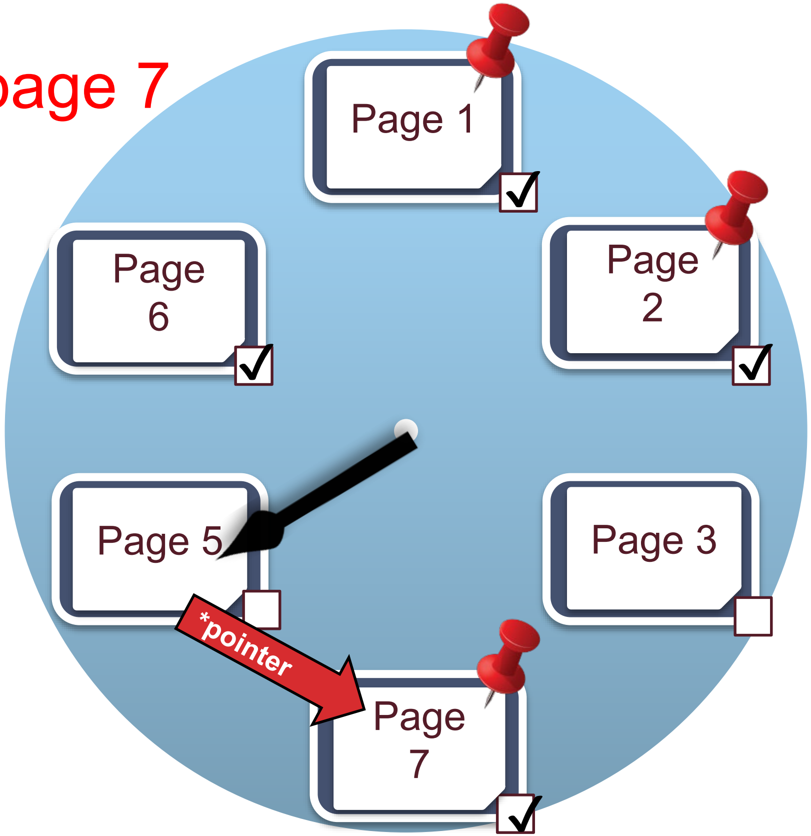


Clock Policy State: Illustrated, Pt 5

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

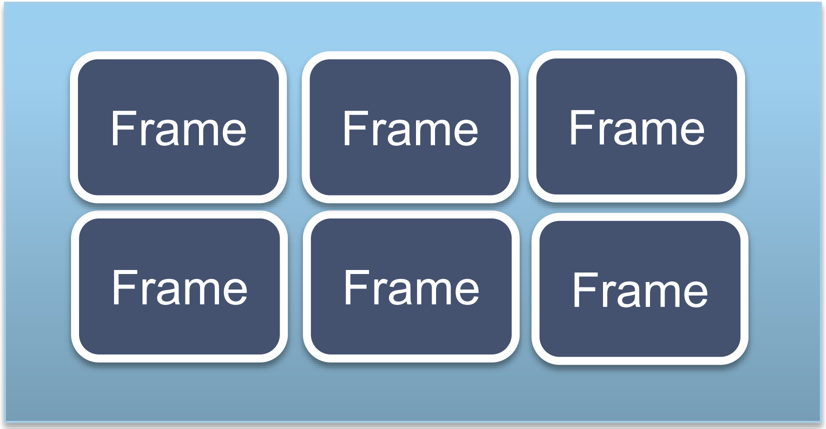
```
page *clock_request_page(int &clk_hand, int pg_num) {
  retval = NULL;
  while (retval == NULL) {
    current = frame_table[clk_hand];
    // the happy case: replace current page
    if (current.pin_count == 0 && current.refbit == 0) {
      if (current.dirty == 1)
        write_page(fi.page, frames[clk_hand]);
      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)

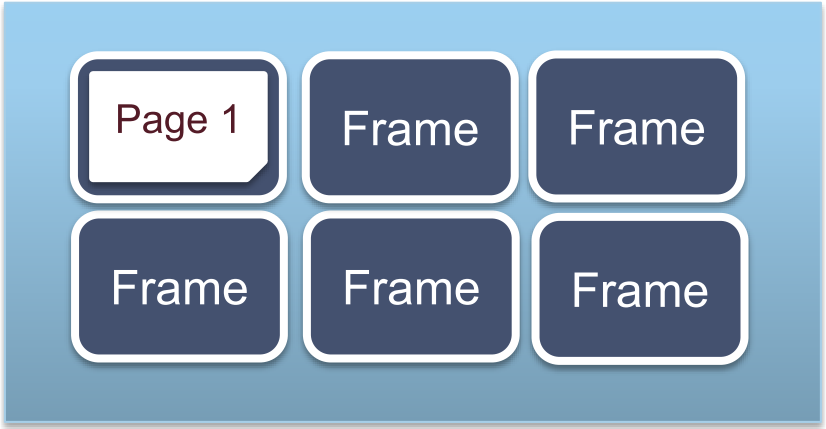
- Cache Hits: 0
- Attempts: 0



Disk Space Manager

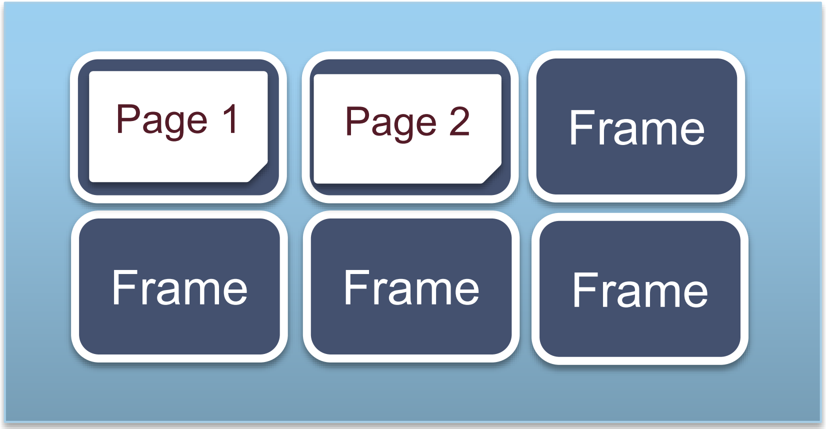
Repeated Scan (LRU): Read Page 1

- Cache Hits: 0
- Attempts: 1

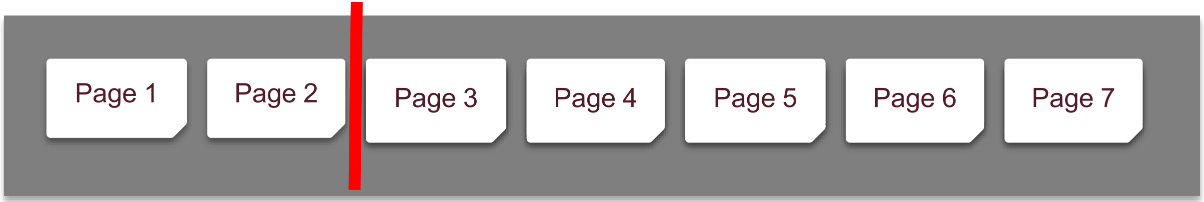


Repeated Scan (LRU): Read Page 2

- Cache Hits: 0
- Attempts: 2

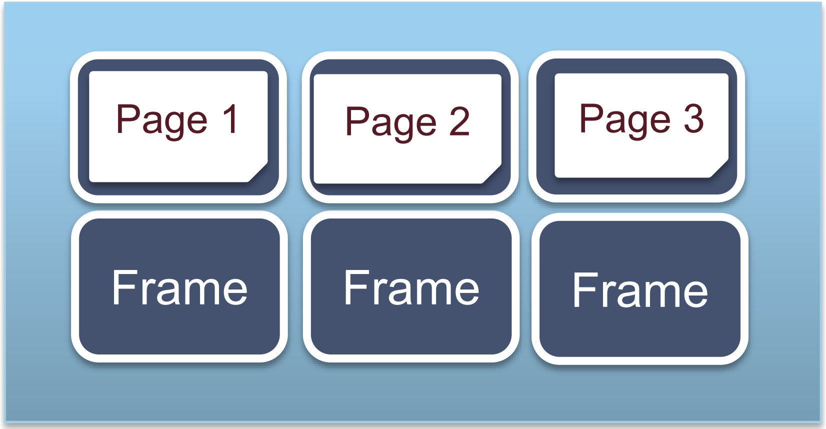


Disk Space Manager

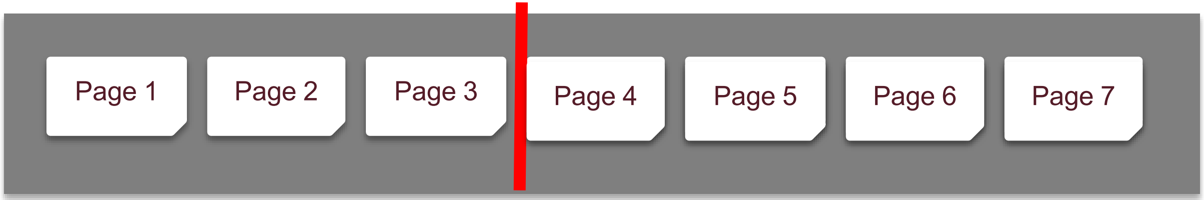


Repeated Scan (LRU): Read Page 3

- Cache Hits: 0
- Attempts 3:

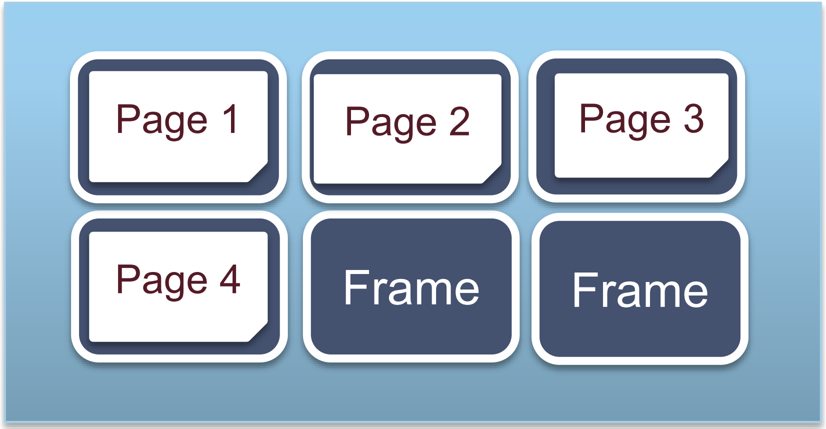


Disk Space Manager



Repeated Scan (LRU): Read Page 4

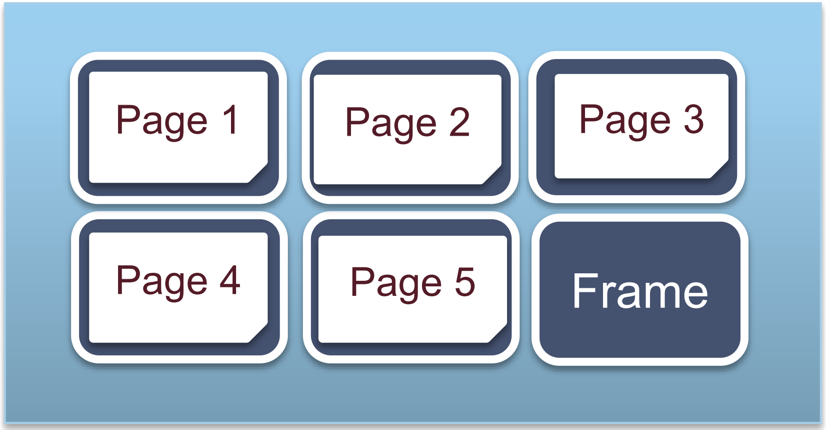
- Cache Hits 0:
- Attempts: 4



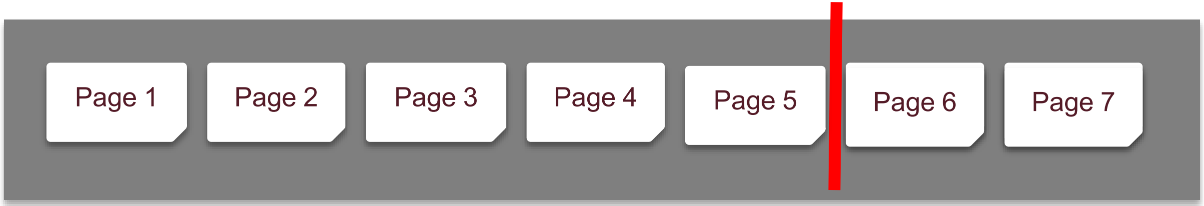
Disk Space Manager

Repeated Scan (LRU): Read Page 5

- Cache Hits: 0
- Attempts: 5

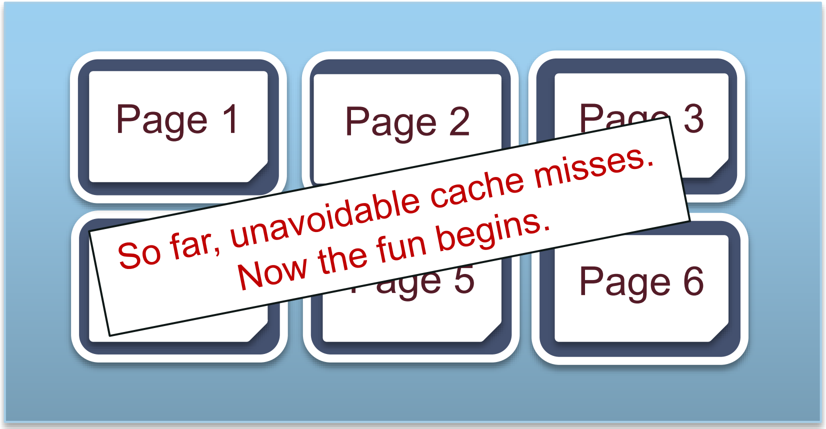


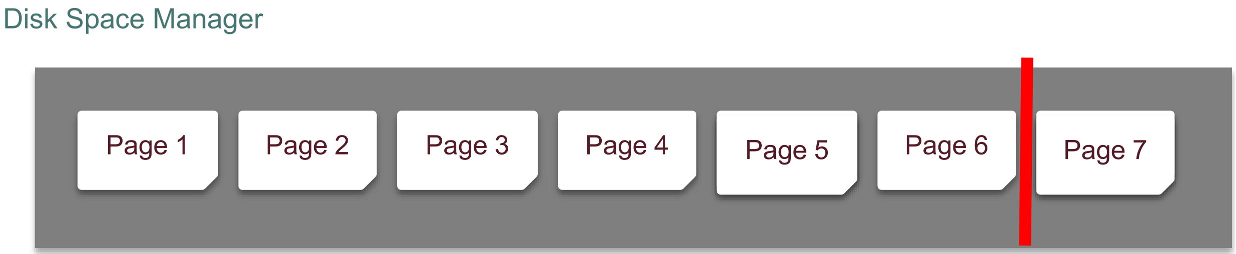
Disk Space Manager



Repeated Scan (LRU): Read Page 6

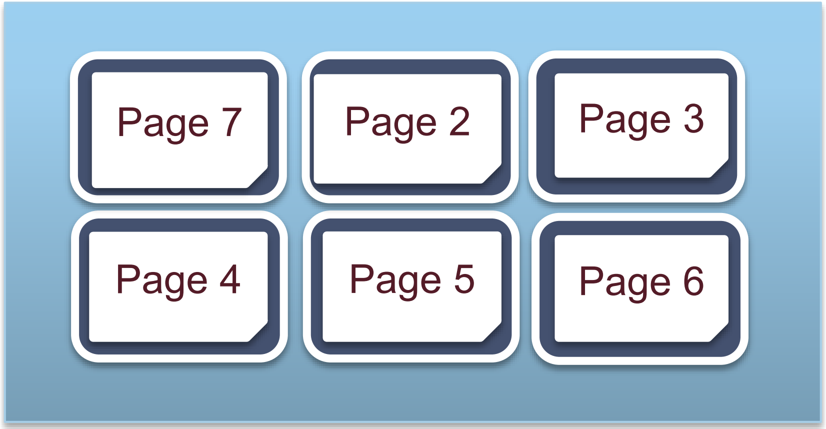
- Cache Hits: 0
- Attempts 6



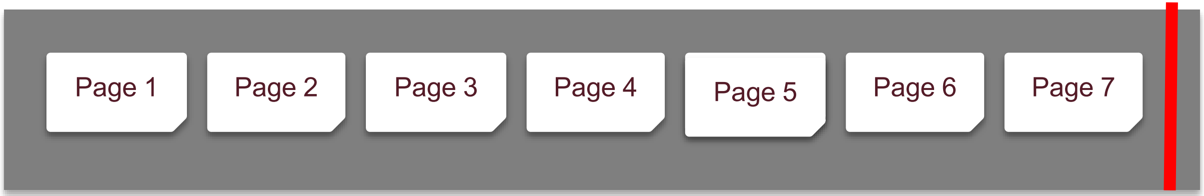


Repeated Scan (LRU): Read Page 7

- Cache Hits: 0
- Attempts: 7

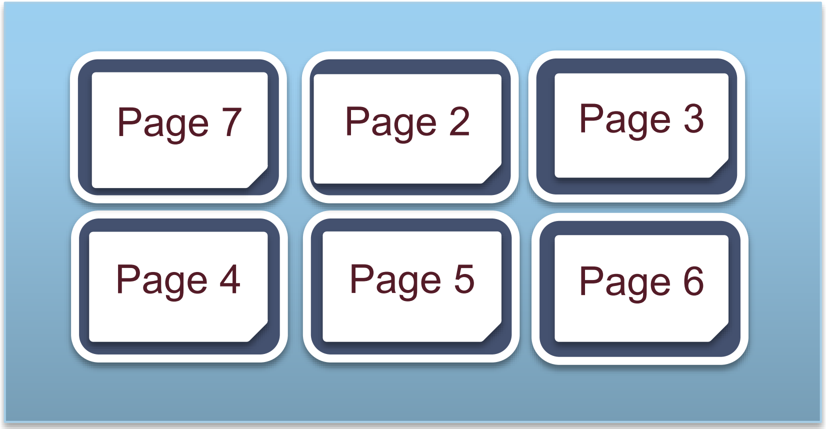


Disk Space Manager



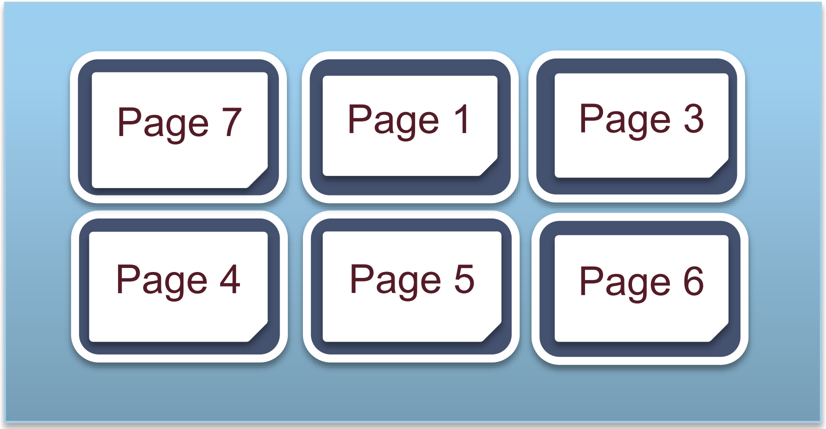
Repeated Scan (LRU): Reset to beginning

- Cache Hits: 0
- Attempts: 7



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

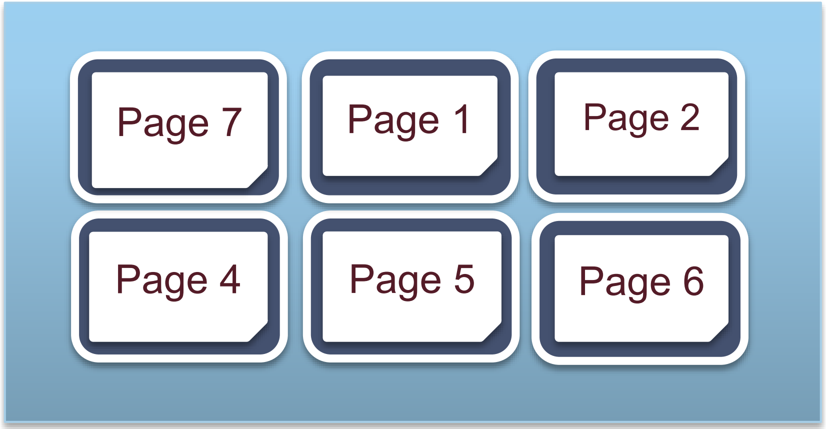
- Cache Hits: 0
- Attempts: 8



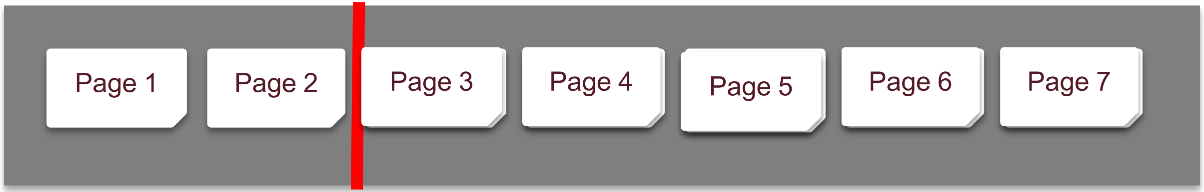
Disk Space Manager

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

- Cache Hits: 0
- Attempts: 9

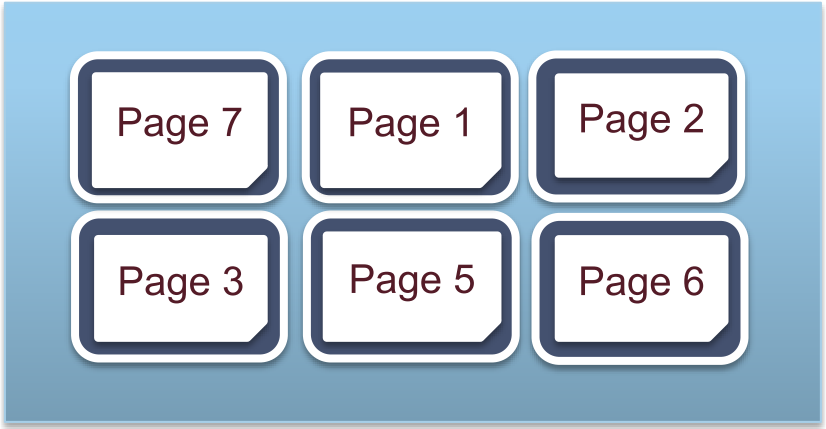


Disk Space Manager



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

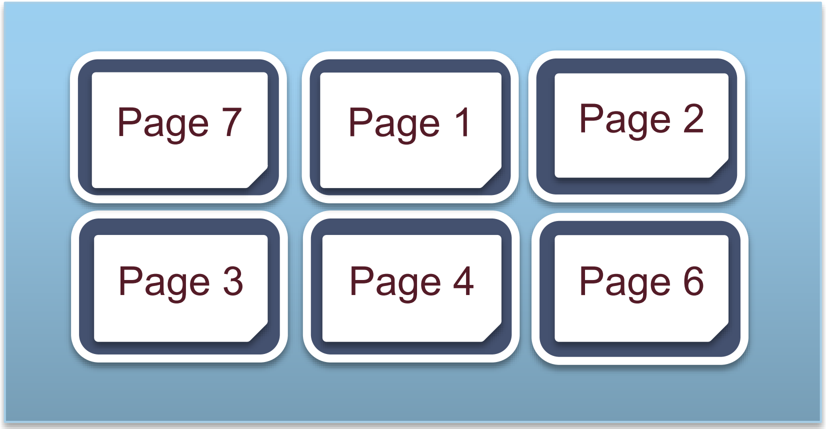
- Cache Hits: 0
- Attempts: 10



Disk Space Manager

Repeated Scan (LRU): Page 4 (again)

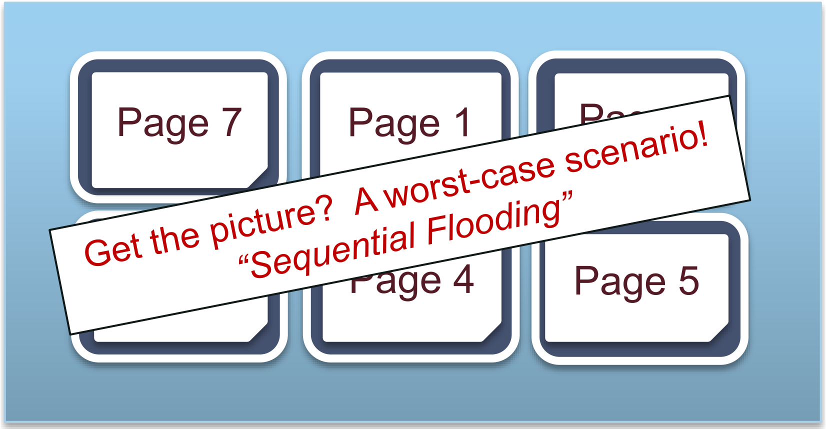
- Cache Hits: 0
- Attempts: 11



Disk Space Manager

Repeated Scan (LRU): Read Page 5, cont

- Cache Hits: 0
- Attempts: 12



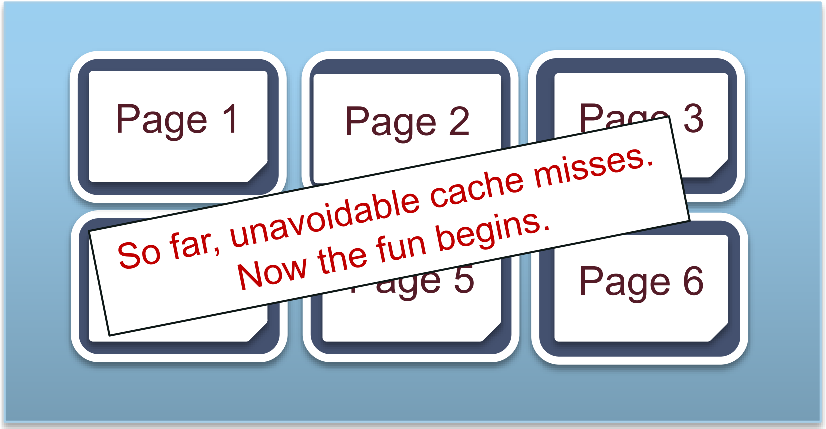
Disk Space Manager

Sequential Scan + LRU

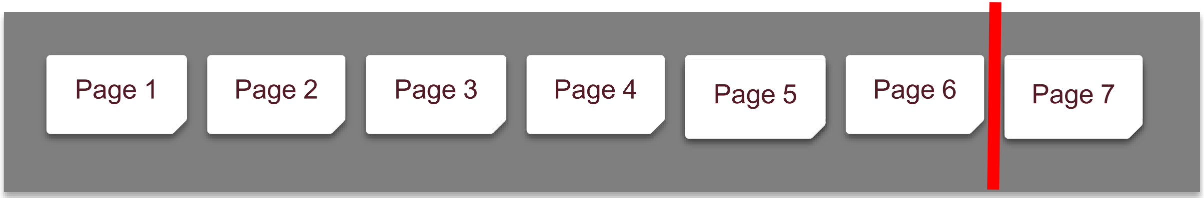
- Sequential flooding
- 0% hit rate in cache!
- Repeated sequential scan very common in database workloads
  - We will see it in nested-loops join
- What could be better?

Repeated Scan (MRU)

- Cache Hits: 0
- Attempts: 6

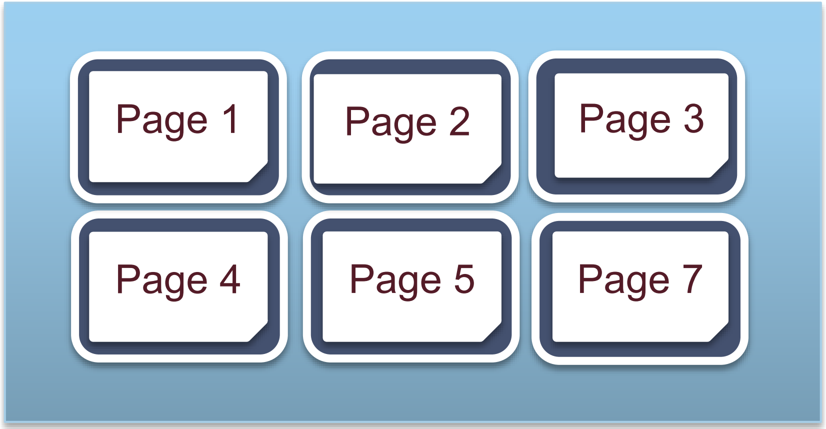


Disk Space Manager

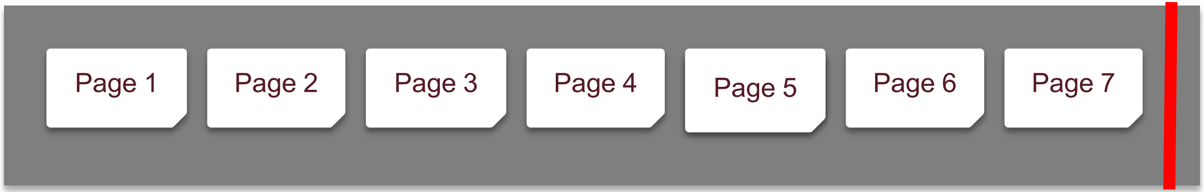


Repeated Scan (MRU): Read Page 7

- Cache Hits: 0
- Attempts: 7

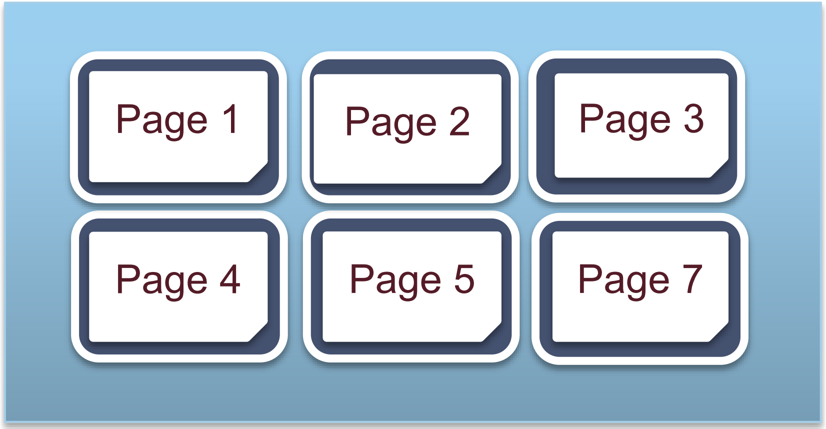


Disk Space Manager

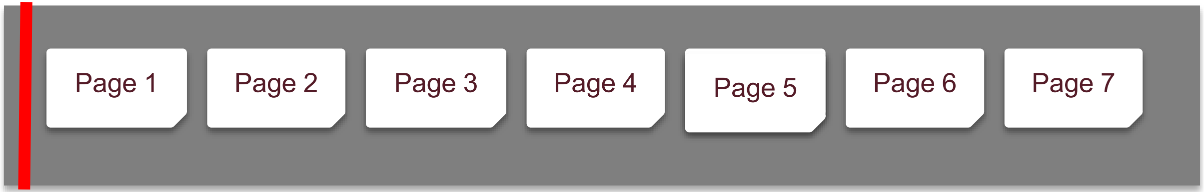


Repeated Scan (MRU): Reset

- Cache Hits: 0
- Attempts: 7

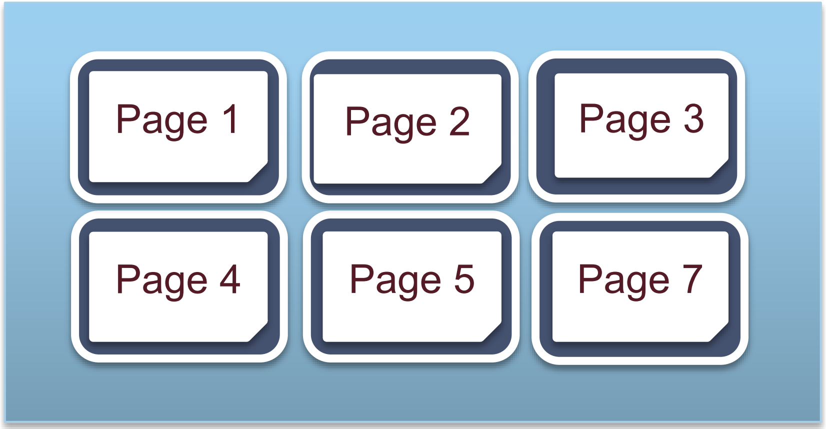


Disk Space Manager

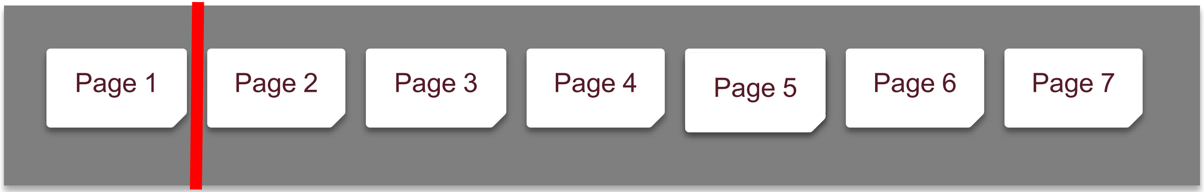


Repeated Scan (MRU): Read Page 1 (again)

- Cache Hits: 1
- Attempts: 8

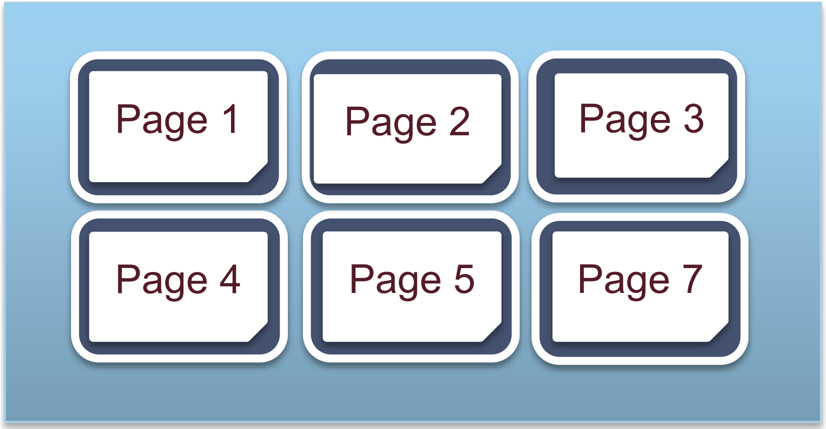


Disk Space Manager

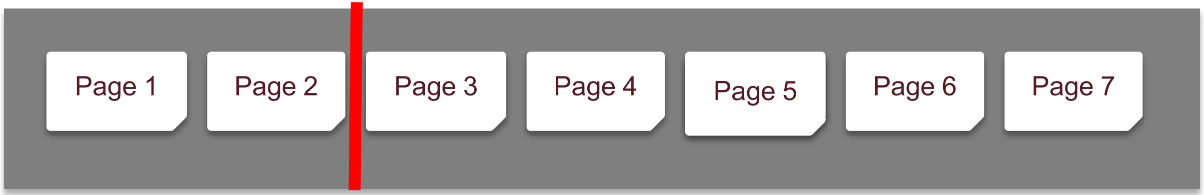


Repeated Scan (MRU): Read Page 2 (again)

- Cache Hits: 2
- Attempts: 9

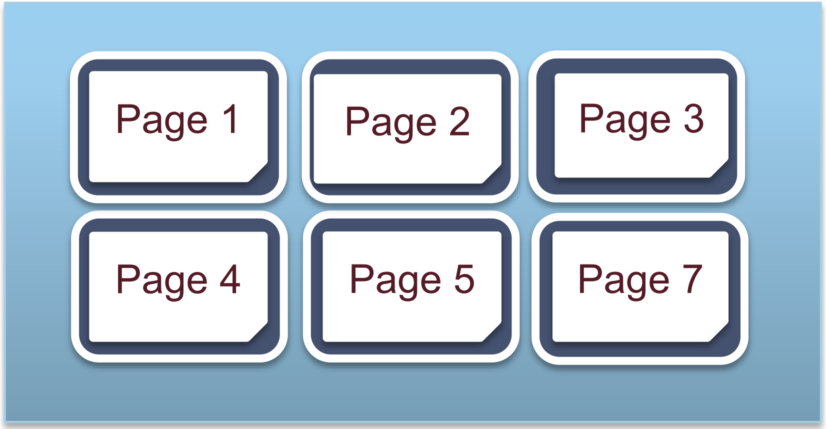


Disk Space Manager



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

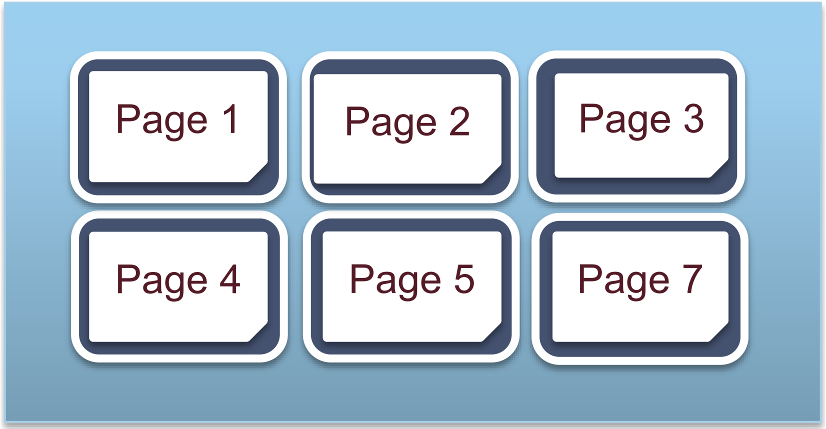
- Cache Hits: 3
- Attempts: 10



Disk Space Manager

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

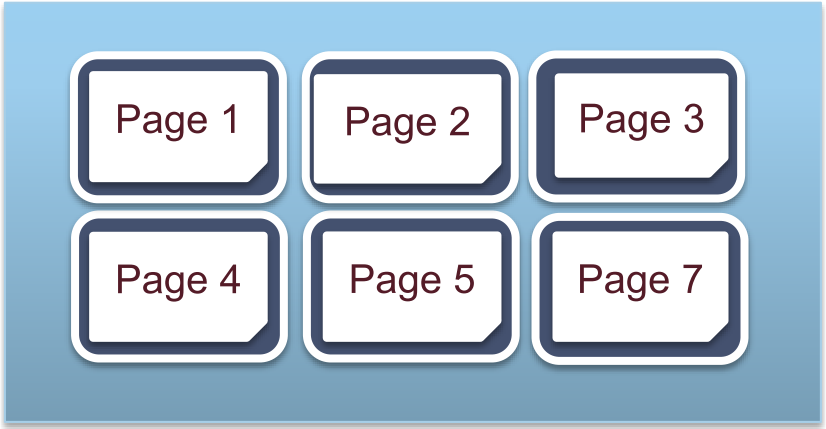
- Cache Hits: 4
- Attempts: 11



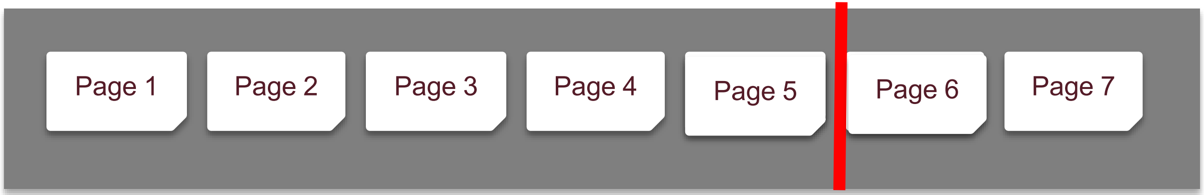
Disk Space Manager

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

- Cache Hits: 5
- Attempts: 12

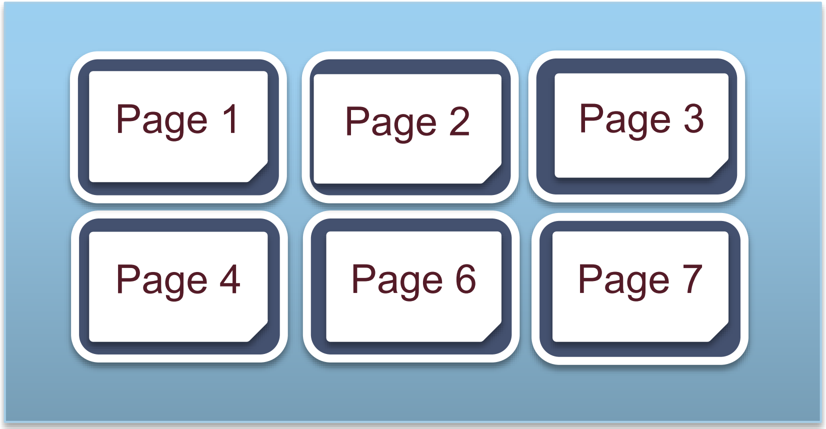


Disk Space Manager

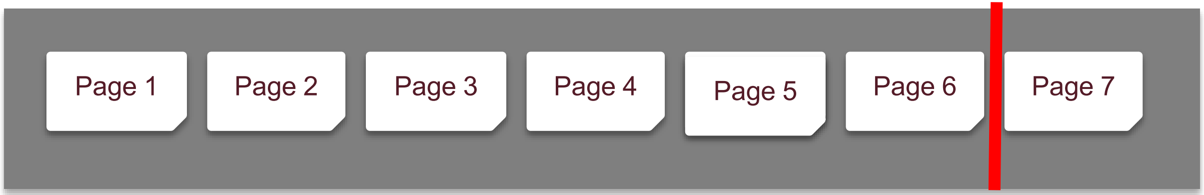


Repeated Scan (MRU): Read Page 6 (again)

- Cache Hits: 5
- Attempts: 13

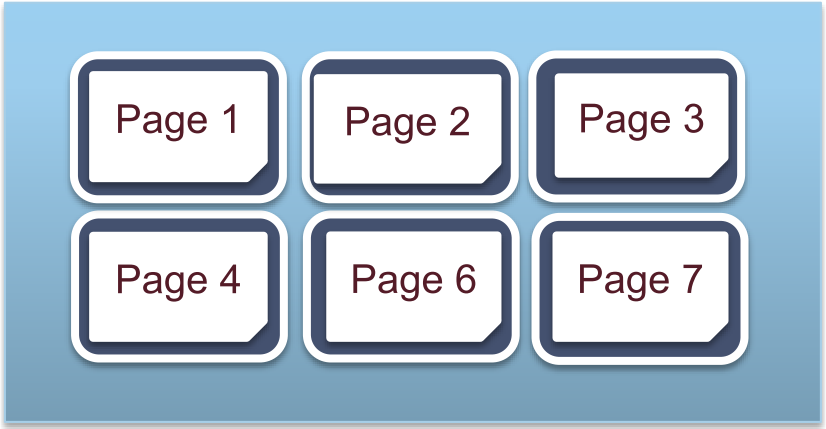


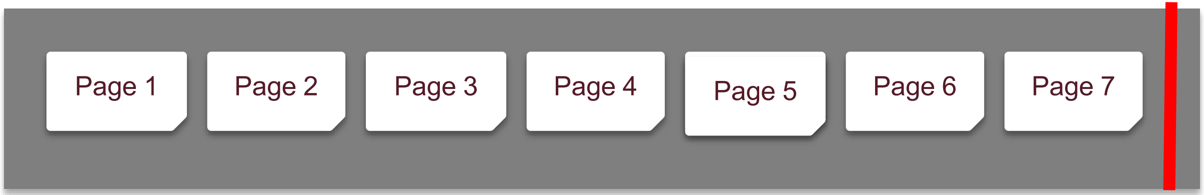
Disk Space Manager



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

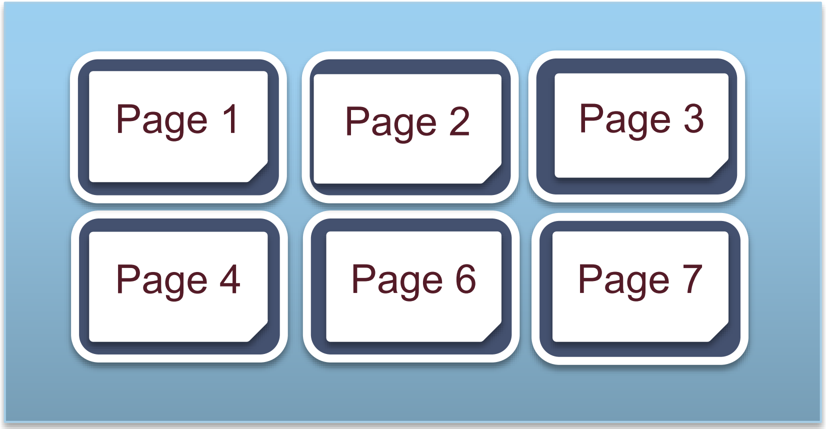
- Cache Hits: 6
- Attempts: 14

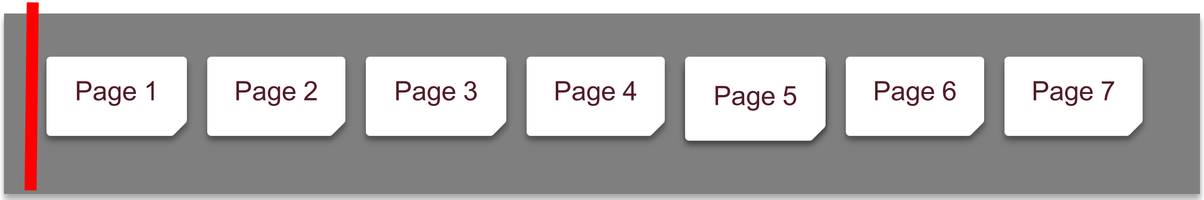




Repeated Scan (MRU): Reset (again)

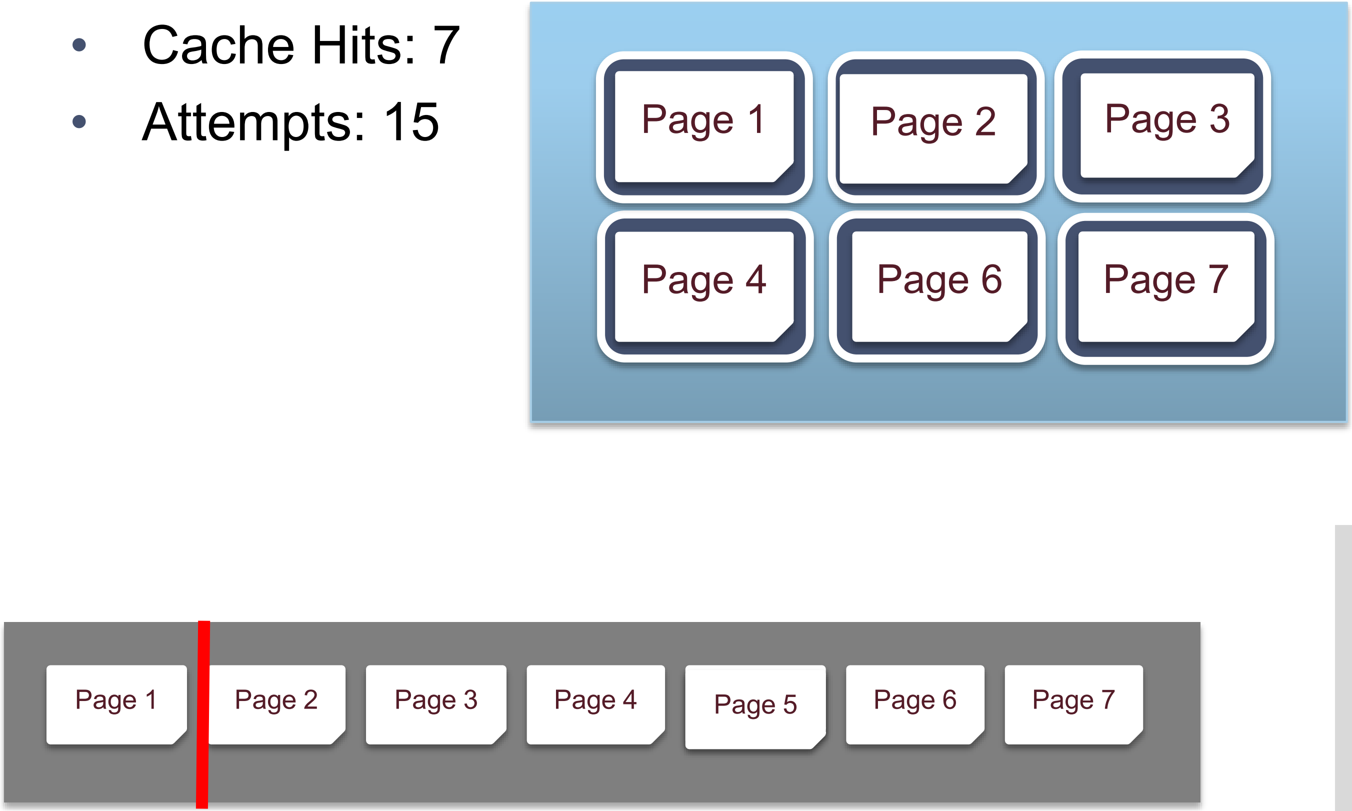
- Cache Hits: 6
- Attempts: 14





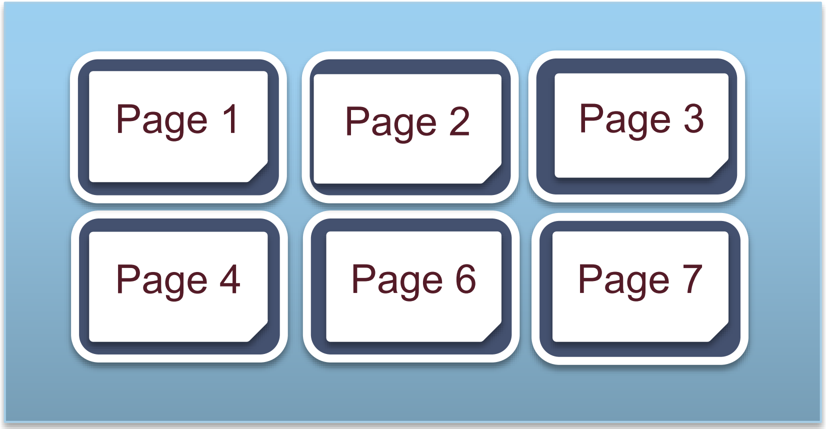
Repeated Scan (MRU): Read Page 1 (again x2)

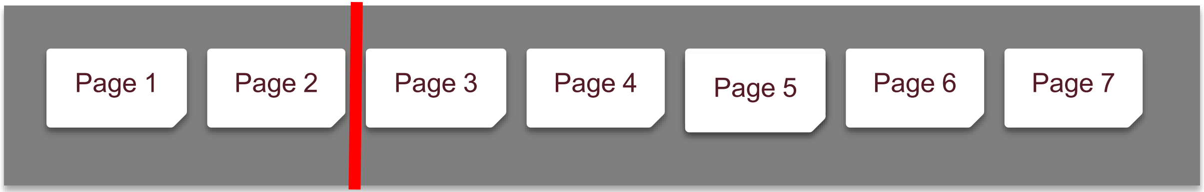
- Cache Hits: 7
- Attempts: 15



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

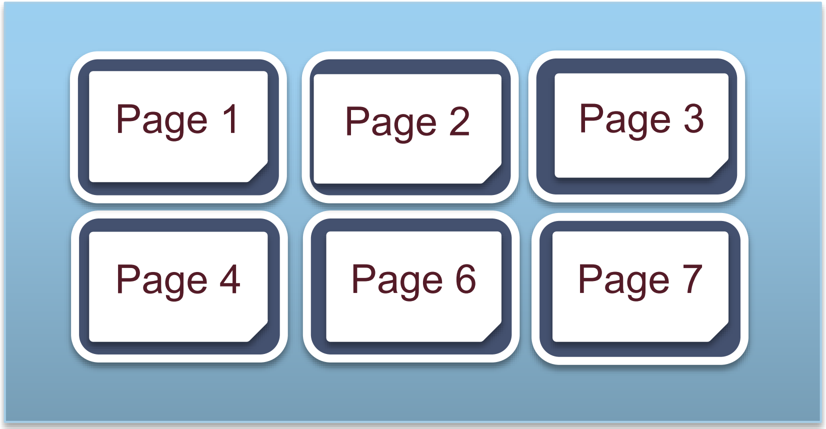
- Cache Hits: 8
- Attempts: 16





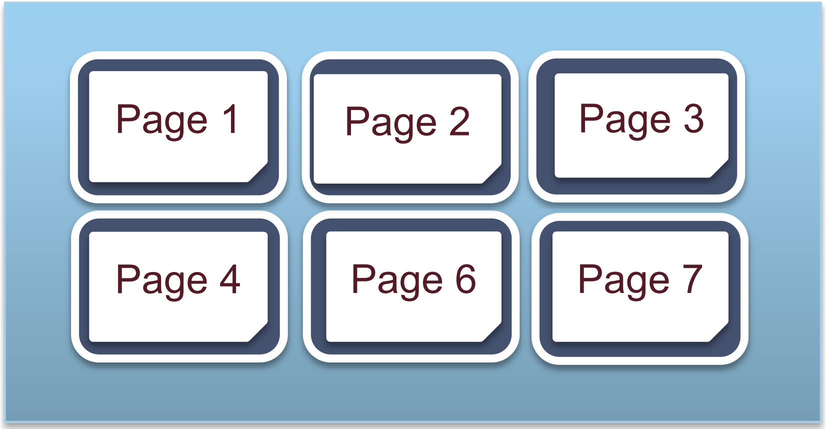
Repeated Scan (MRU): Read Page 3 (again x2)

- Cache Hits: 9
- Attempts: 17



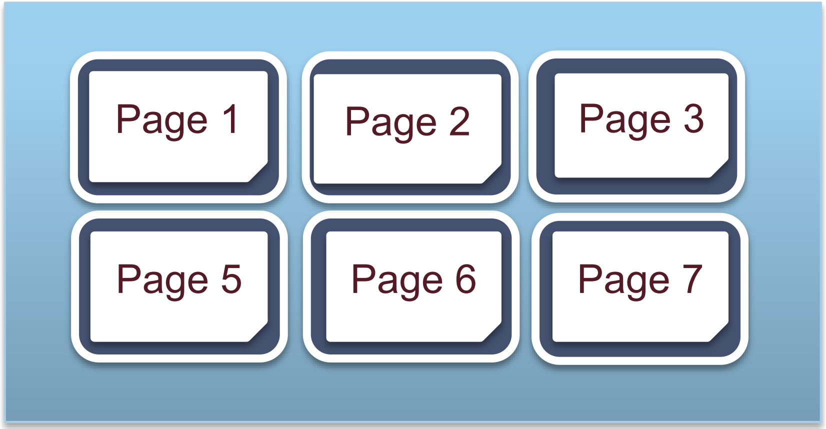
Repeated Scan (MRU): Read Page 4 (again x2)

- Cache Hits: 10
- Attempts: 18



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

- Cache Hits: 10
- Attempts: 19



General Case: SeqScan + MRU

B buffers

N > B pages in file

1. First N attempts: 0 hits
2. Next N attempts: B-1 hits
   Pages 1 through B-1
3. Next N attempts: B-1 hits
   Pages N through B-2
4. Next N attempts: B-1 hits
   Pages N-1 through B-3

   …

In limit: ~(B-1)/N 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 BufMgr
  - 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
- Find fancier stochastic policies
  - 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
  - E.g. special-case for indexes, use LRU-2 otherwise
  - FWIW, PostgreSQL currently uses CLOCK
  - Imagine workloads for a big cloud DBMS like AWS Aurora!

DBMS vs OS Buffer Cache

- Doesn't the filesystem (OS) manage buffers and pages too?

- Issues:
- Portability: different FS, different behavior
- OS limitations: DBMS requires ability to force pages to disk
  - Required for recovery, as we'll see
- OS limitations: DBMS can predict its own page reference patterns
  - E.g. consider scanning the leaves of a B+-tree
  - Affects both page replacement and prefetching

Summing Up

- Buffer Manager provides a level of indirection
  - 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