CS150: Database & Datamining
Lecture 8: The IO Model

ShanghaiTech-SIST

Spring 2019

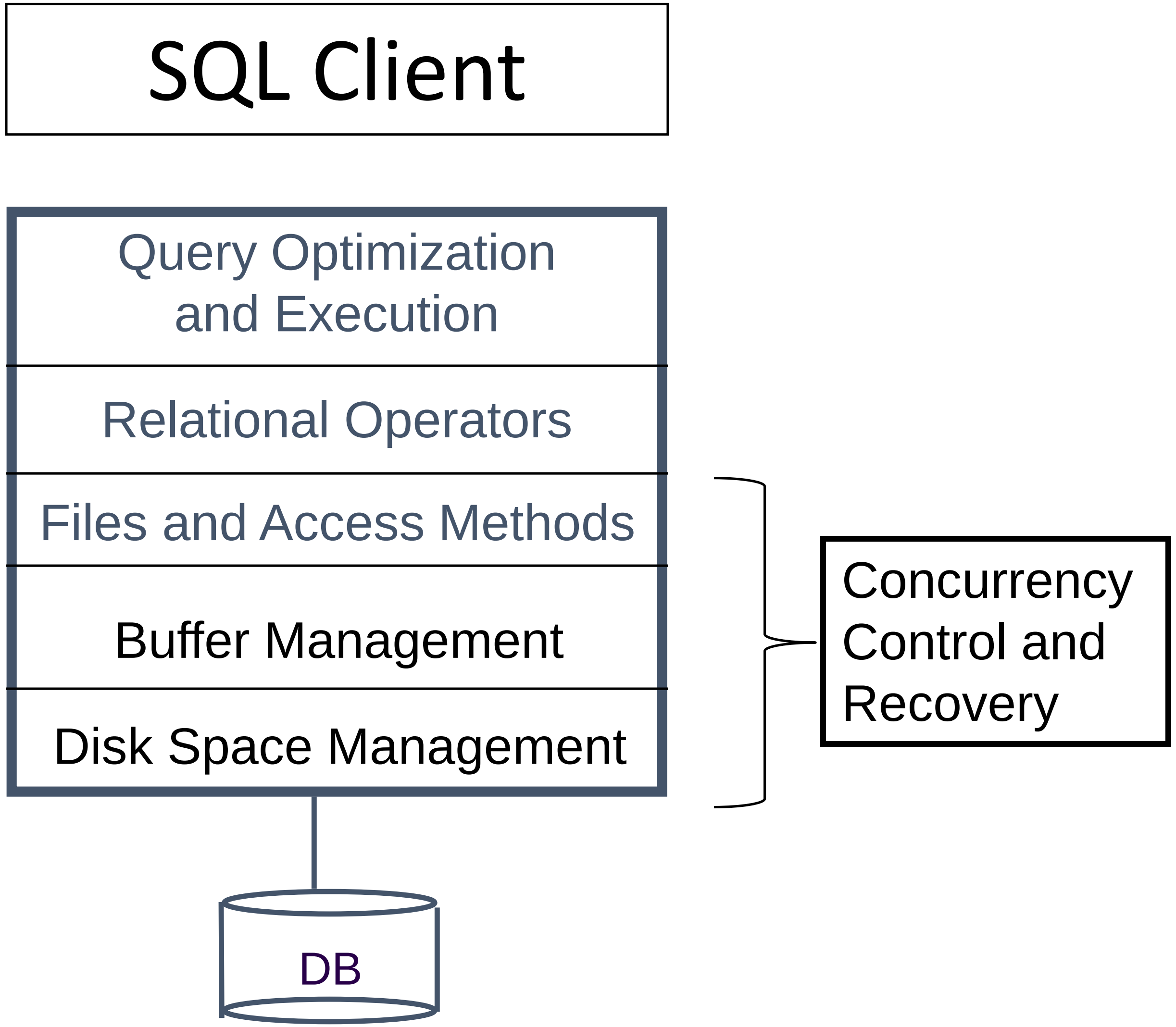
Acknowledgement: Slides are adopted from the Berkeley course CS186 by Joey Gonzalez and Joe Hellerstein, Stanford CS145 by Peter Bailis.

Transition to **Mechanisms**

1. So you can **understand** what the database is doing!
    1. Understand the CS challenges of a database and how to use it.
    2. Understand how to optimize a query

2. Many **mechanisms** have become **stand-alone systems**
    - **Indexing** to Key-value stores
    - Embedded join processing
    - SQL-like languages take some aspect of what we discuss (PIG, Hive)

Block diagram of a DBMS



Today’s Lecture

1. The Disk and Files

2. The Buffer

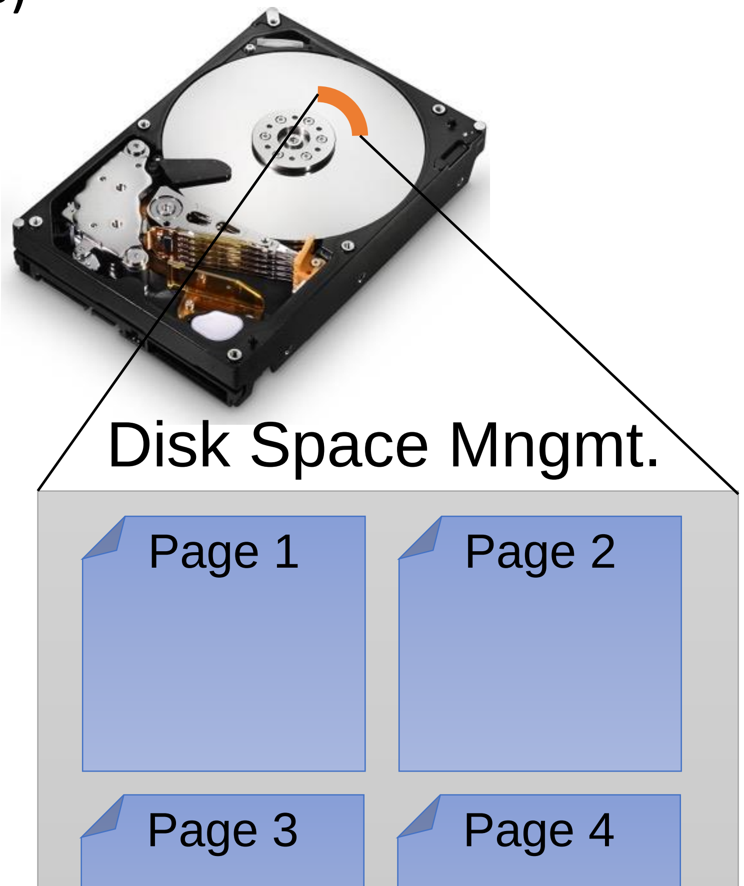
3. External Merge Algorithm

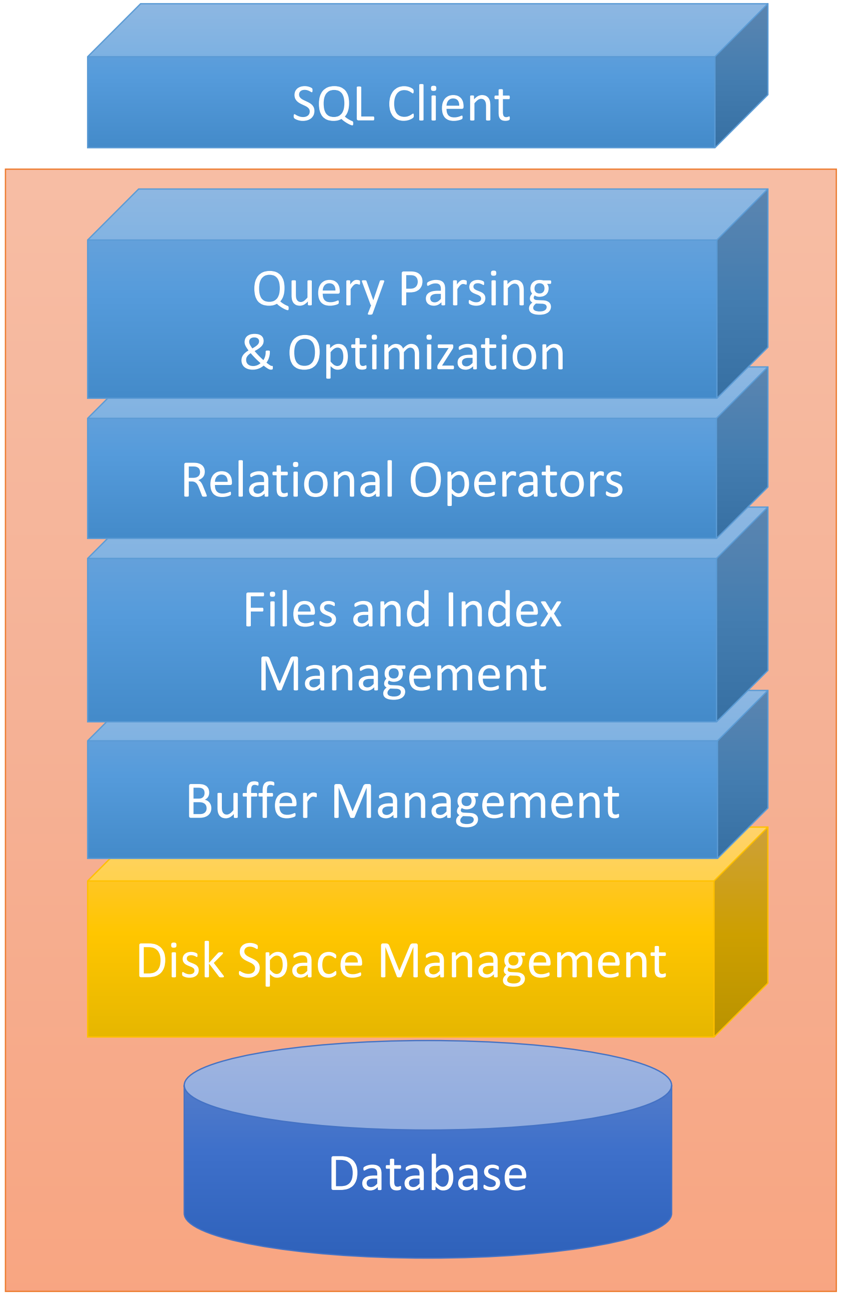
4. External Merge Sort Algorithm

1. The Disk and Files

Architecture of a DBMS

Translates page requests into physical bytes on one or more device(s)



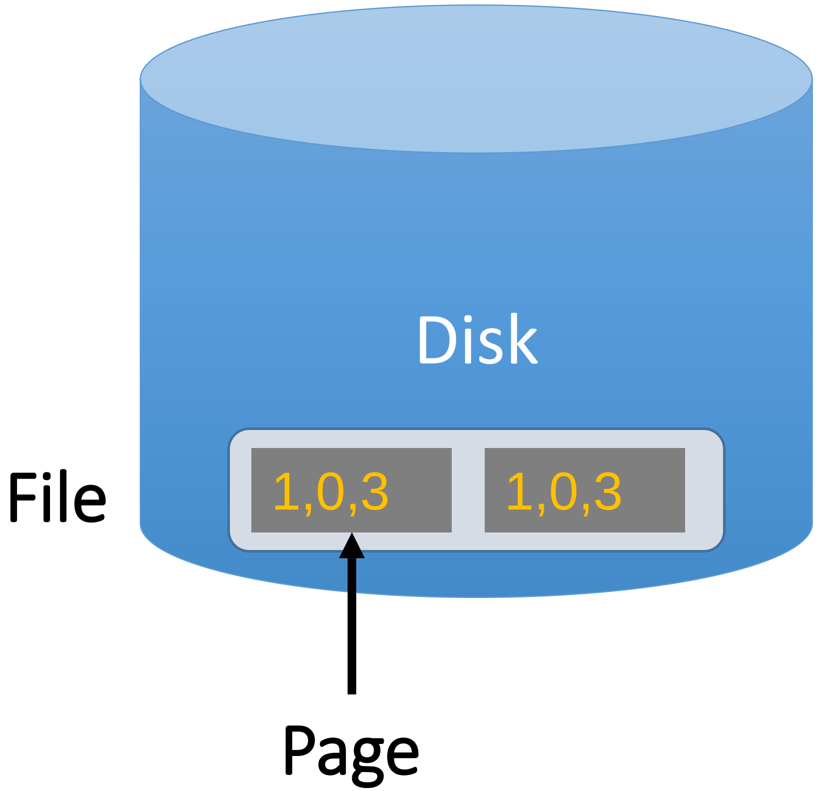


Overview: Files of Pages of Records

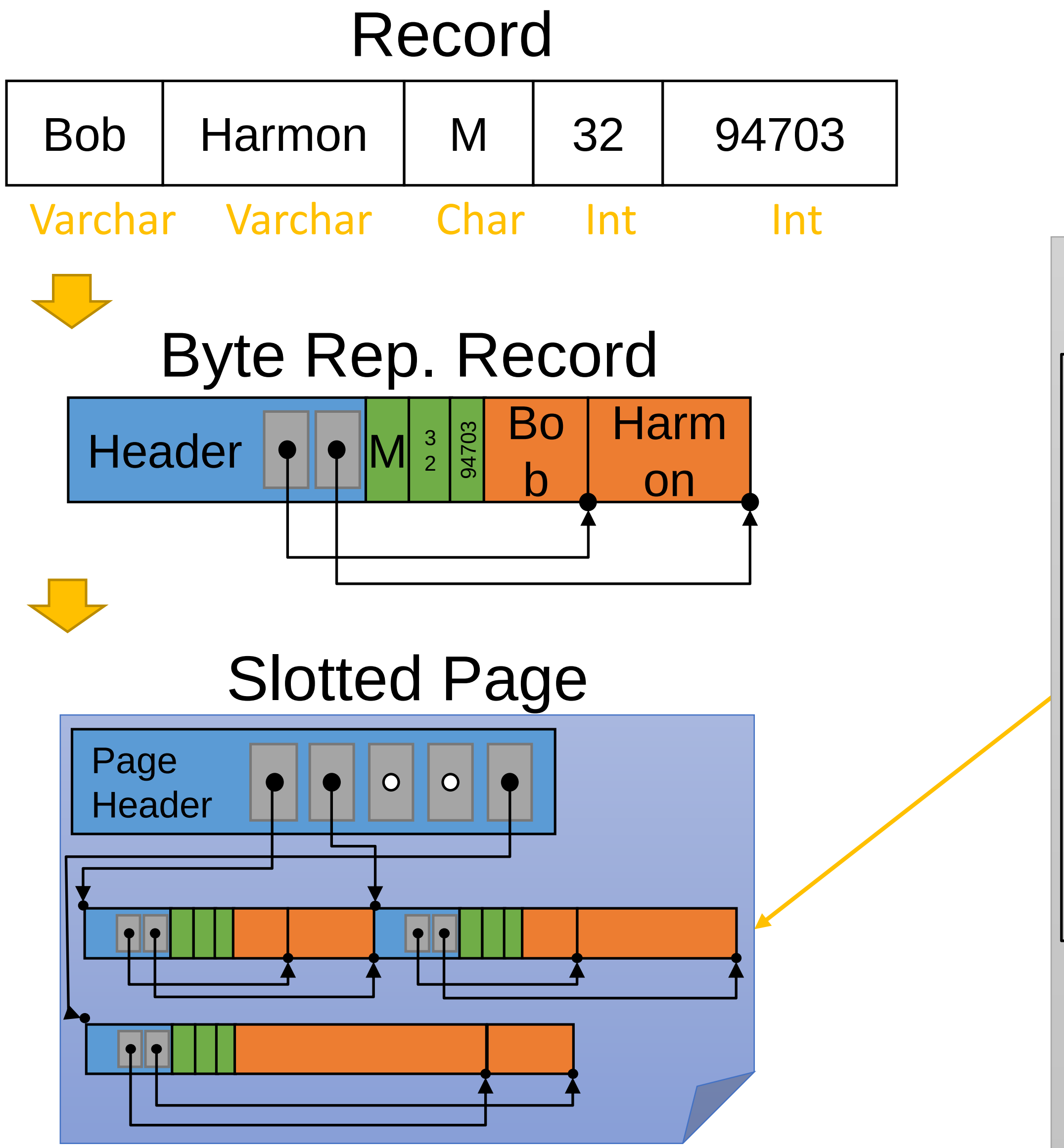
- Tables stored as a *logical files* consisting of *pages* each containing a collection of *records*
- Pages are managed
  - in memory by the buffer manager: higher levels of database only operate in memory
  - on disk by the disk space manager: reads and writes pages to physical disk/files

A Simplified Filesystem Model

- For us, a **<u>page</u>** is a ***fixed-sized array*** of memory
  - Think: One or more disk blocks
  - Interface:
    - write to an entry (called a **slot**) or set to “None”
  - DBMS also needs to handle variable length fields
    - Page layout is important for good hardware utilization as well (see next next lecture)
- And a **<u>file</u>** is a *variable-length list* of pages
  - Interface: create / open / close; next_page(); etc.

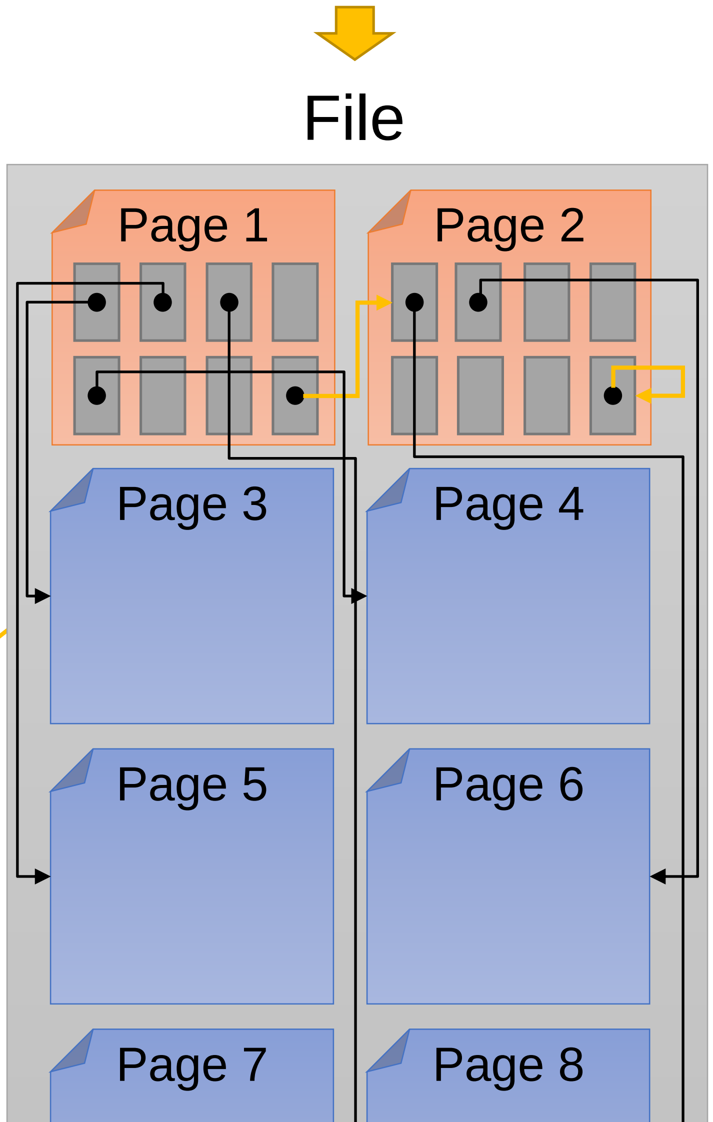


Overview



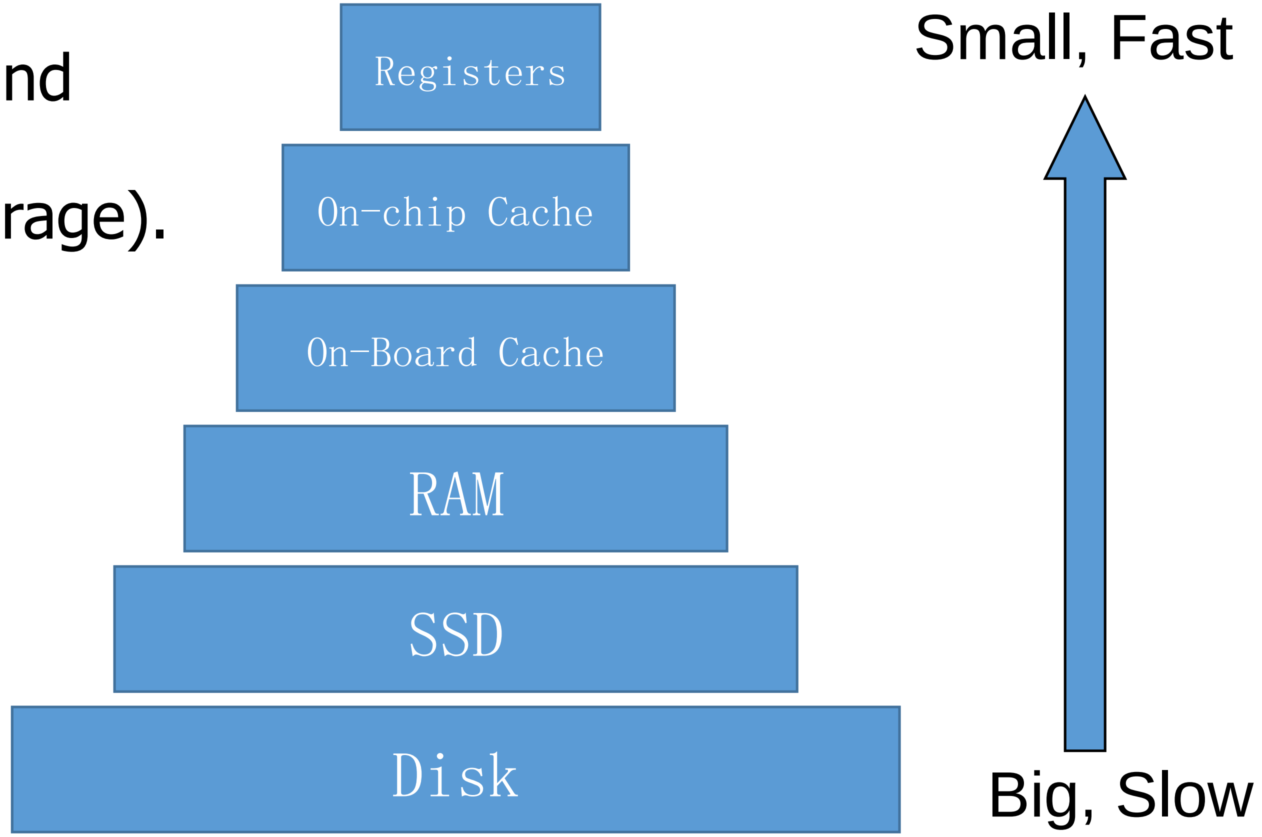
Table

Name	Addr	Sex	Age	Zip
Bob	Harmon	M	32	94703
Alice	Mabel	F	33	94703
Jose	Chavez	M	31	94110
Jane	Chavez	F	30	94110



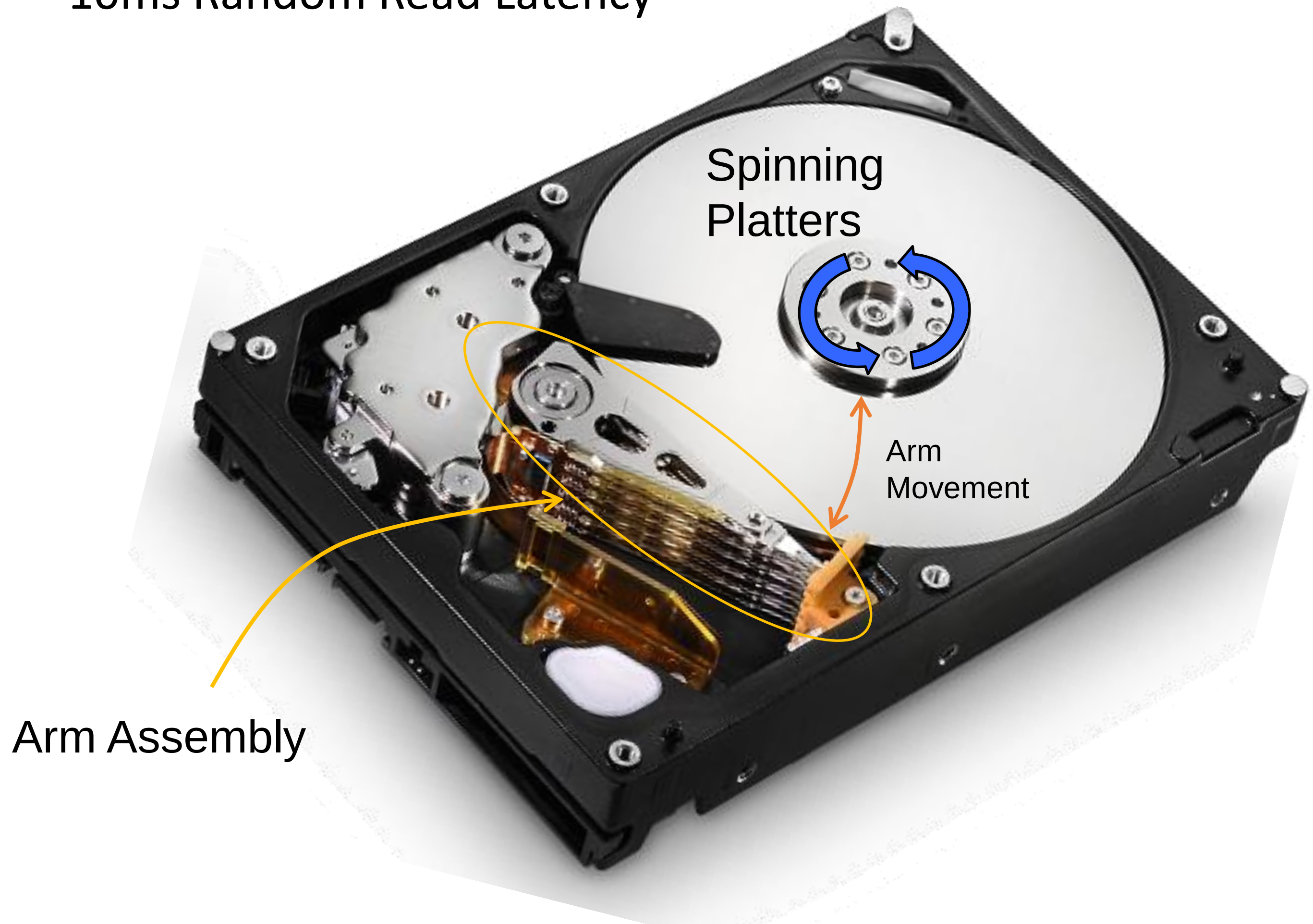
The Storage Hierarchy

- Main memory (RAM) for currently used data.
- Disk for main database and backups/logs (secondary & tertiary storage).
- The role of Flash (SSD) varies by deployment
  - Sometimes the DB
  - Sometimes a cache



Disks

- DBMS stores information on Disks and SSDs.
  - Disks are a mechanical anachronism (slow!)
    - 10ms Random Read Latency



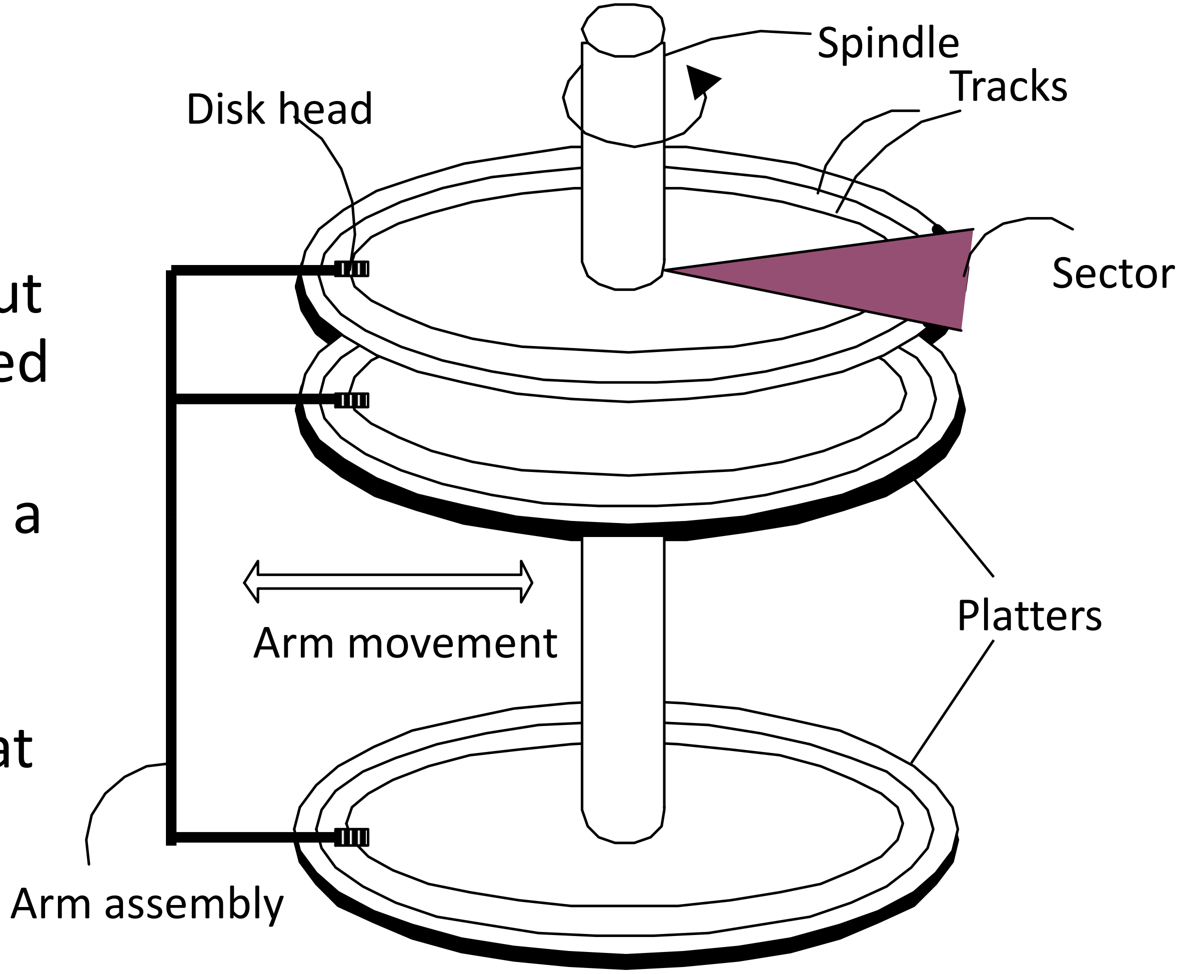
Components of a Disk

- Platters spin (say 15000 rpm)

- Arm assembly moved in or out to position a head on a desired track.
  - Tracks under heads make a cylinder (imaginary)

- Only one head reads/writes at any one time

- Block/page size is a multiple of (fixed) sector size

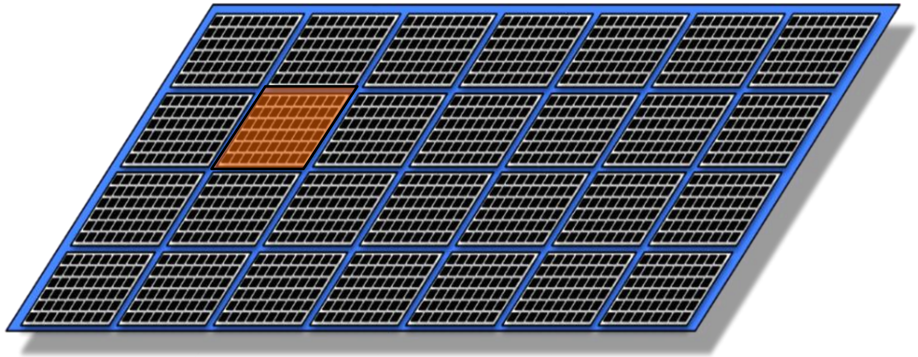


Arranging Pages on Disk

- *"Next"* page concept:
  - pages on same track, followed by
  - pages on same cylinder, followed by
  - pages on adjacent cylinder
- Arrange file pages sequentially on disk
  - minimize seek and rotational delay.
- For a sequential scan, *pre-fetch*
  - several pages at a time!
- Read large consecutive blocks

Notes on Flash (SSD)

- Issues in current generation (NAND)
  - 4-8K reads, 1-2MB writes
  - Only 2k-3k erasures before failure, so move writes around ("wear leveling")
  - *Write amplification*: big units, need to reorg for garbage collection & wear
- So... read is fast and *predictable*
  - Single read access time: 0.03 ms
  - 4KB random reads: ~500MB/sec
  - Sequential reads: ~525MB/sec
  - 64K: 0.48msec
- But.. write is not! Slower for random
  - Single write access time: 0.03ms
  - 4KB random writes: ~120MB/sec
  - Sequential writes: ~480MB/sec



Disks and Files

- DBMS stores information on **Disks** and **SSDs**.
  - Disks are a mechanical **anachronism** (slow!)
  - SSDs faster, **slow relative to memory**, costly writes
- DBMS operate at **Block Level**
  - Read and Write large **chunks seq. bytes**
    - Leverage cache hierarchy and HW pre-fetch
    - Amortize seek delays on HDDs and Writes on SSD
  - *Sequentially: Next* disk block is fastest
  - Maximize usage of data per R/W
- Organize data for fast in memory processing (i.e., mapping)

A note on (confusing) terminology

Block = Unit of transfer for disk read/write

- 64KB - 128KB is a good number today
- Book says 4KB

Page = Fixed size contiguous chunk of memory

- Assume same size as block
- Refer to corresponding blocks on disk

For simplicity we use Block and Page interchangeably.

Disk Space Management

Lowest layer of DBMS, manages space on disk

- Mapping pages to locations on disk
- Loading pages from disk to memory
- Saving pages back to disk & ensuring writes

Higher levels call upon this layer to:

- read/write a pages
- allocate/de-allocate logical pages

Request for a *sequence* of pages best satisfied by pages stored sequentially on disk

- Physical details hidden from higher levels of system
- Higher levels may assume **Next Page** is fast!

Disk Space Management Implementation

Proposal 1: Talk to the device directly

- Could be very fast if you knew the device well
- What happens when devices change?

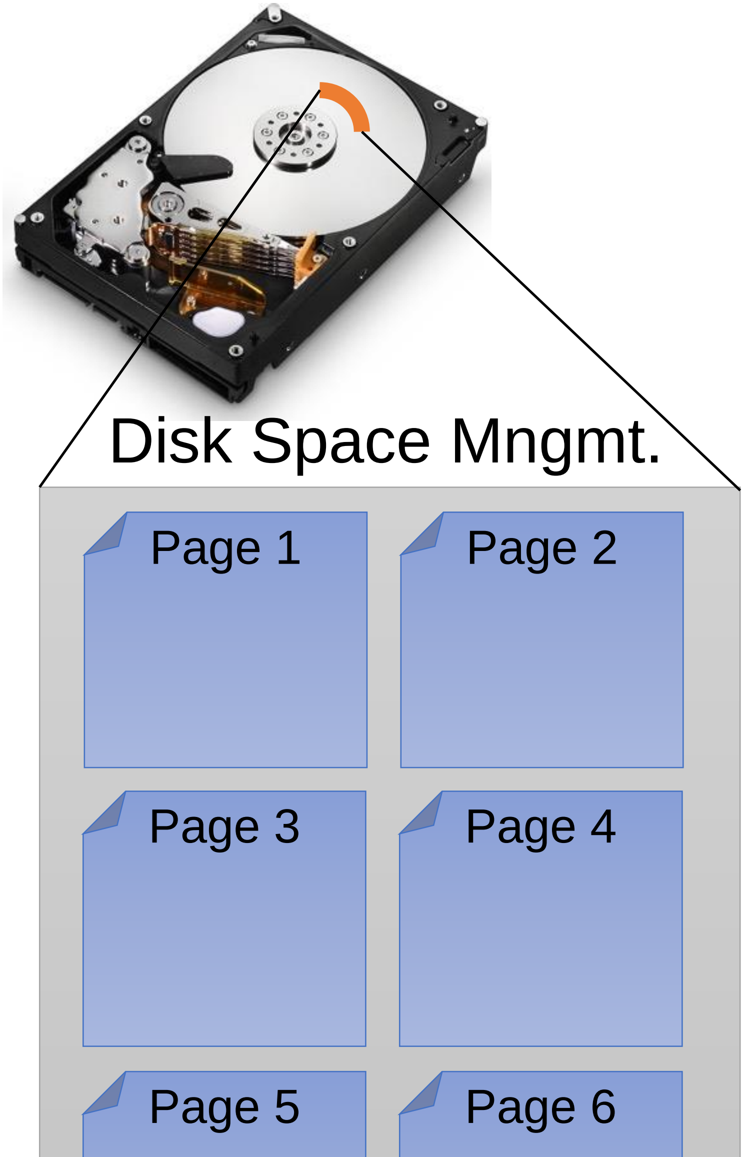
Proposal 2: Run over filesystem (FS)

- Allocate single large “contiguous” file and assume sequential / nearby byte access are fast
- Most FS optimize for sequential access and temporal locality (buffer cache on hot items)
  - Sometimes disable FS buffering
- May span multiple files on multiple disks / machines

Typically sits on top of local file system

Get Page 4 Get Page 5

Disk Space Management

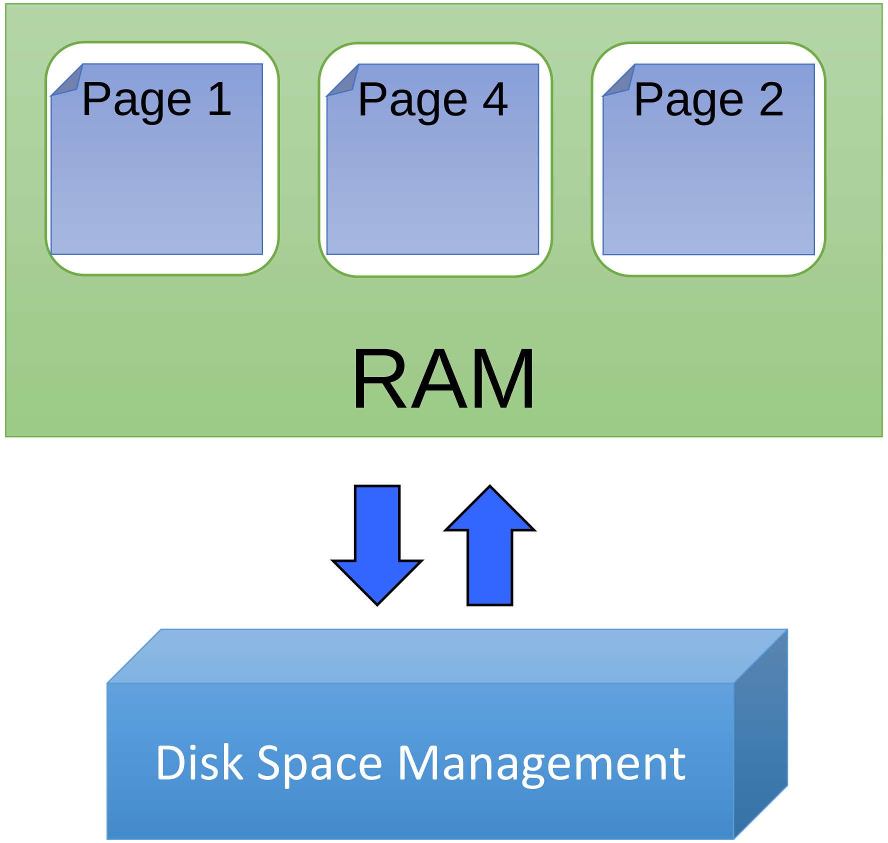


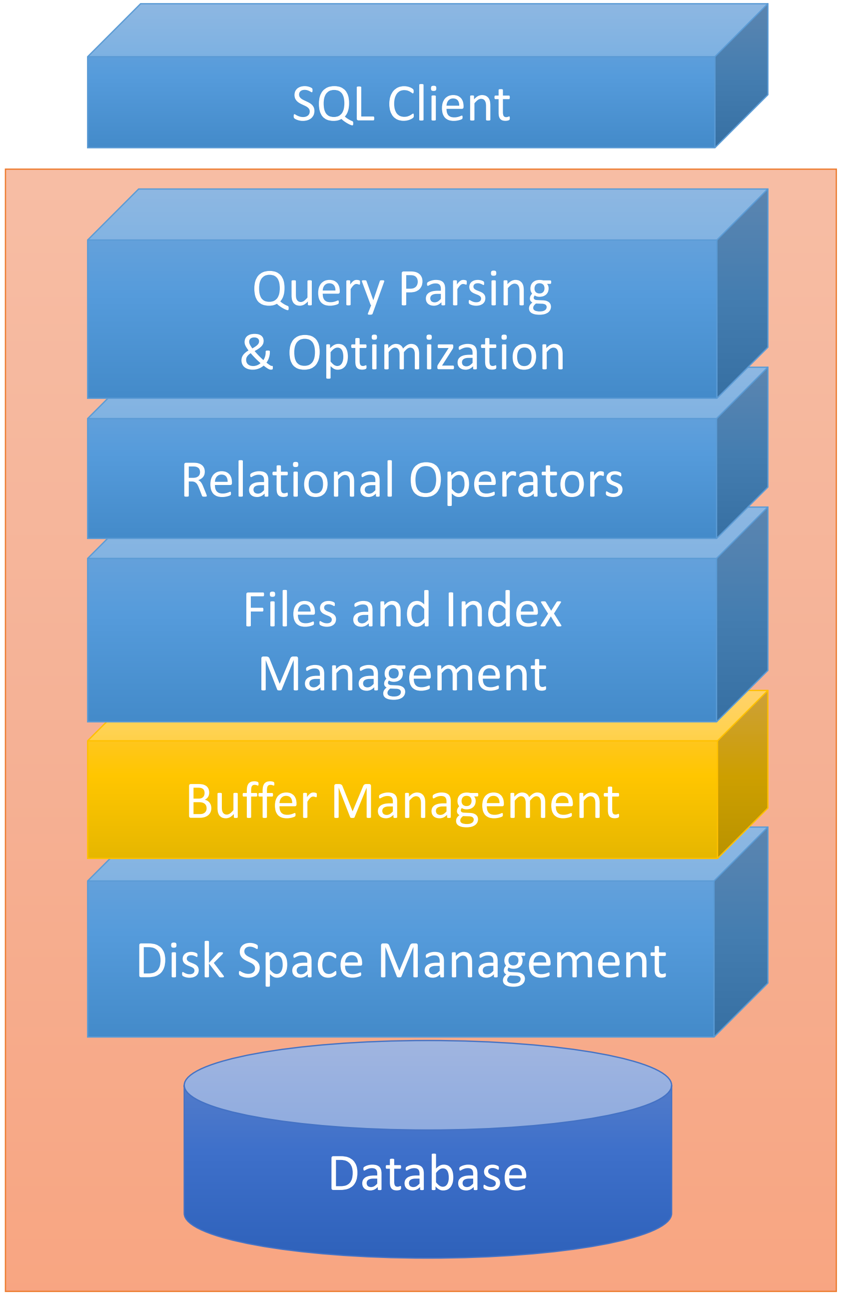
- Provide API to read and write pages to device
- Pages: block level organization of bytes on disk
- Ensures next locality and abstracts FS/Device details

2. The Buffer

Architecture of a DBMS

Illusion of operating in memory



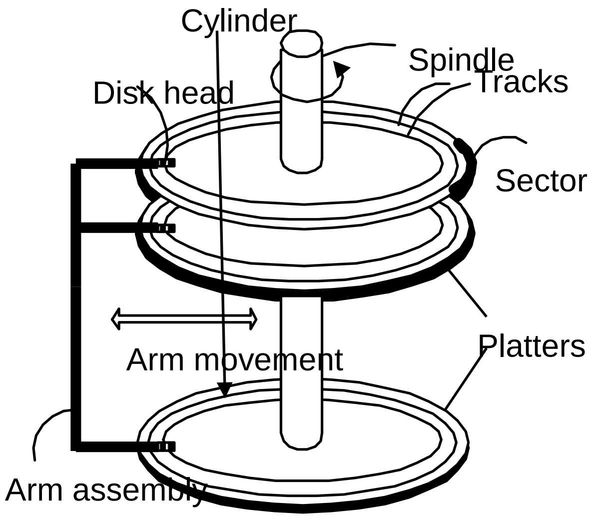


What you will learn about in this section

1. RECAP: Storage and memory model

2. Buffer primer

High-level: Disk vs. Main Memory



Disk:

- ***Slow:*** Sequential *block* access
  - Read a blocks (not byte) at a time, so sequential access is cheaper than random
  - **Disk read / writes are expensive!**
- ***Durable:*** We will assume that once on disk, data is safe!
- ***Cheap***

Random Access Memory (RAM) or Main Memory:

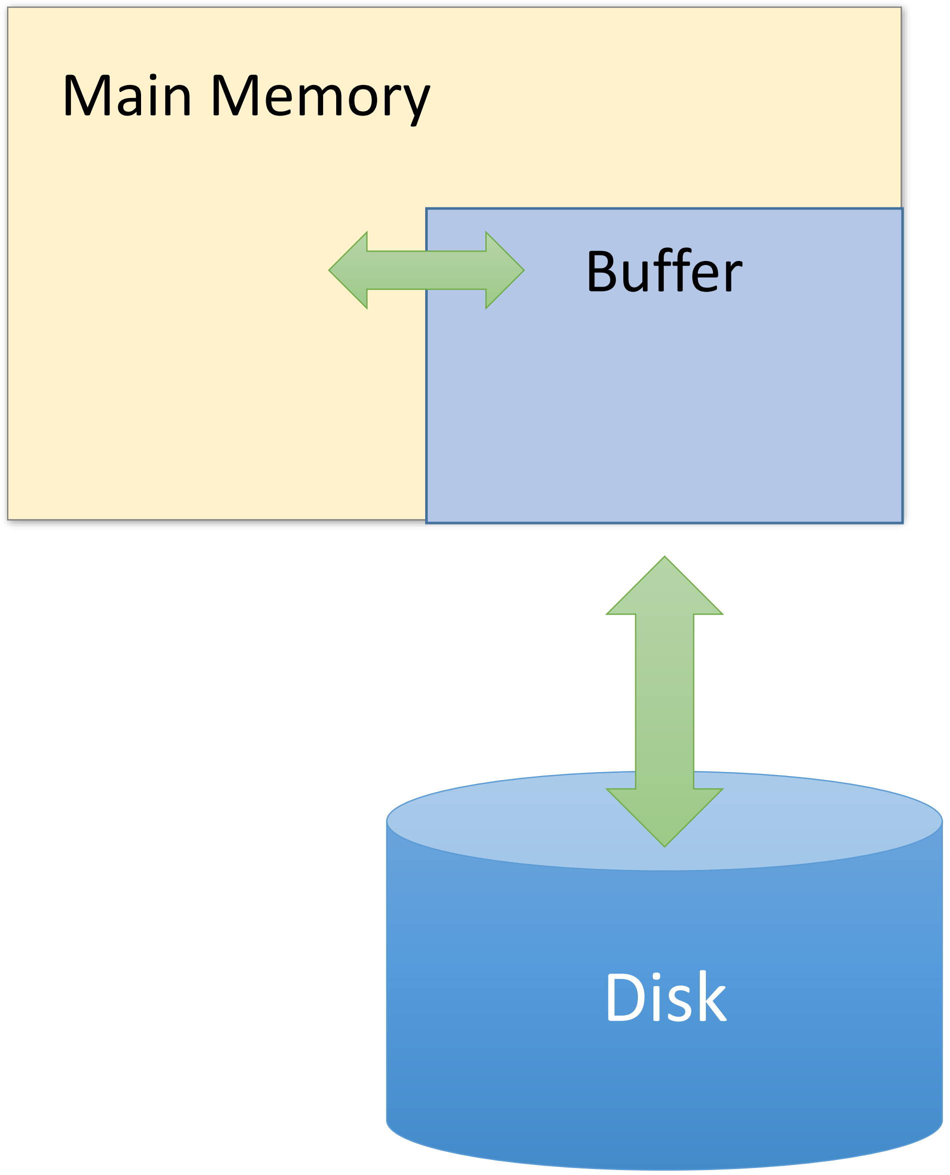
- ***Fast:*** Random access, byte addressable
  - ~10x faster for sequential access
  - ~100,000x faster for random access!
- ***Volatile:*** Data can be lost if e.g. crash occurs, power goes out, etc!
- ***Expensive:*** For $100, get 16GB of RAM vs. 2TB of disk!

The Buffer

- A **buffer** is a region of physical memory used to store *temporary data*

    - *In this lecture:* a region in main memory used to store **intermediate data between disk and processes**

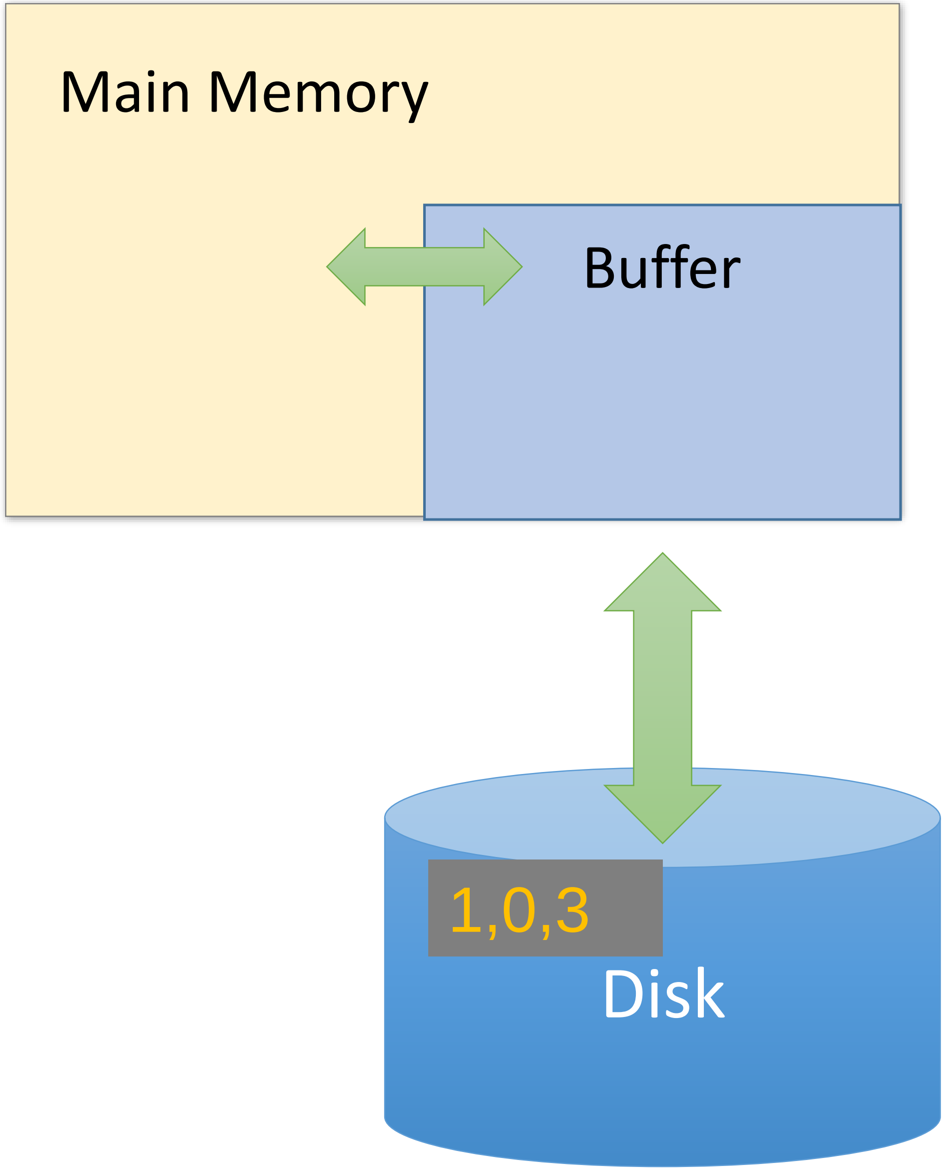
- *Key idea:* Reading / writing to disk is slow- need to cache data!



The (Simplified) Buffer

- In this class: We'll consider a buffer located in **main memory** that operates over **pages** and **files**:

  - <u>**Read(page):**</u> Read page from disk -> buffer *if not already in buffer*

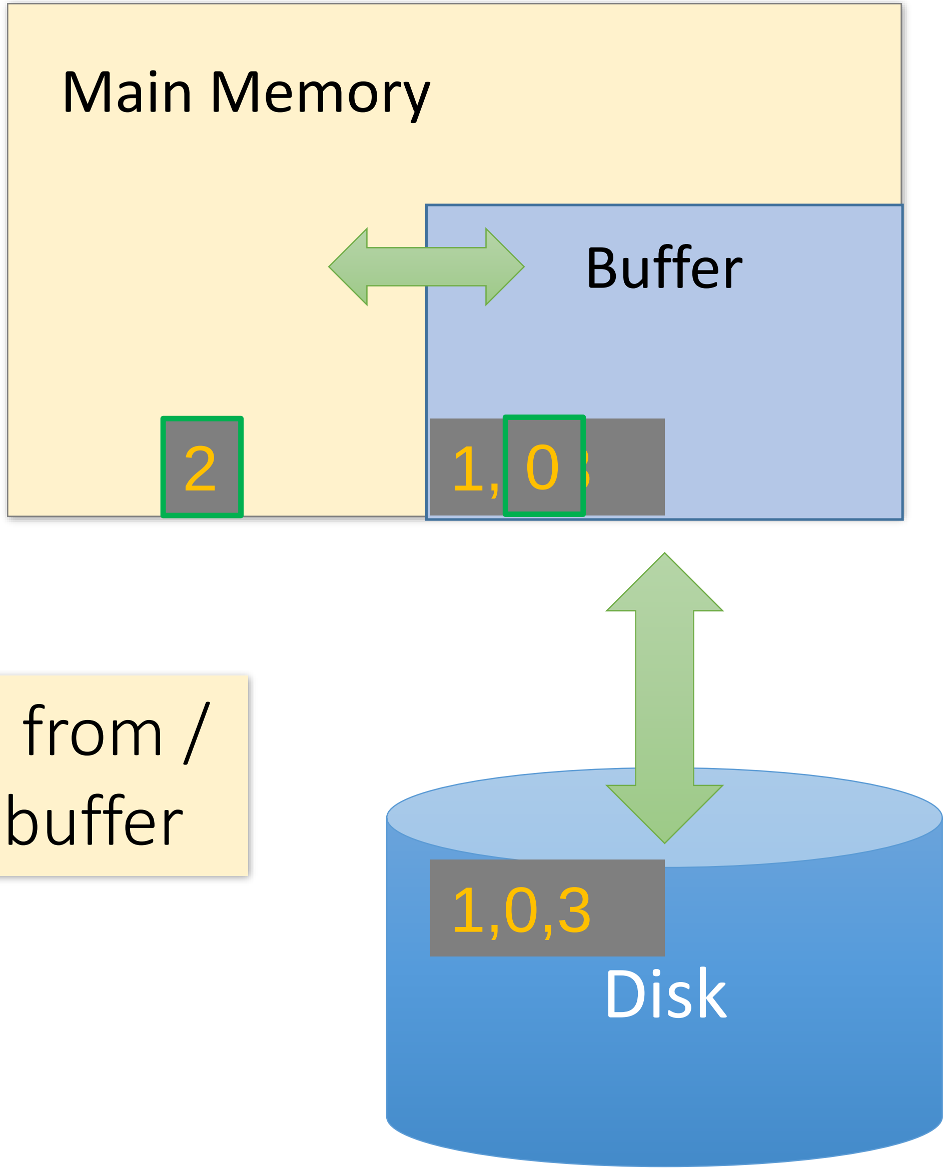


The (Simplified) Buffer

- In this class: We'll consider a buffer located in **main memory** that operates over **pages** and **files**:

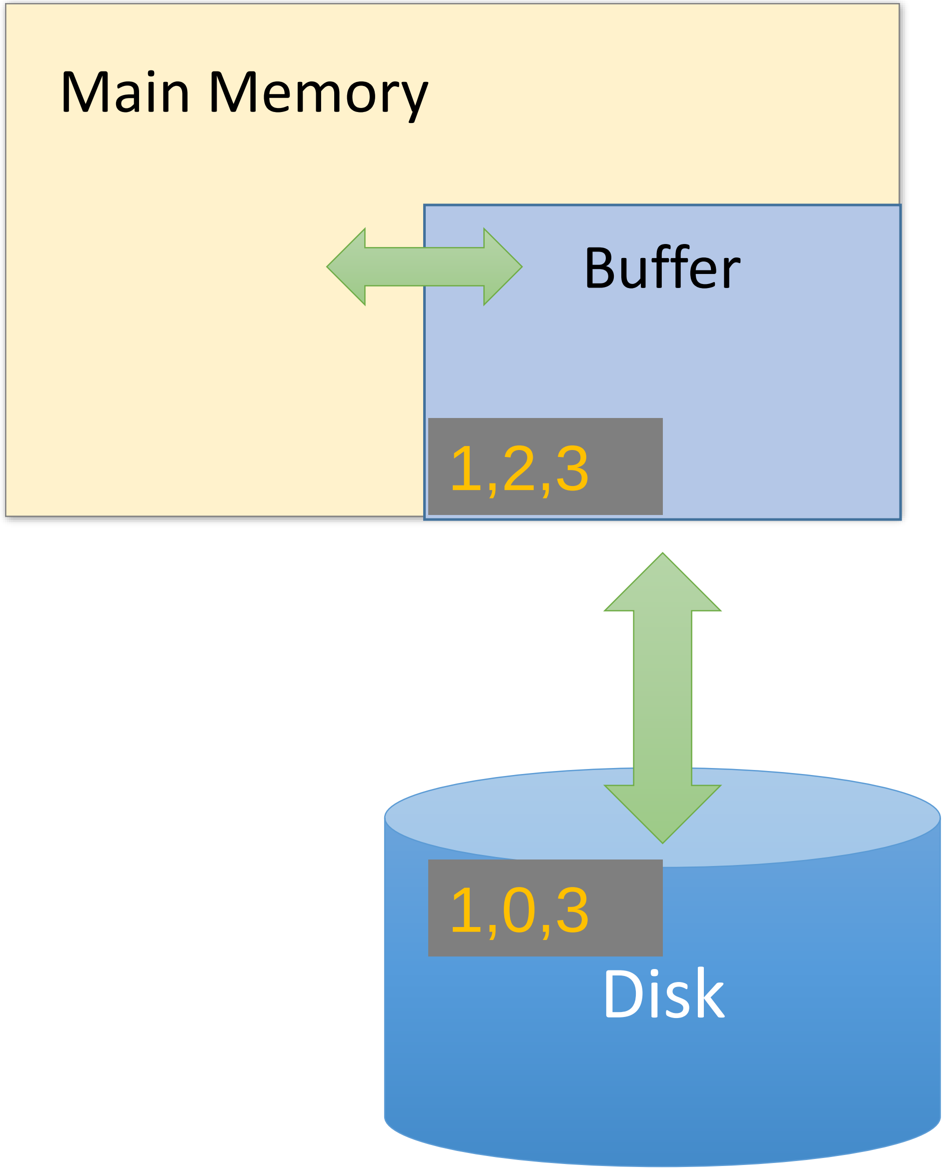
  - **<u>Read(page):</u>** Read page from disk -> buffer *if not already in buffer*

Processes can then read from / write to the page in the buffer



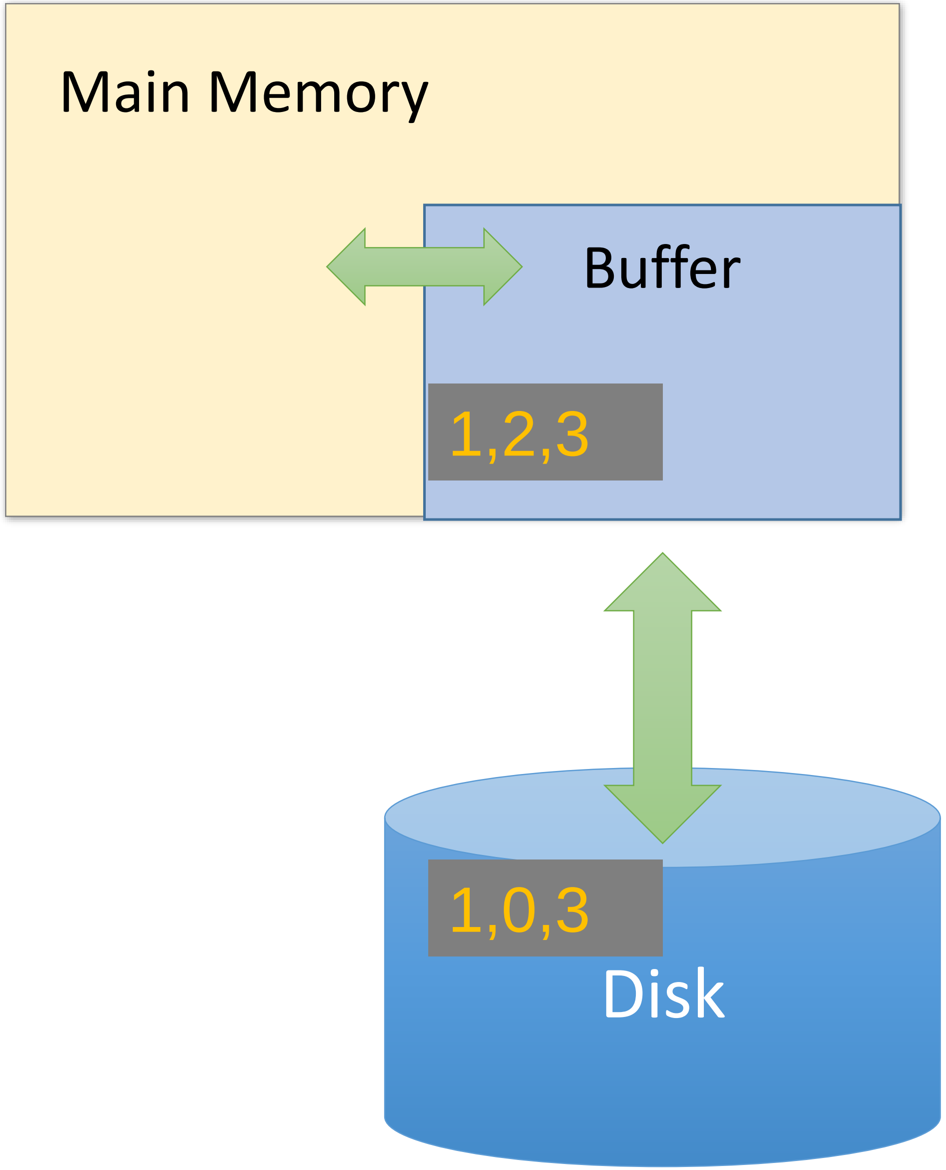
The (Simplified) Buffer

- In this class: We'll consider a buffer located in **main memory** that operates over **pages** and **files**:
  - **<u>Read(page):</u>** Read page from disk -> buffer *if not already in buffer*
  - **<u>Flush(page):</u>** Evict page from buffer & write to disk



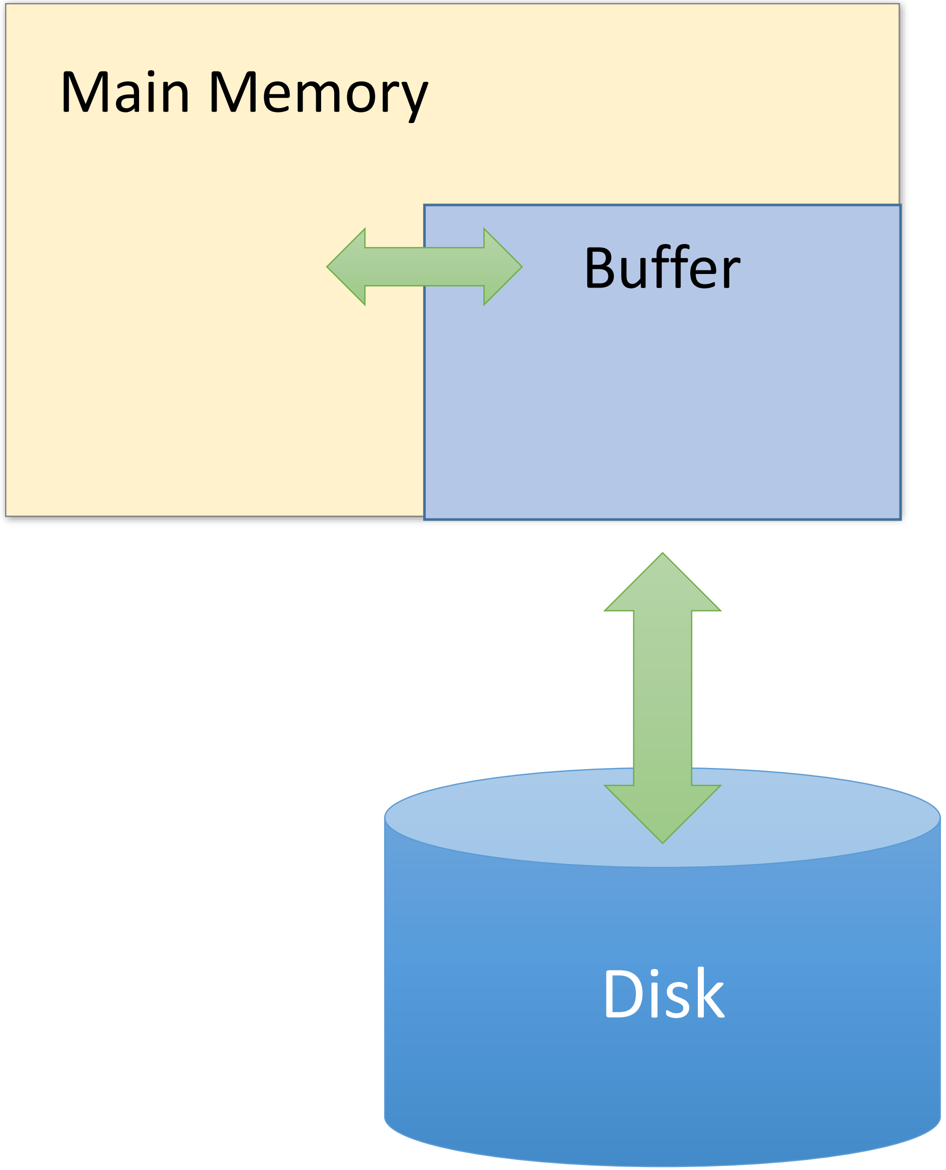
The (Simplified) Buffer

- In this class: We'll consider a buffer located in **main memory** that operates over **pages** and **files**:
  - **<u>Read(page):</u>** Read page from disk -> buffer *if not already in buffer*
  - **<u>Flush(page):</u>** Evict page from buffer & write to disk
  - **<u>Release(page):</u>** Evict page from buffer *without* writing to disk



Managing Disk: The DBMS Buffer

- Database maintains its own buffer
  - Why? The OS already does this…
  - DB knows more about access patterns.
    - Watch for how this shows up! (cf. *Sequential Flooding*)
  - Recovery and logging require ability to **flush** to disk.



When a Page is Requested ...

- Buffer pool information "table" contains:
  <frame#, pageid, pin_count, dirty>

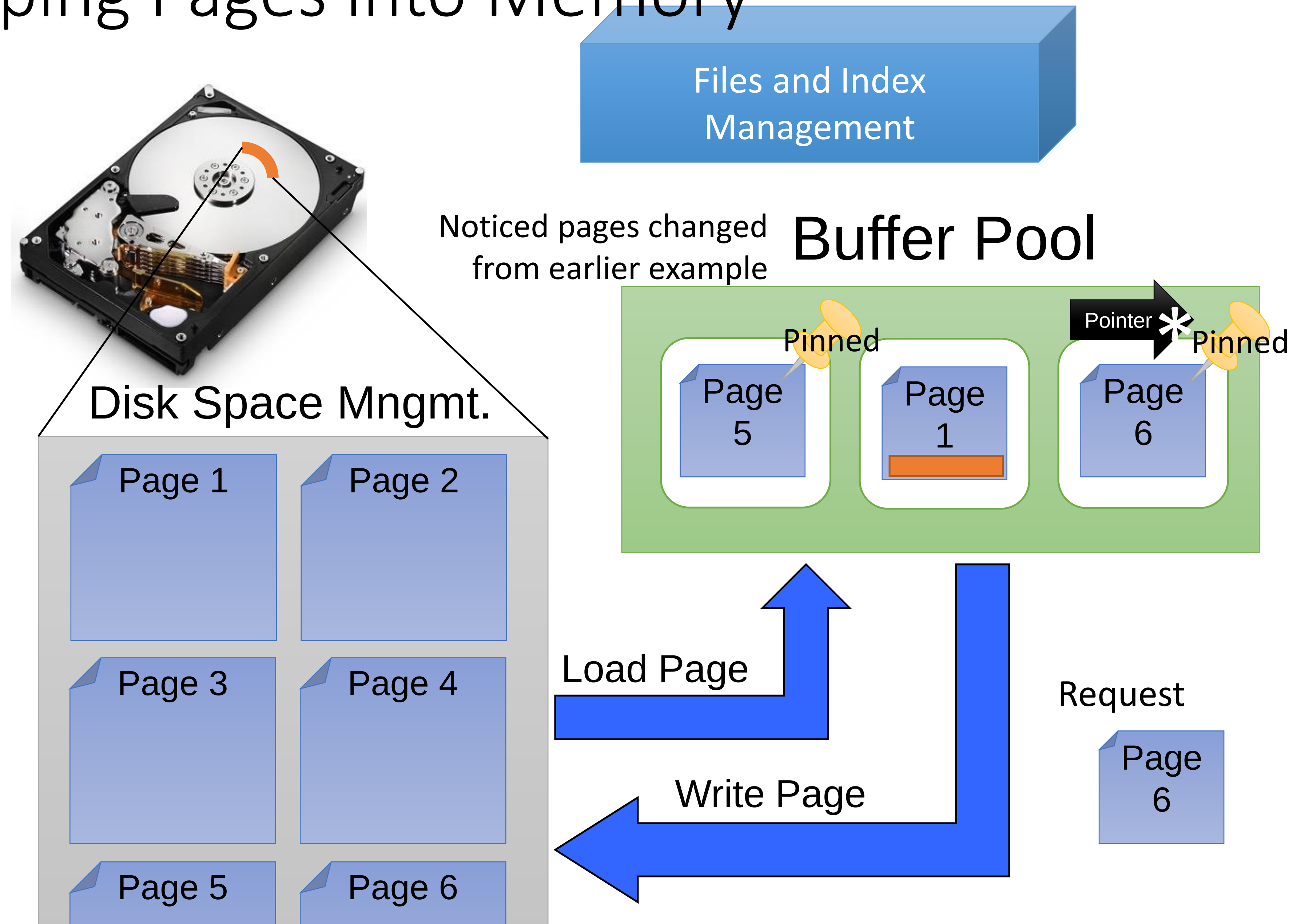
1. If requested page is not in pool:
   a. Choose a frame for *replacement.*
      Only "un-pinned" pages are candidates!
   b. If frame "dirty", write current page to disk
   c. Read requested page into frame

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!

Mapping Pages into Memory



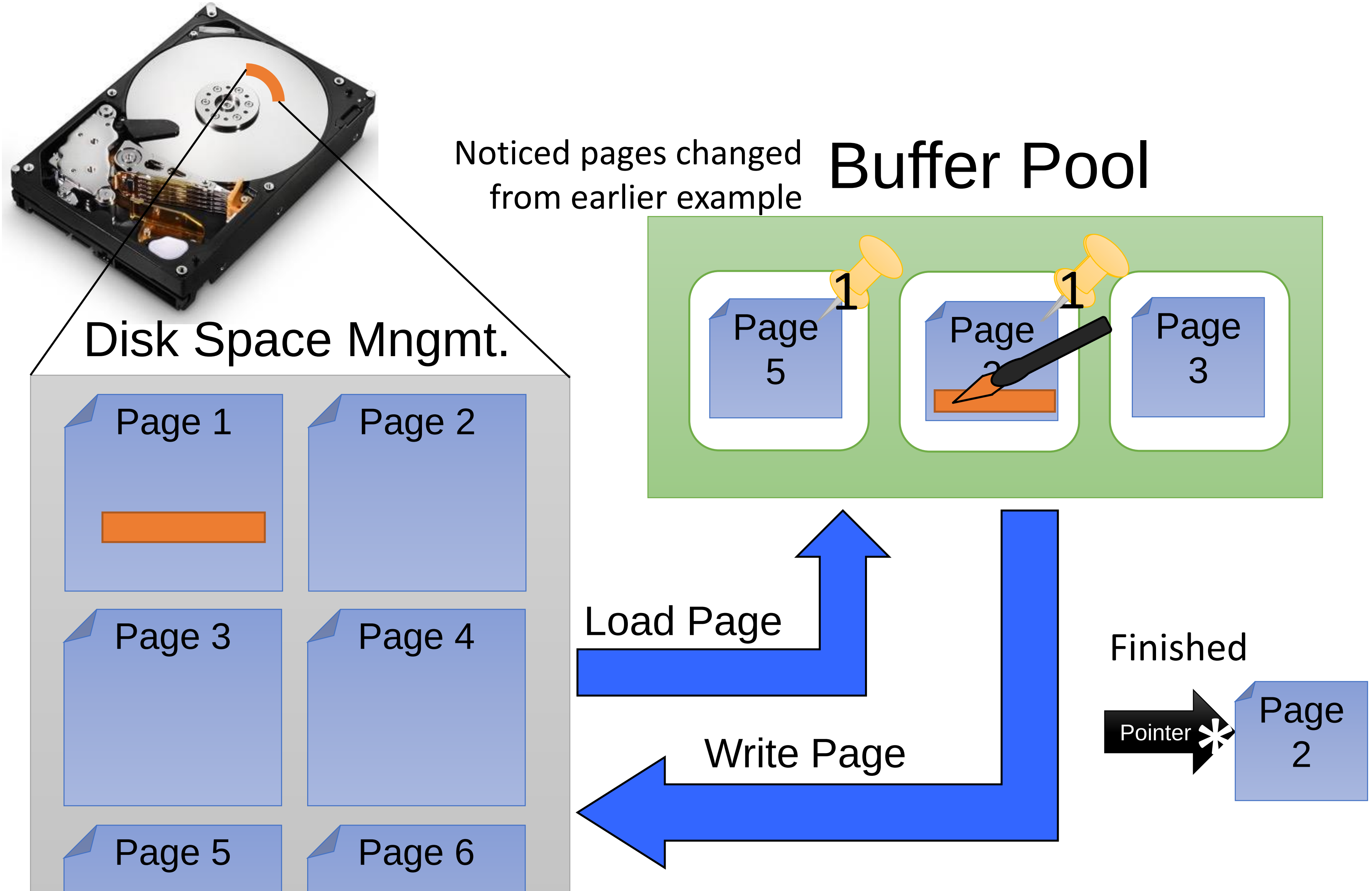
After Requestor Finishes

- Requestor of page must:
  1. indicate whether page was modified via *dirty* bit.
  2. *unpin* it (soon preferably!) why?

- 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”)

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

Mapping Pages into Memory



Page Replacement Policy

- 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/O’s;
  - Depends on the *access pattern.*

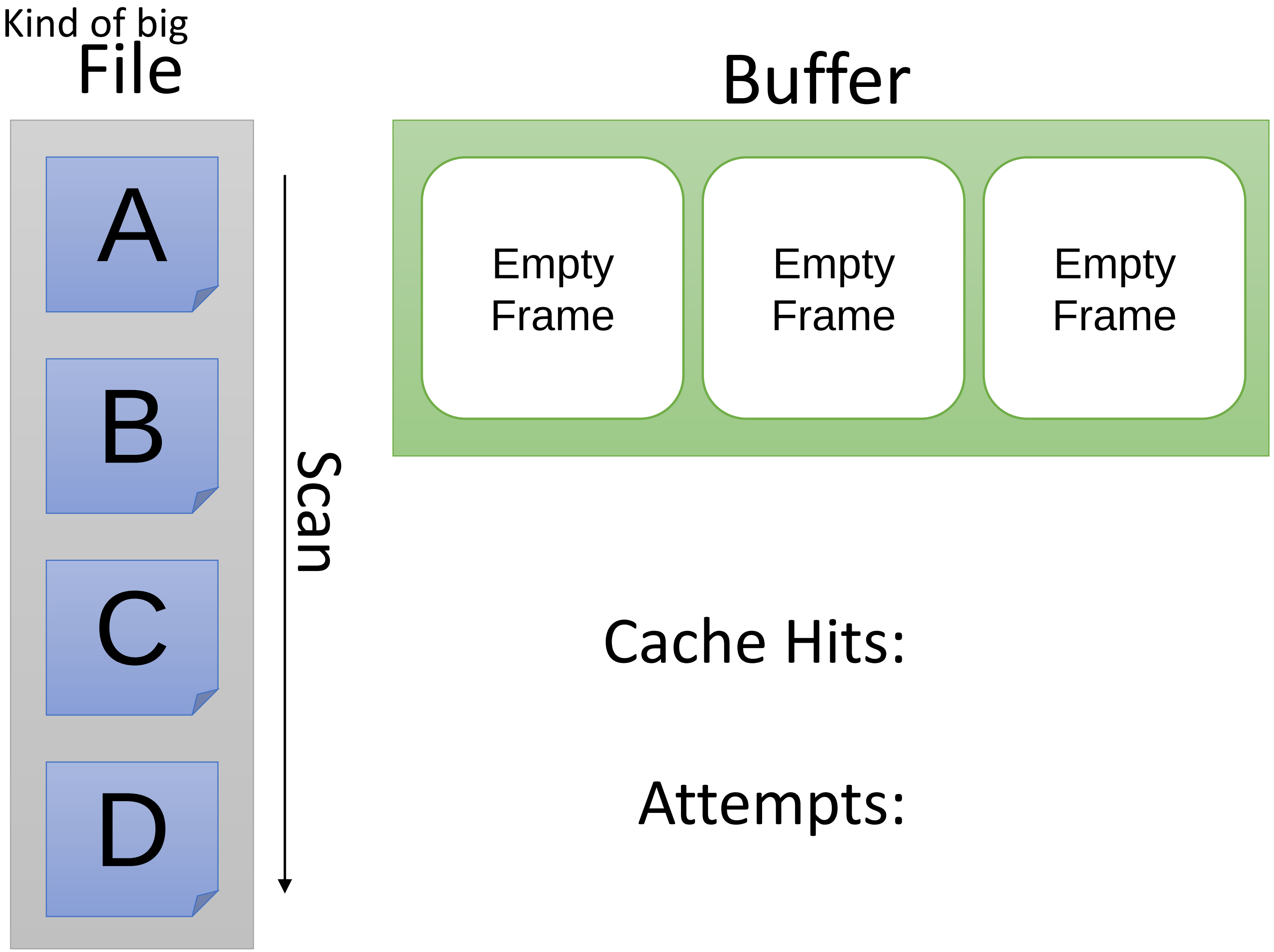
LRU Replacement Policy

- *Least Recently Used (LRU)*
  - Pinned Frame: not available to replace
  - track time each frame last *unpinned* (end of use)
  - replace the frame which least recently unpinned
- Very common policy: intuitive and simple
  - Works well for repeated accesses to popular pages (temporal locality)
  - *Can be costly. Why?*
    - *Need to maintain heap data-structure*
    - *Solution?*

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
- When might it perform poorly
  - *What about repeated scans of big files?*

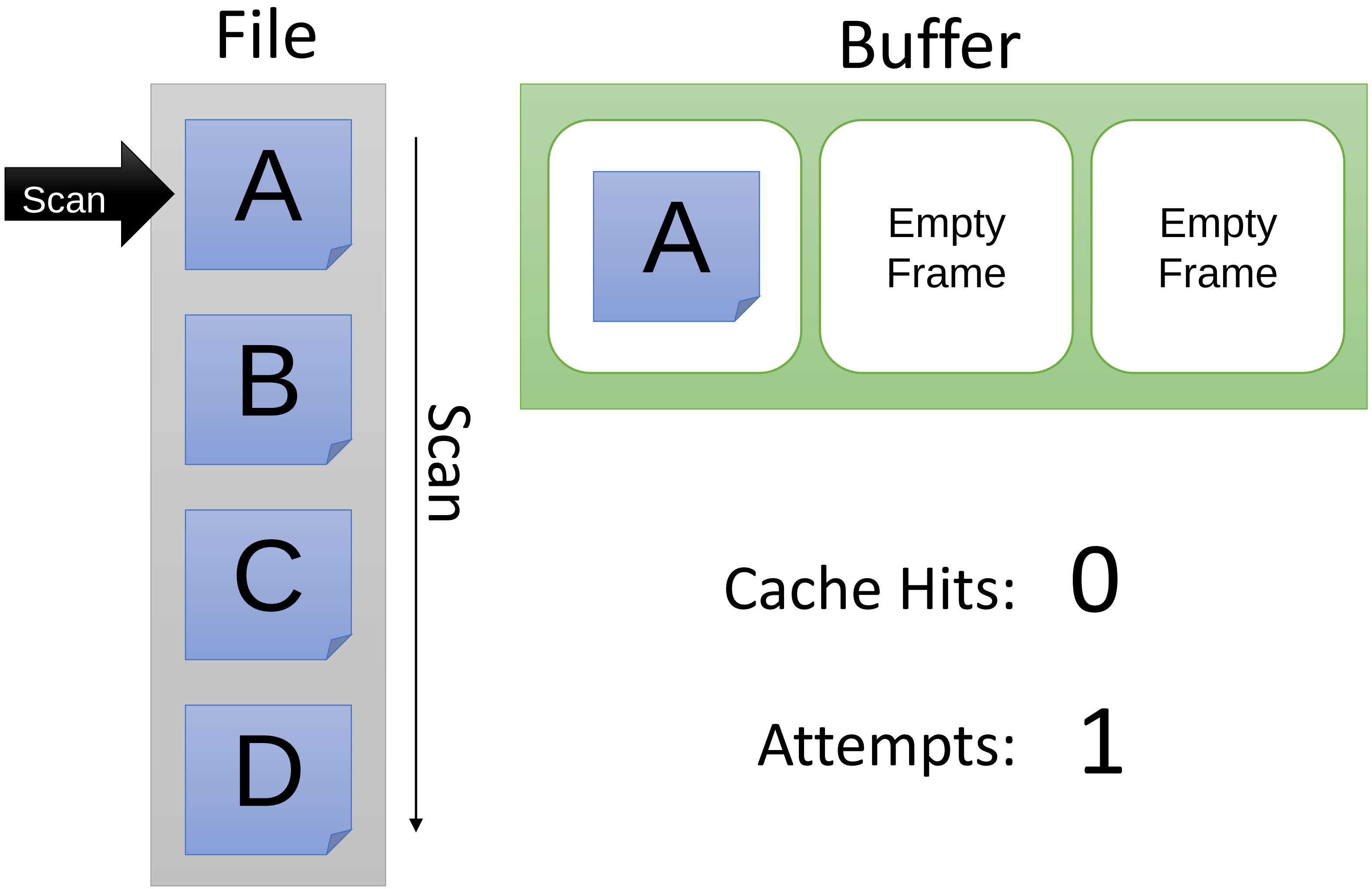
Repeated Scan of Big File (LRU)



Cache Hits:

Attempts:

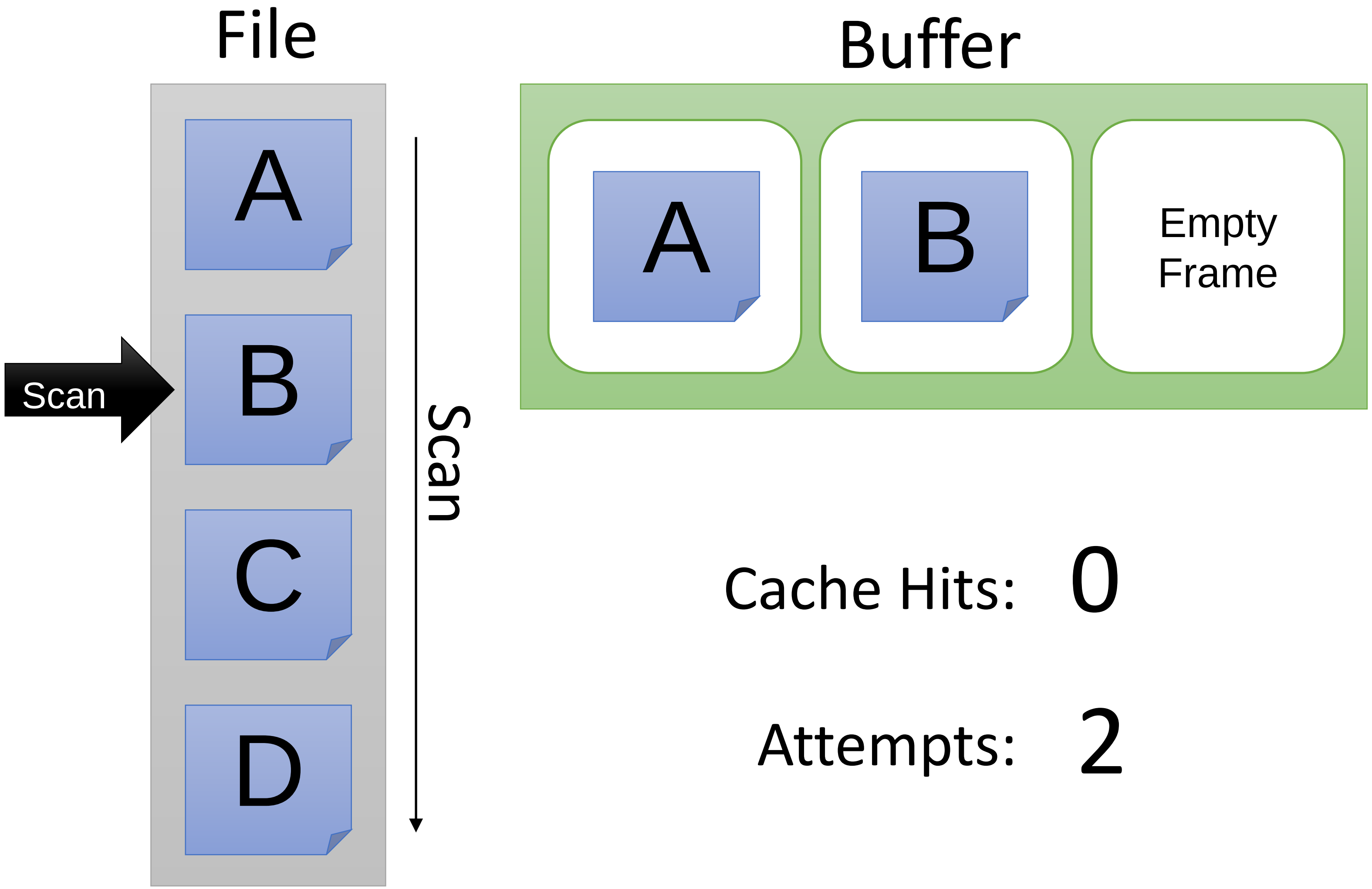
Repeated Scan of Big File (LRU)



Cache Hits: 0

Attempts: 1

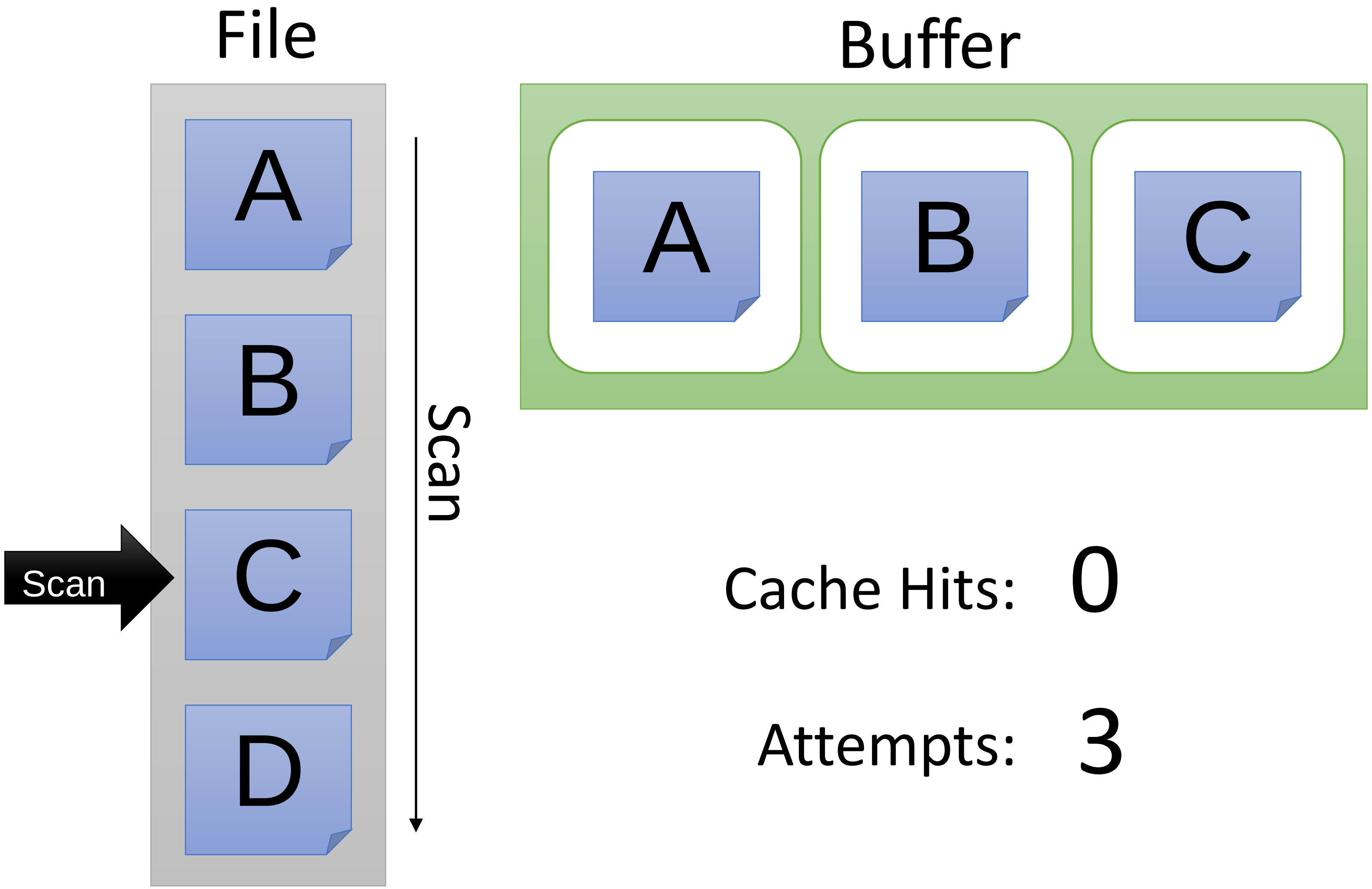
Repeated Scan of Big File (LRU)



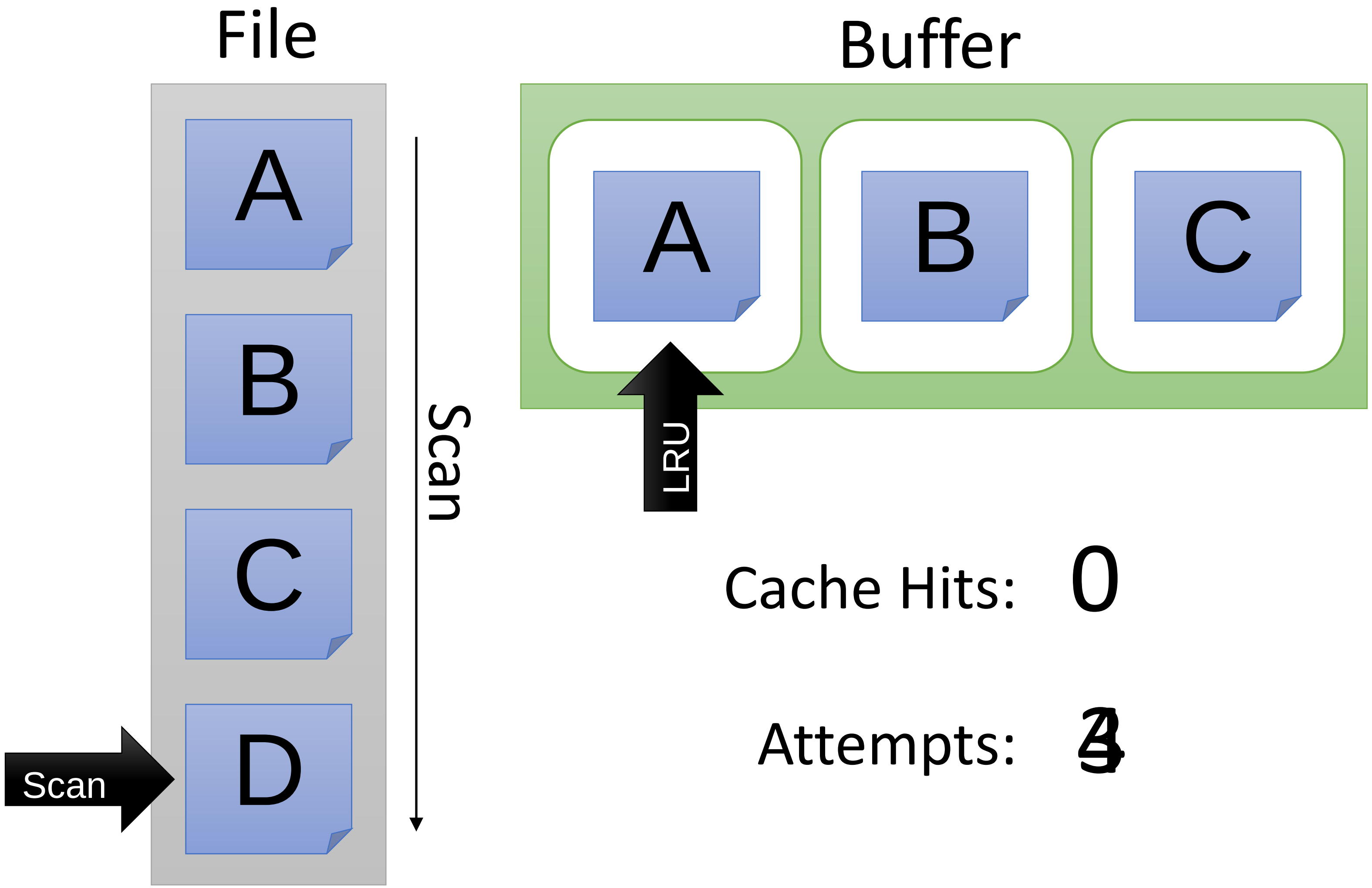
Cache Hits: 0

Attempts: 2

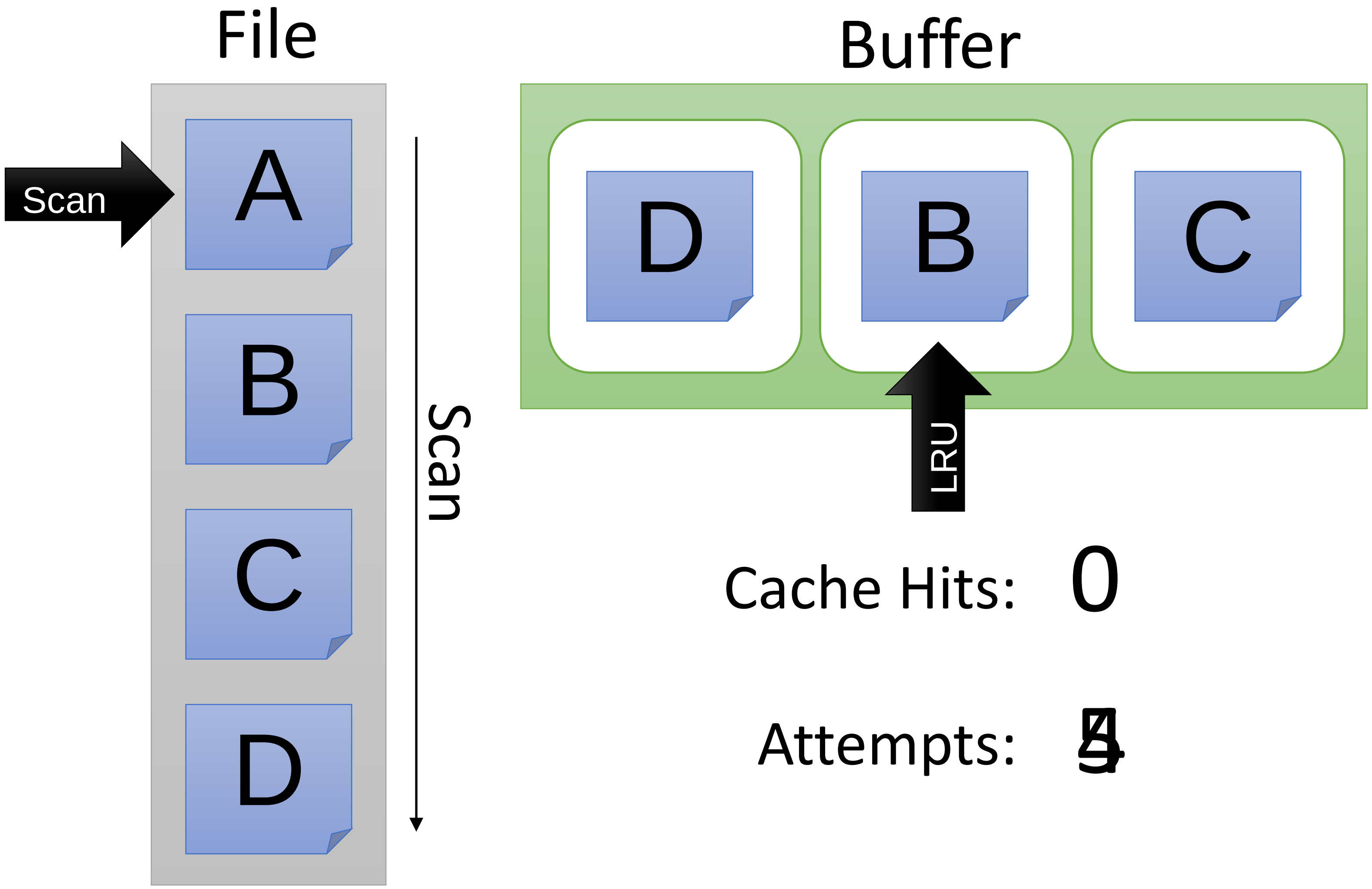
Repeated Scan of Big File (LRU)



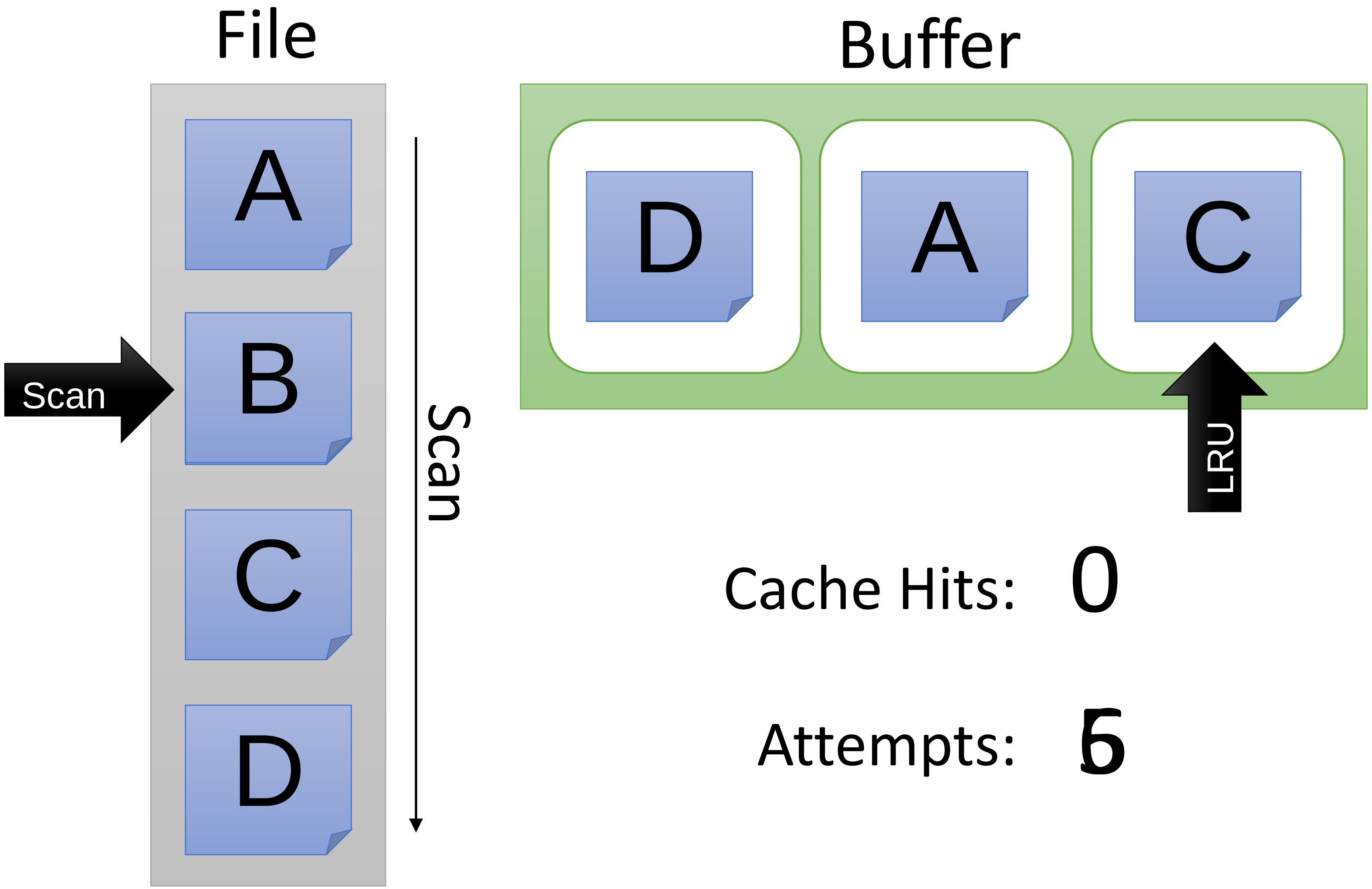
Repeated Scan of Big File (LRU)



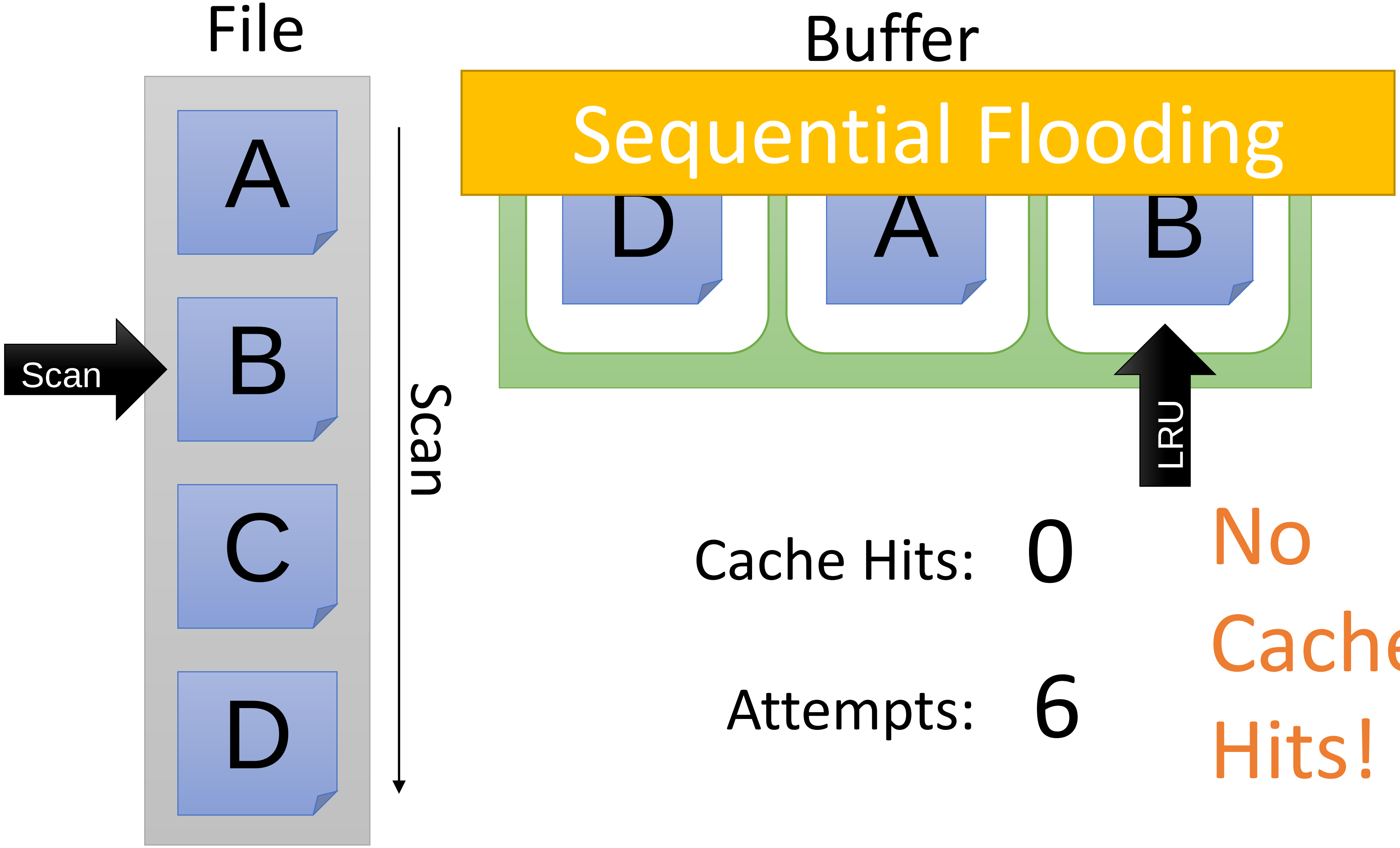
Repeated Scan of Big File (LRU)



Repeated Scan of Big File (LRU)



Repeated Scan of Big File (LRU)



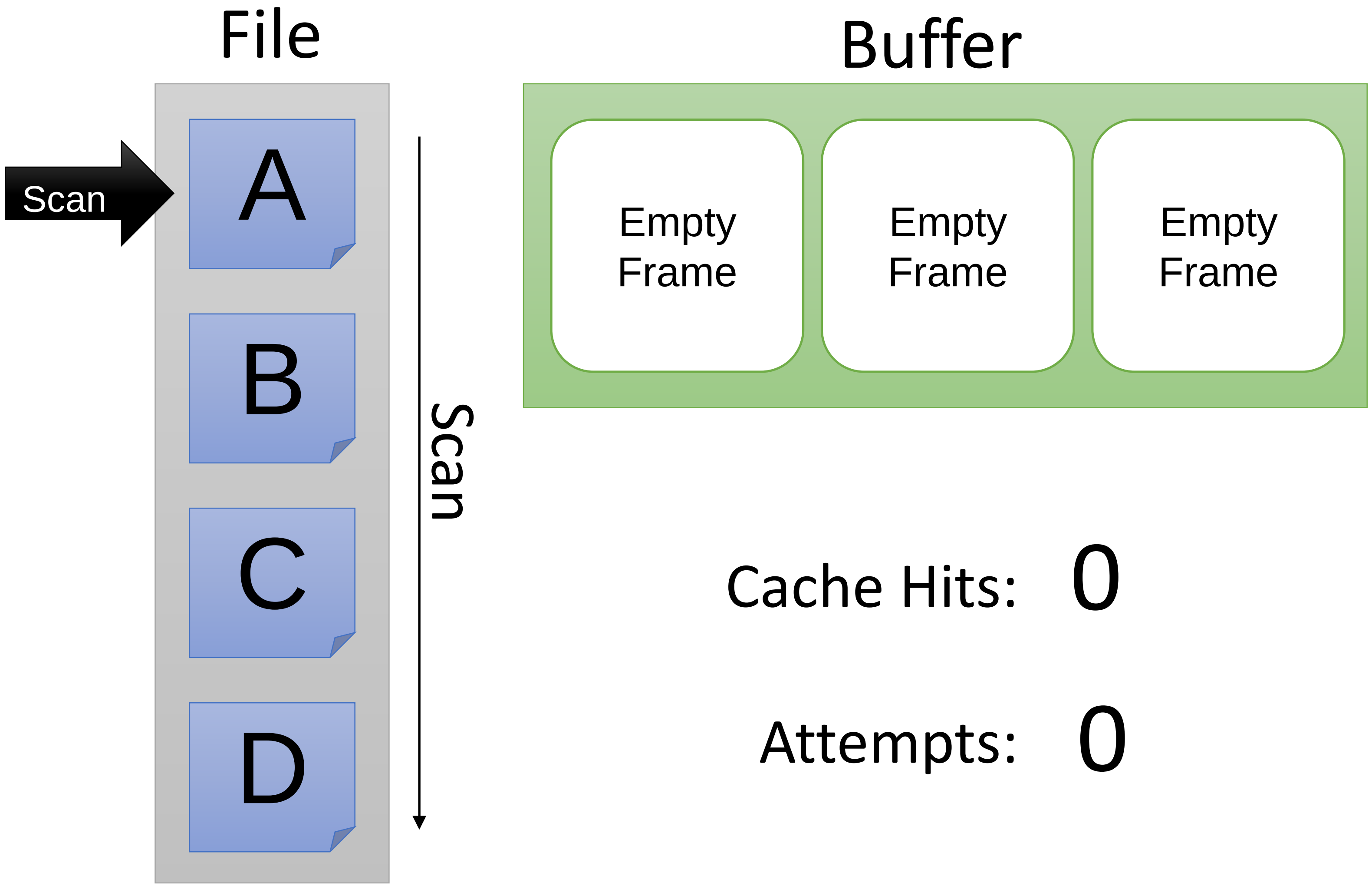
Cache Hits: 0

Attempts: 6

No Cache Hits!

Repeated Scan of Big File **(MRU)**

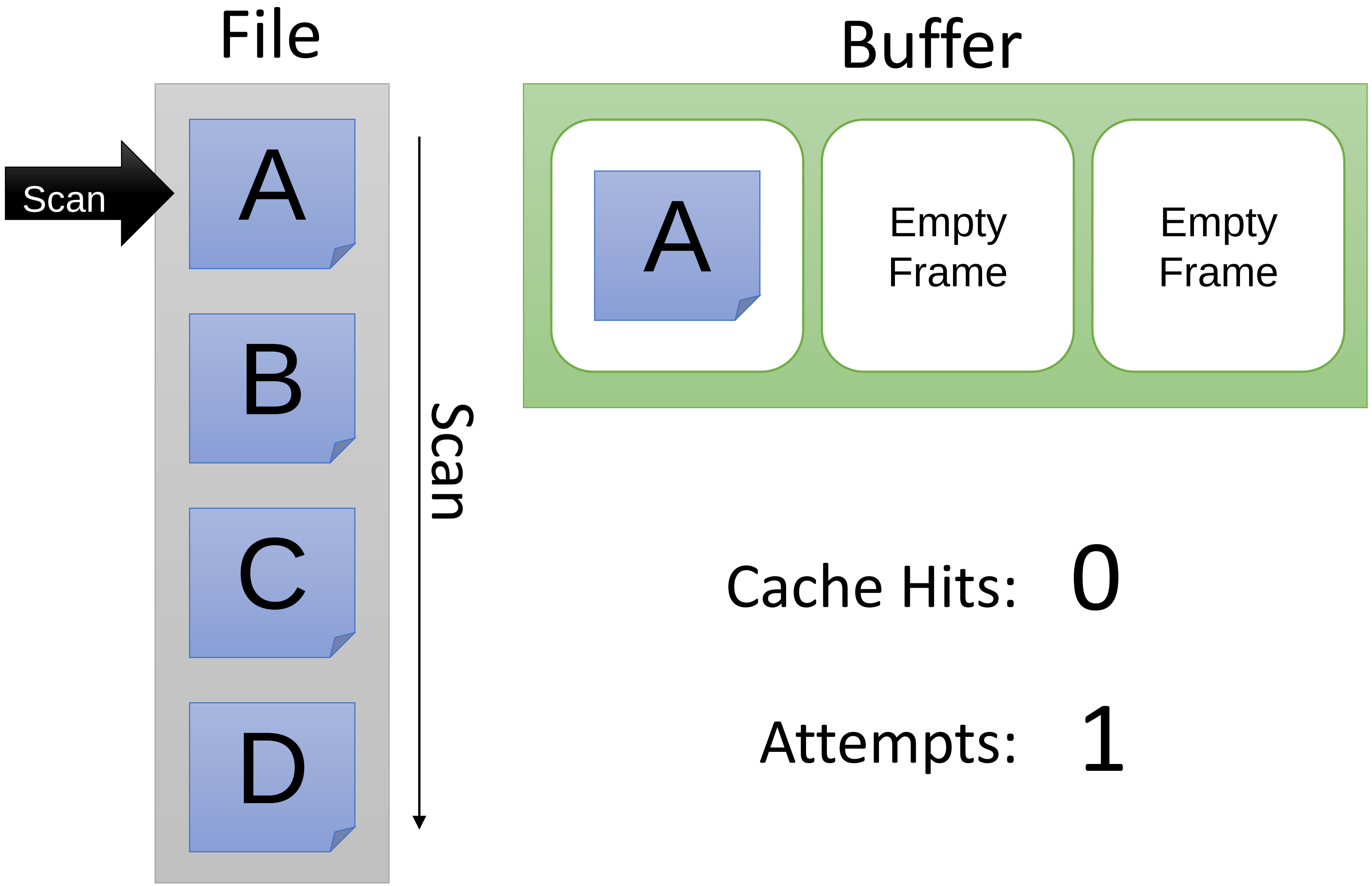
Most Recently Used



Cache Hits: 0

Attempts: 0

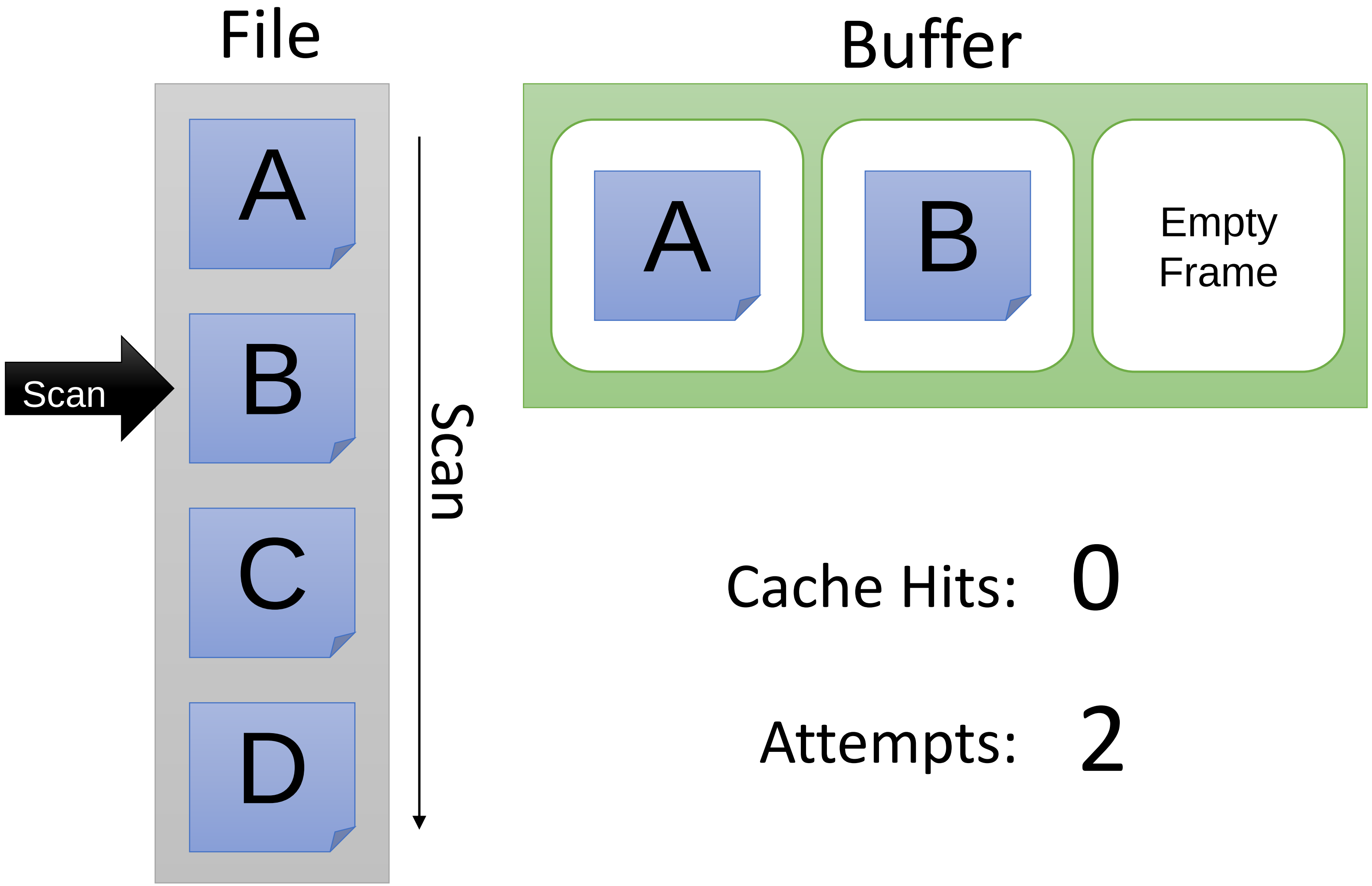
Repeated Scan of Big File (MRU)



Cache Hits: 0

Attempts: 1

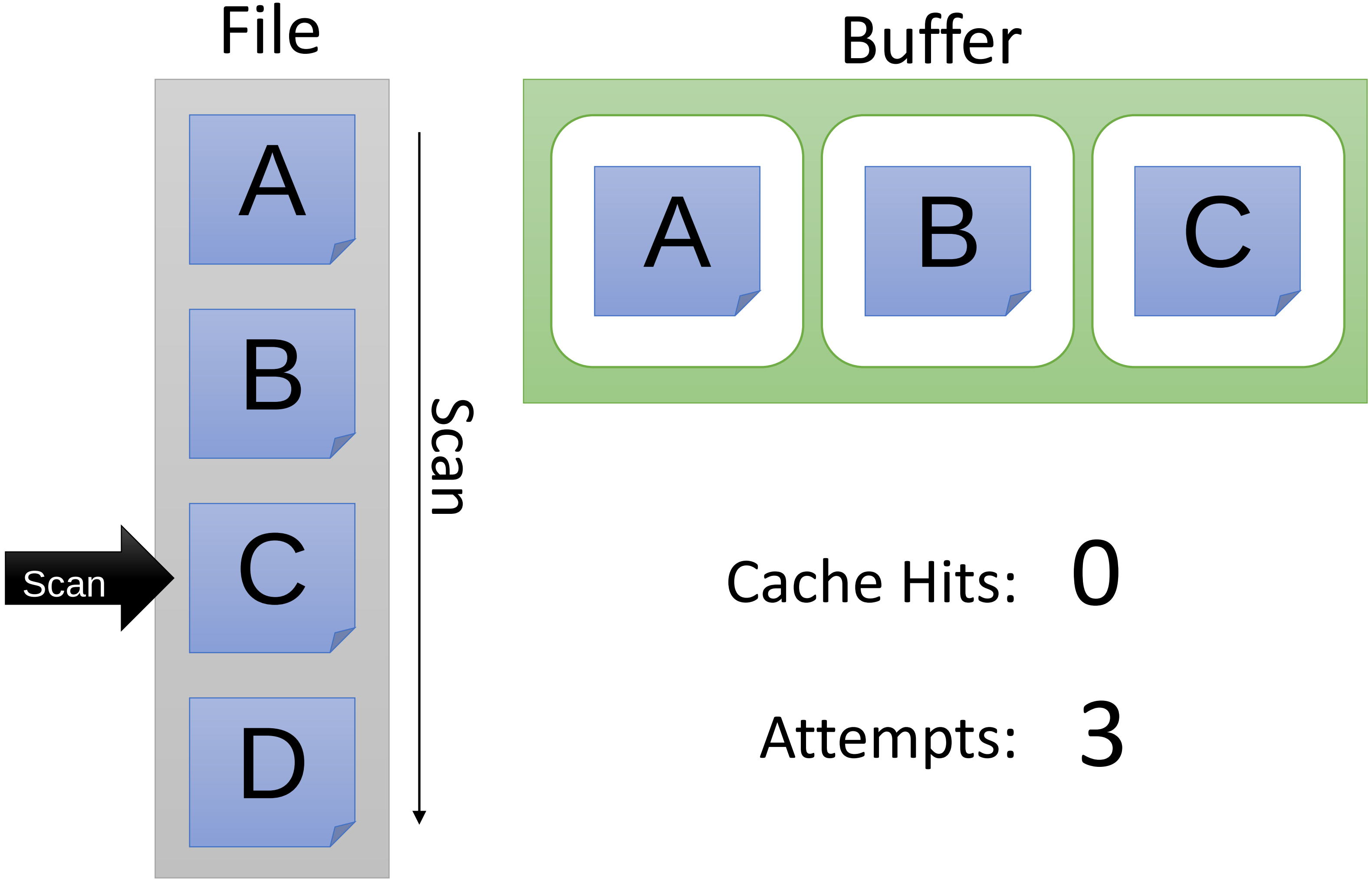
Repeated Scan of Big File (MRU)



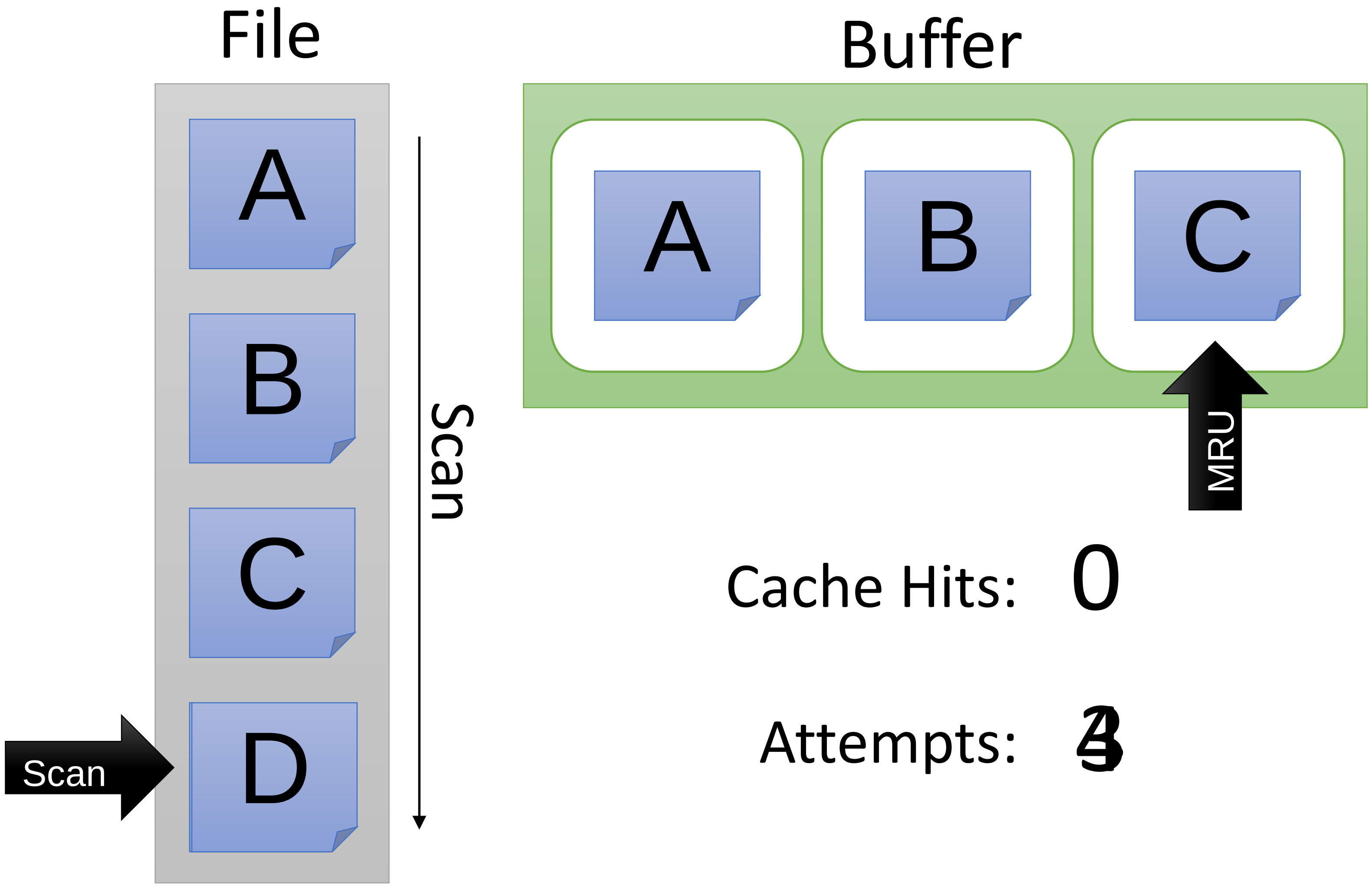
Cache Hits: 0

Attempts: 2

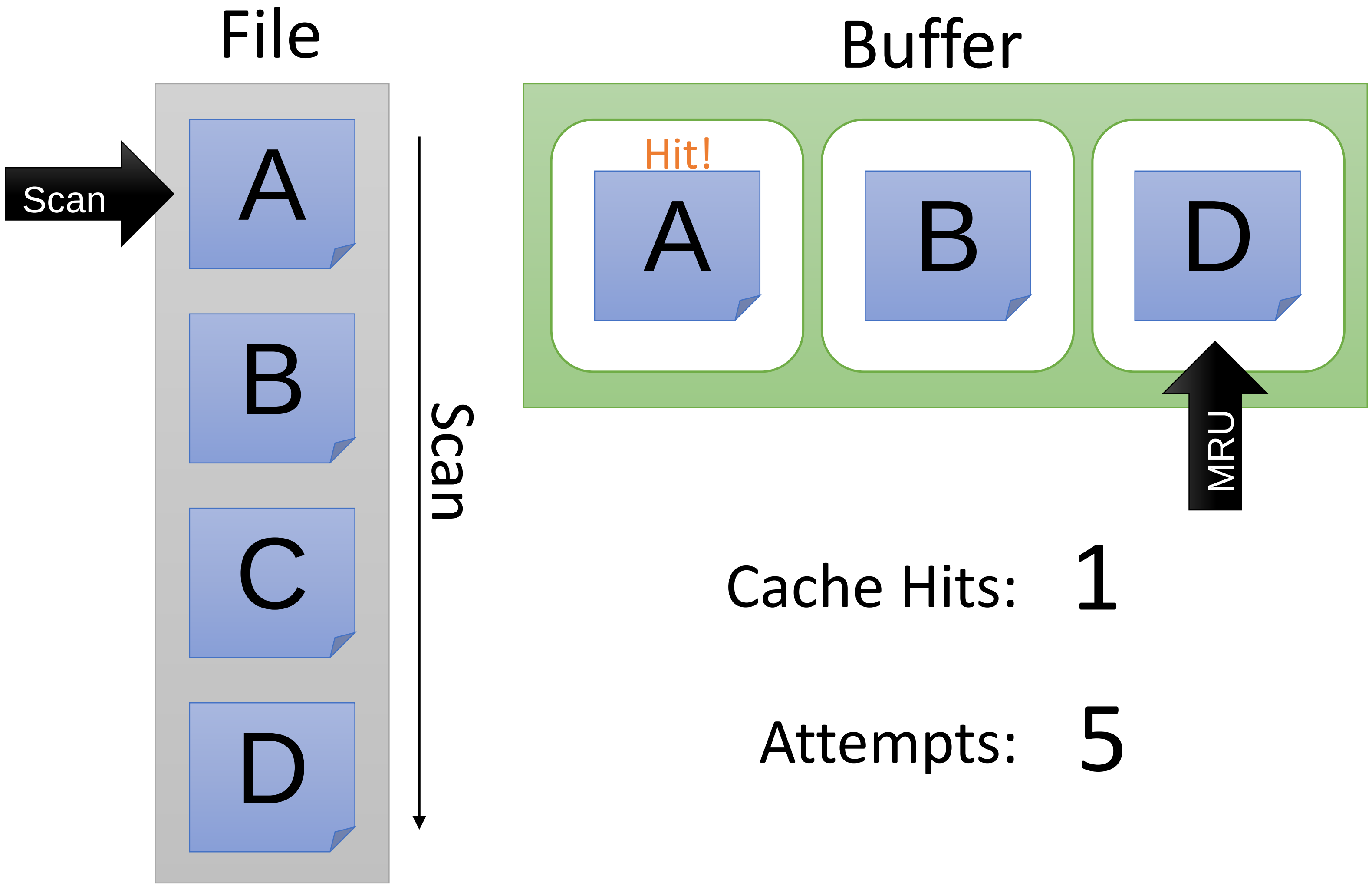
Repeated Scan of Big File (MRU)



Repeated Scan of Big File (MRU)



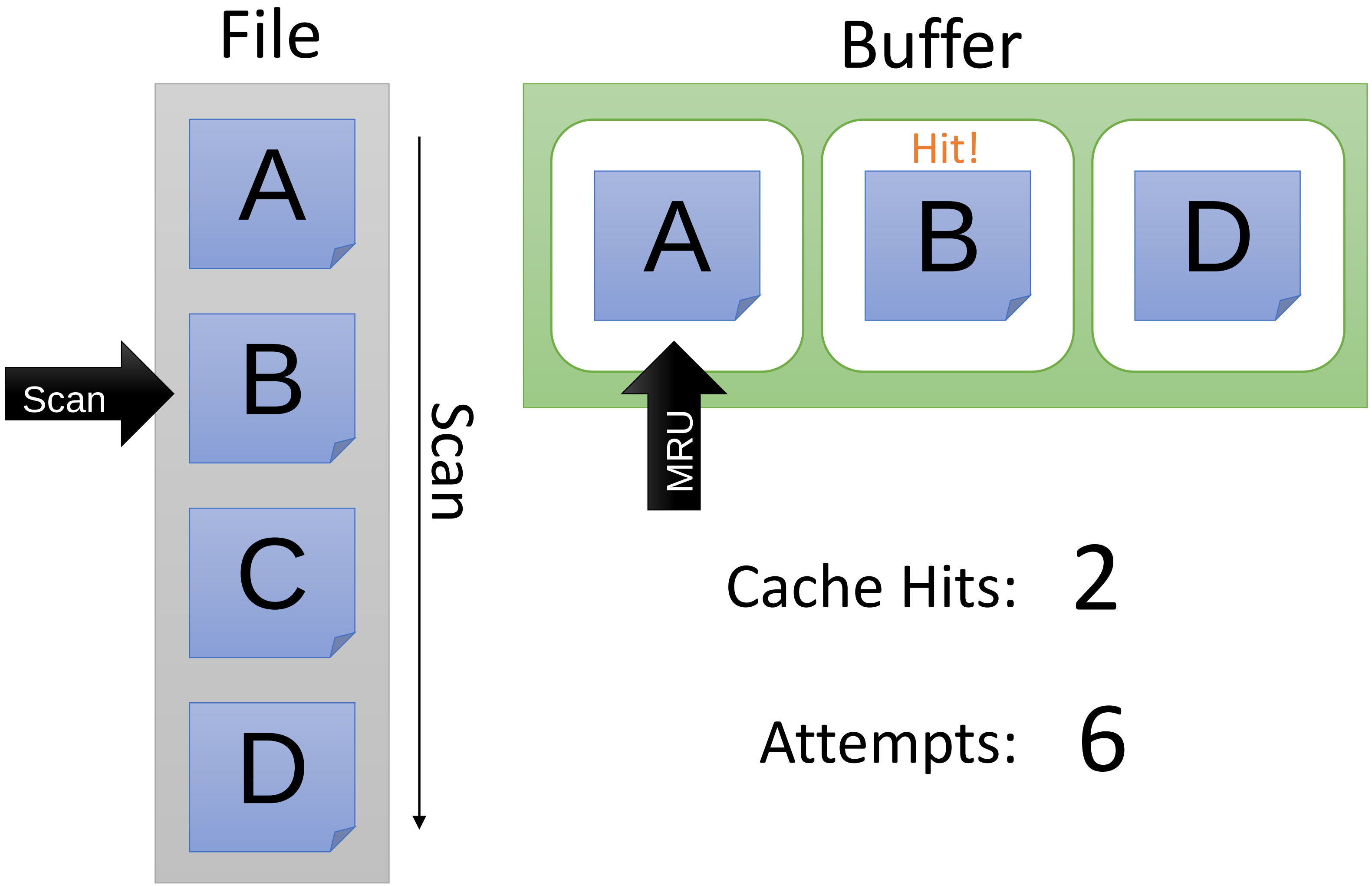
Repeated Scan of Big File (MRU)



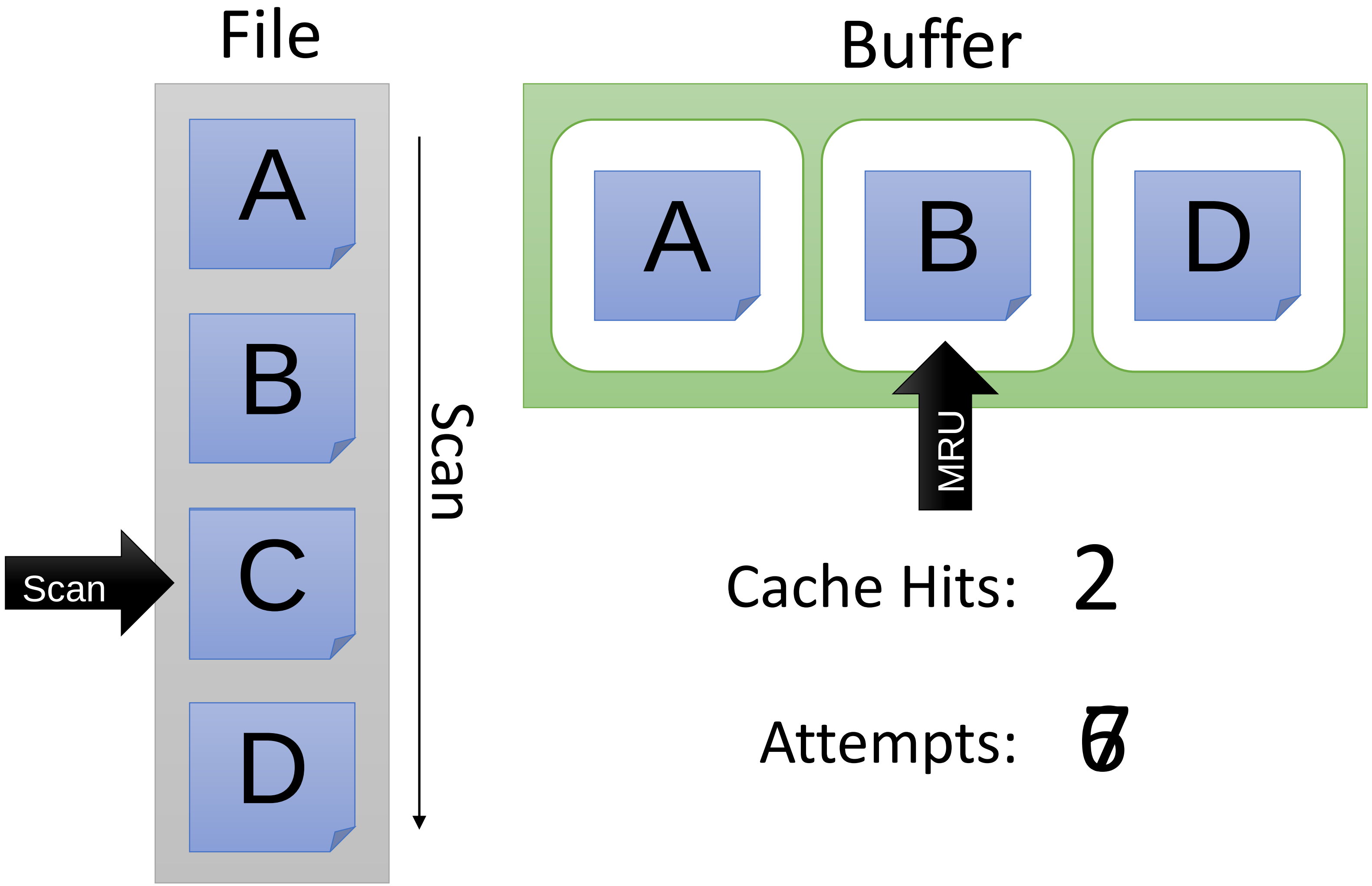
Cache Hits: 1

Attempts: 5

Repeated Scan of Big File (MRU)



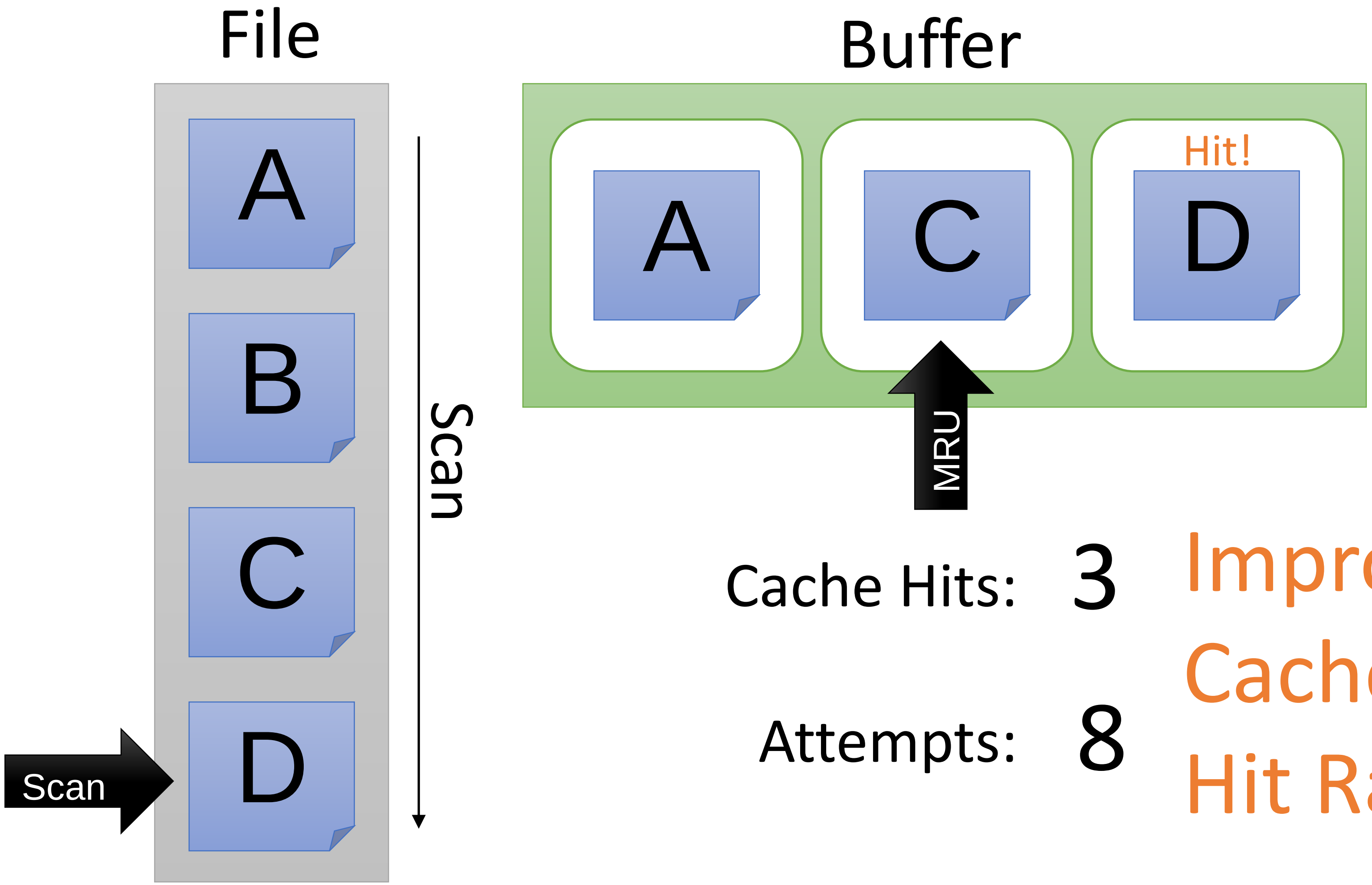
Repeated Scan of Big File (MRU)



Cache Hits: 2

Attempts: 7

Repeated Scan of Big File (MRU)

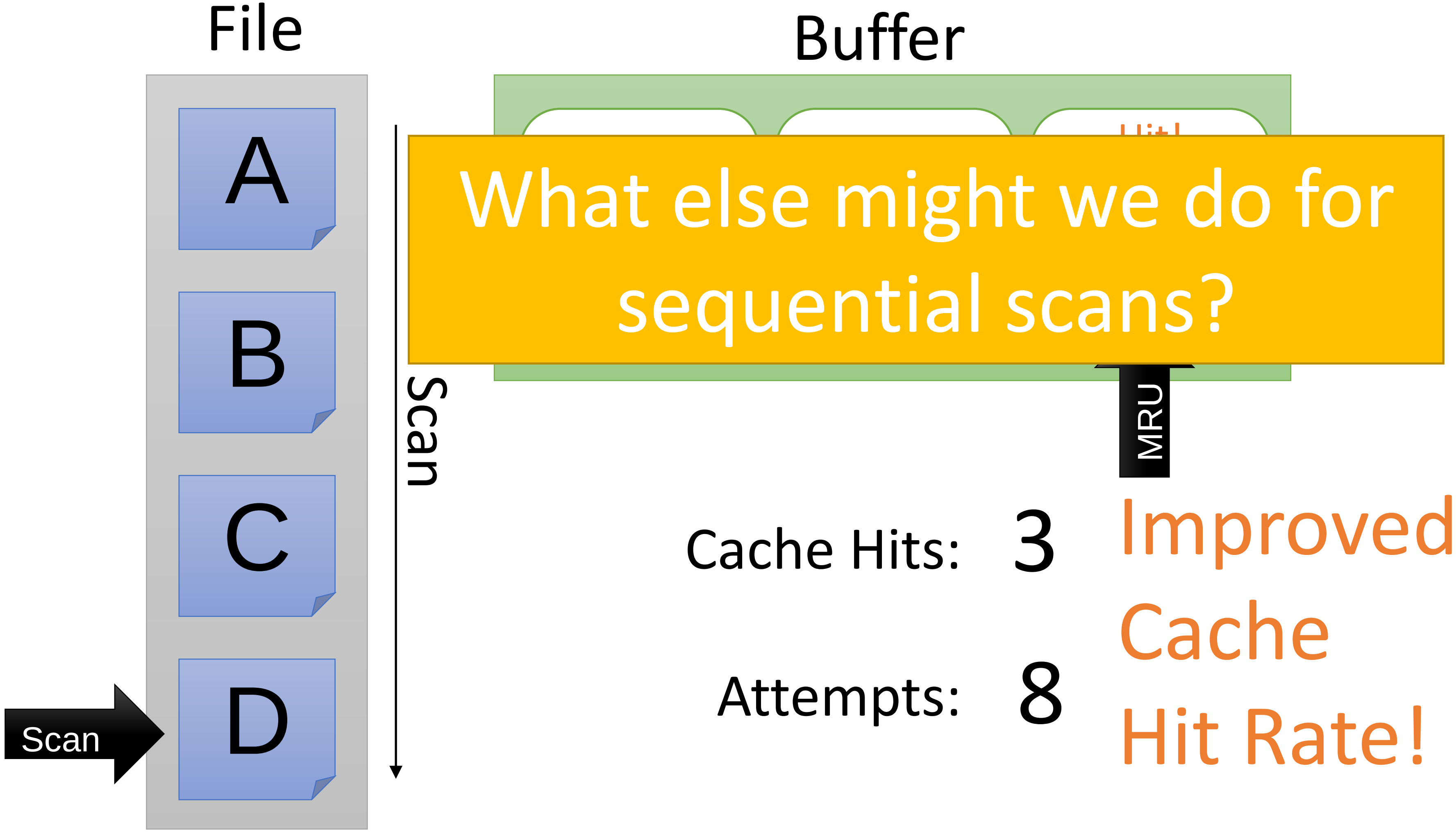


Cache Hits: 3

Attempts: 8

Improved Cache Hit Rate!

Repeated Scan of Big File (MRU)

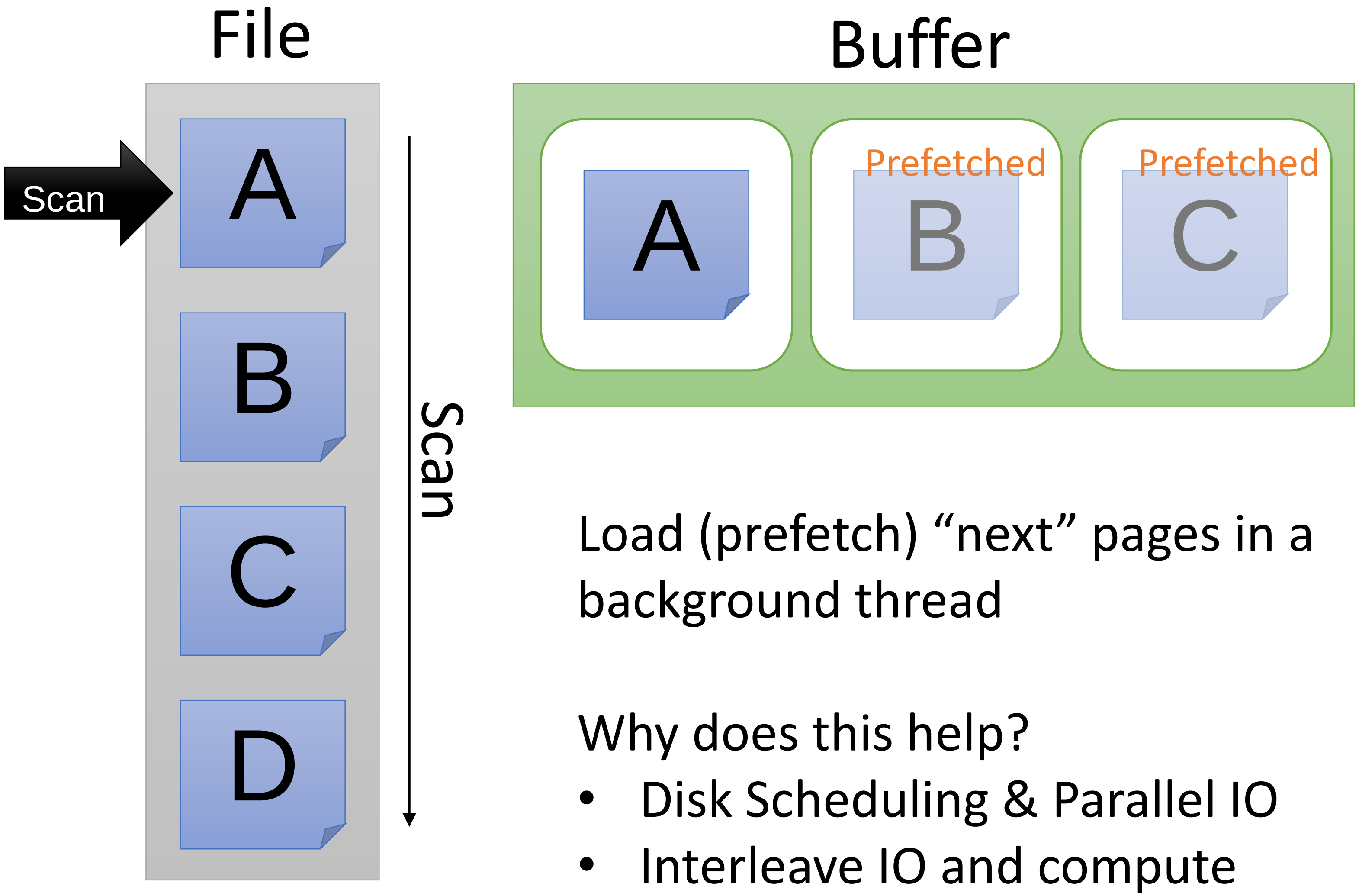


Cache Hits: 3

Attempts: 8

Improved Cache Hit Rate!

Background Prefetching



Load (prefetch) “next” pages in a background thread

Why does this help?

- Disk Scheduling & Parallel IO
- Interleave IO and compute

The Buffer Manager

- A **buffer manager** handles supporting operations for the buffer:
  - Primarily, handles & executes the "replacement policy"
    - i.e. finds a page in buffer to flush/release if buffer is full and a new page needs to be read in
  - DBMSs typically implement their own buffer management routines

3. External Merge Algorithm

Challenge: Merging Big Files with Small Memory

- How do we *efficiently* merge two sorted files when both are much larger than our main memory buffer?

- **Key point:** Disk IO (R/W) dominates the algorithm cost

Our first example of an **“IO aware”** algorithm / cost model

External Merge Algorithm

- **Input**: 2 sorted lists of length M and N

- **Output:** 1 sorted list of length M + N

- **Required:** At least 3 Buffer Pages

- **IOs**: 2(M+N)

Key (Simple) Idea

To find an element that is no larger than all elements in two lists, one only needs to compare minimum elements from each list.

If:

$$A_1 \leq A_2 \leq \cdots \leq A_N$$
$$B_1 \leq B_2 \leq \cdots \leq B_M$$

Then:

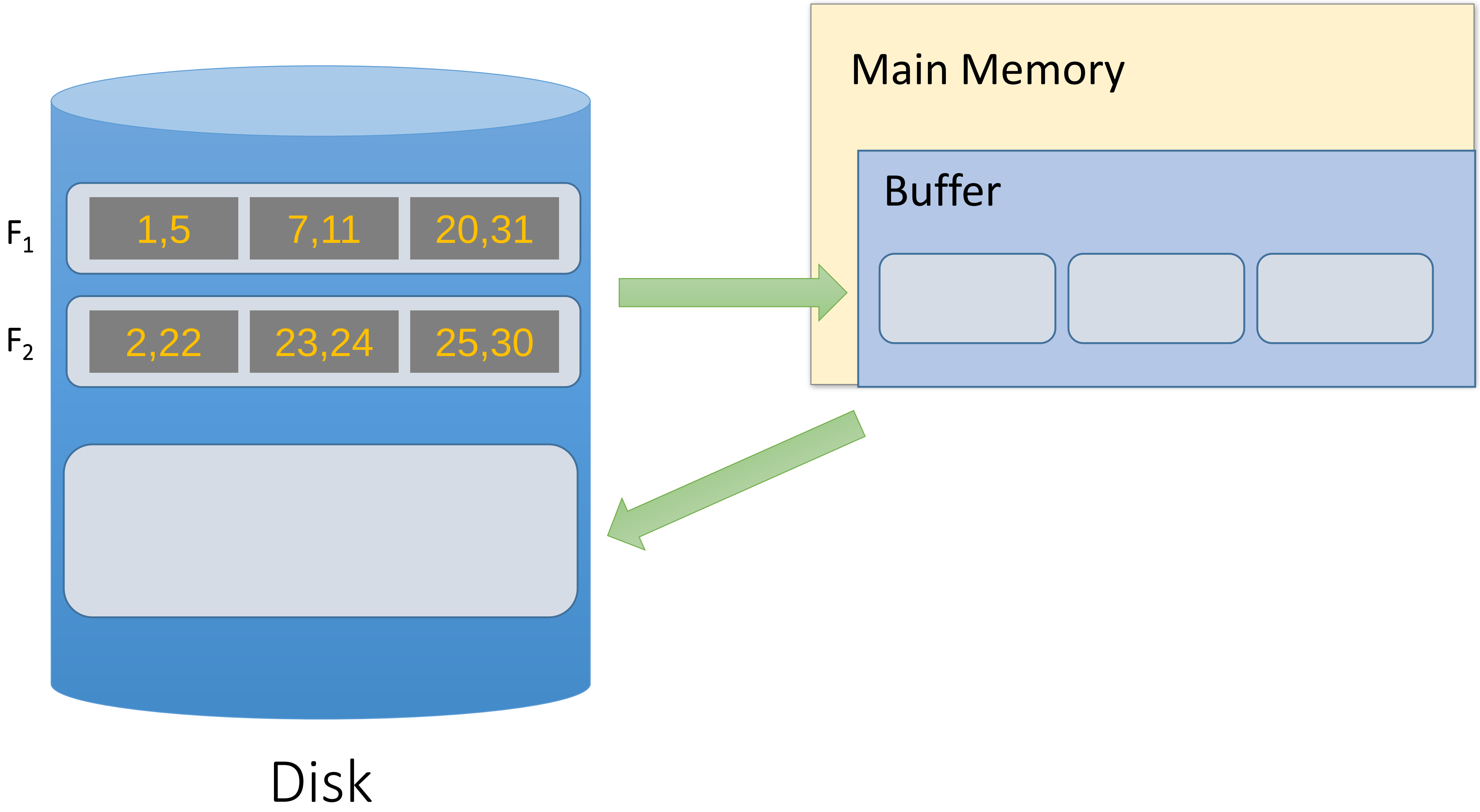
$$Min(A_1, B_1) \leq A_i$$
$$Min(A_1, B_1) \leq B_j$$

for i=1….N and j=1….M

External Merge Algorithm

Input:
Two sorted
files

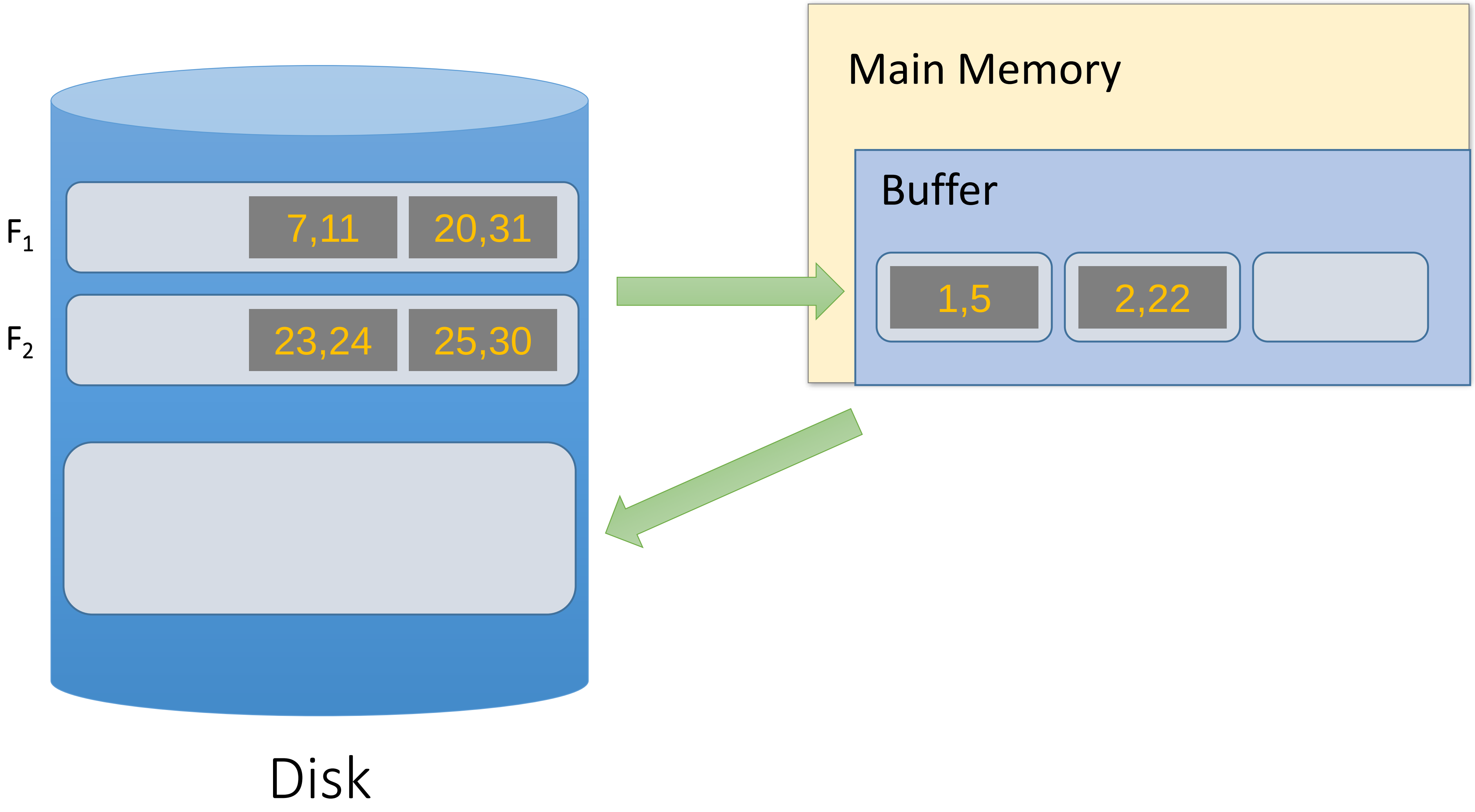
Output:
One *merged*
sorted file



External Merge Algorithm

Input:
Two sorted files

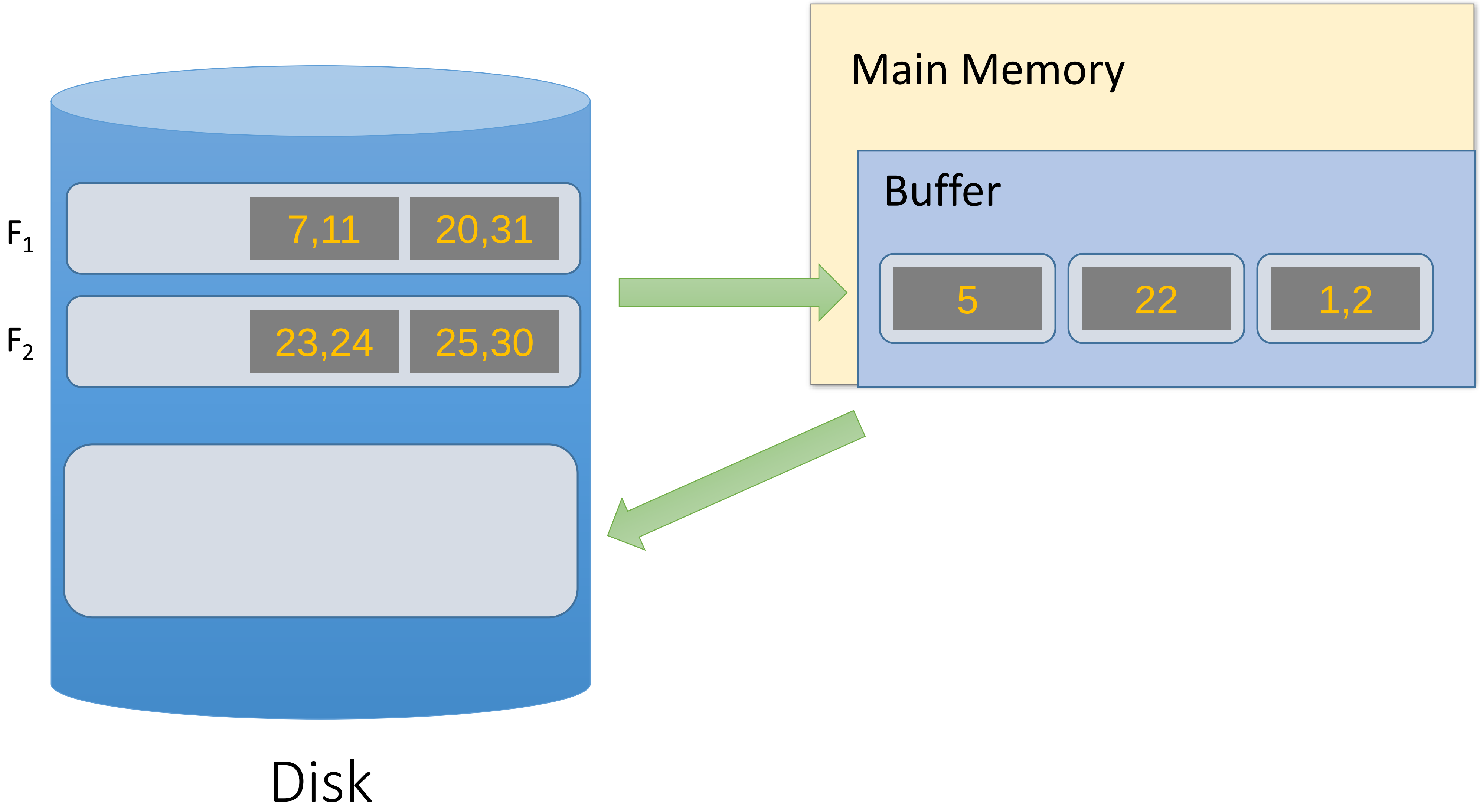
Output:
One *merged* sorted file



External Merge Algorithm

Input:
Two sorted
files

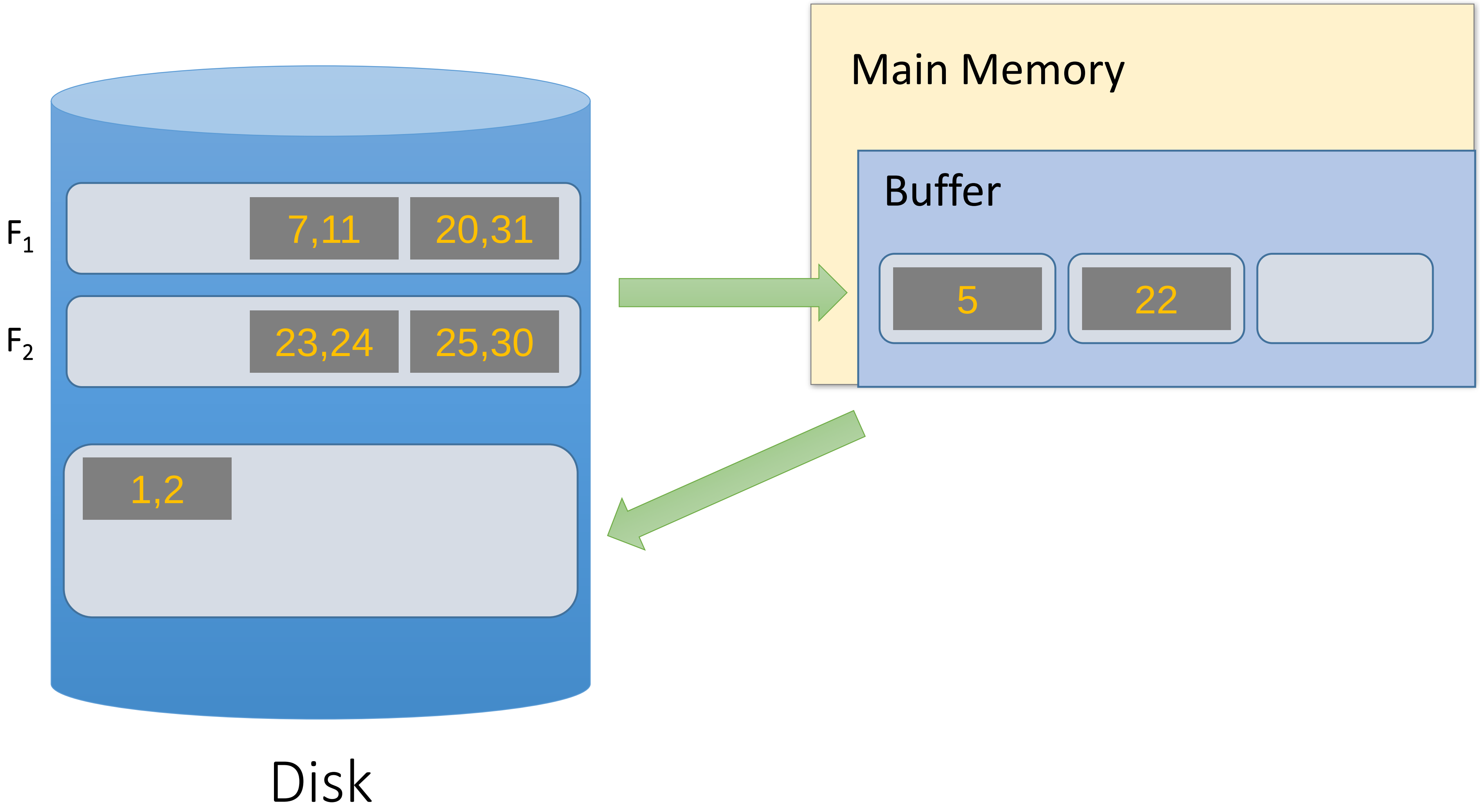
Output:
One *merged*
sorted file



External Merge Algorithm

Input:
Two sorted
files

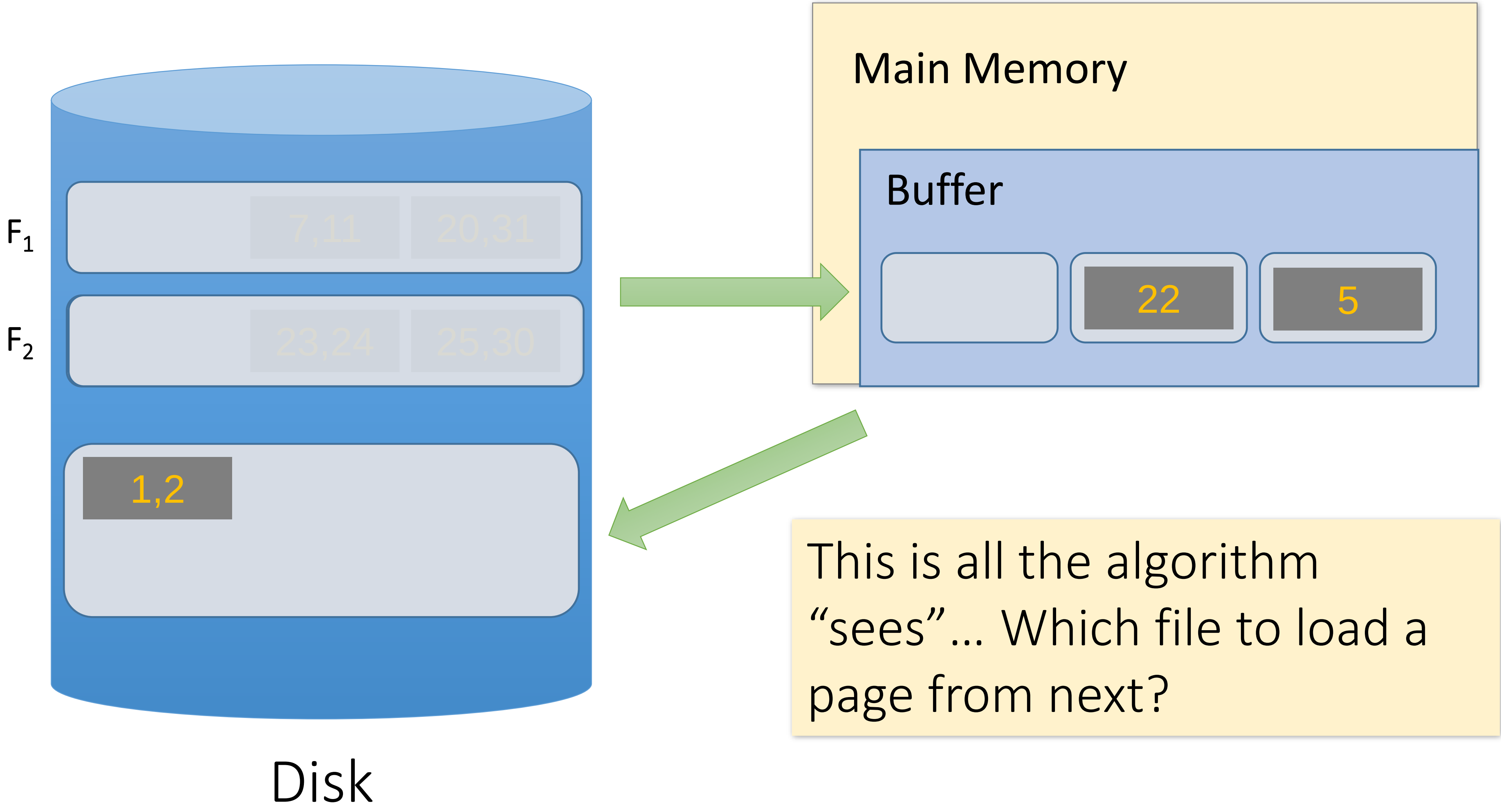
Output:
One *merged*
sorted file



External Merge Algorithm

Input:
Two sorted files

Output:
One *merged* sorted file

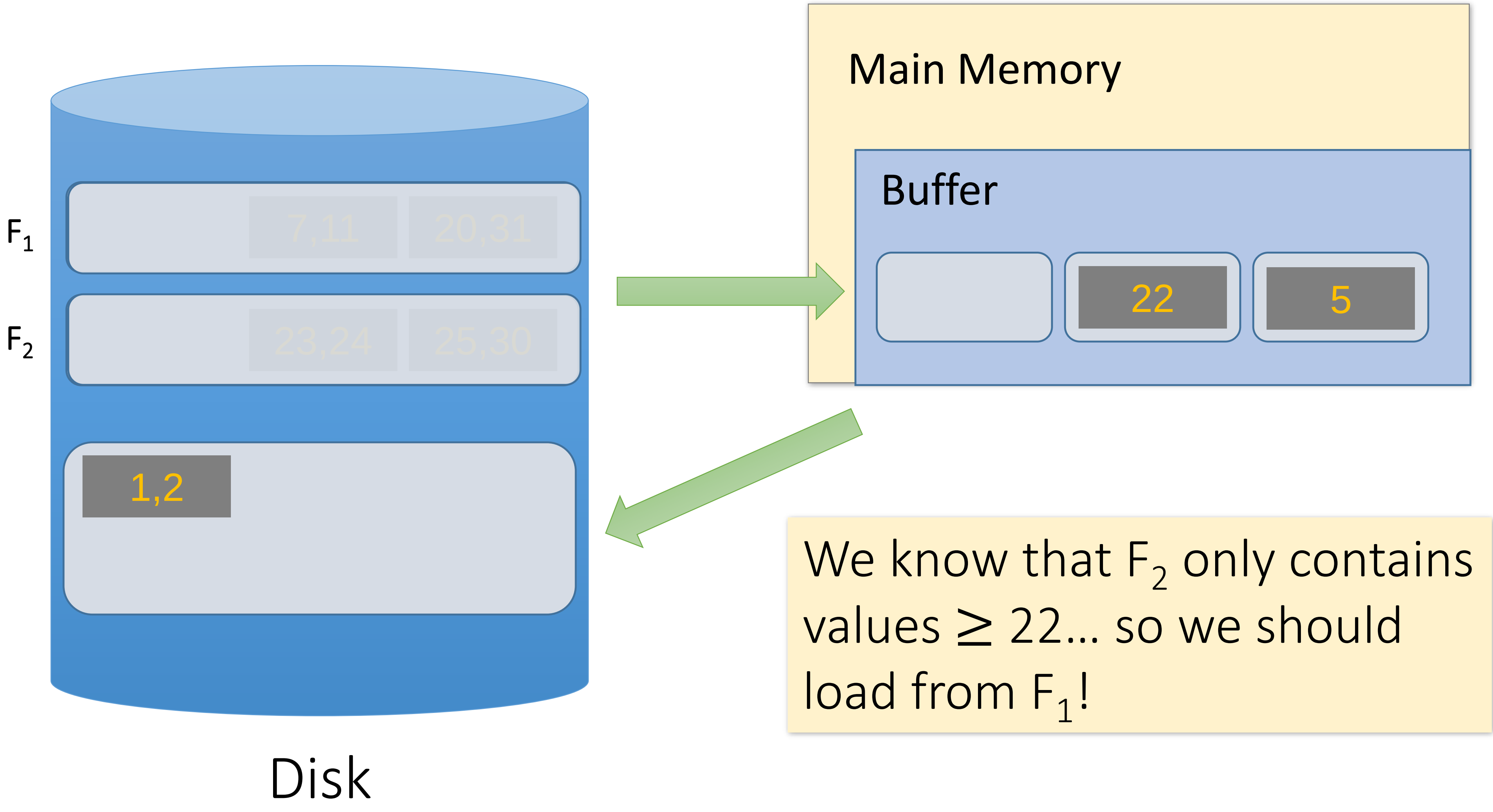


This is all the algorithm "sees"… Which file to load a page from next?

External Merge Algorithm

Input:
Two sorted files

Output:
One *merged* sorted file

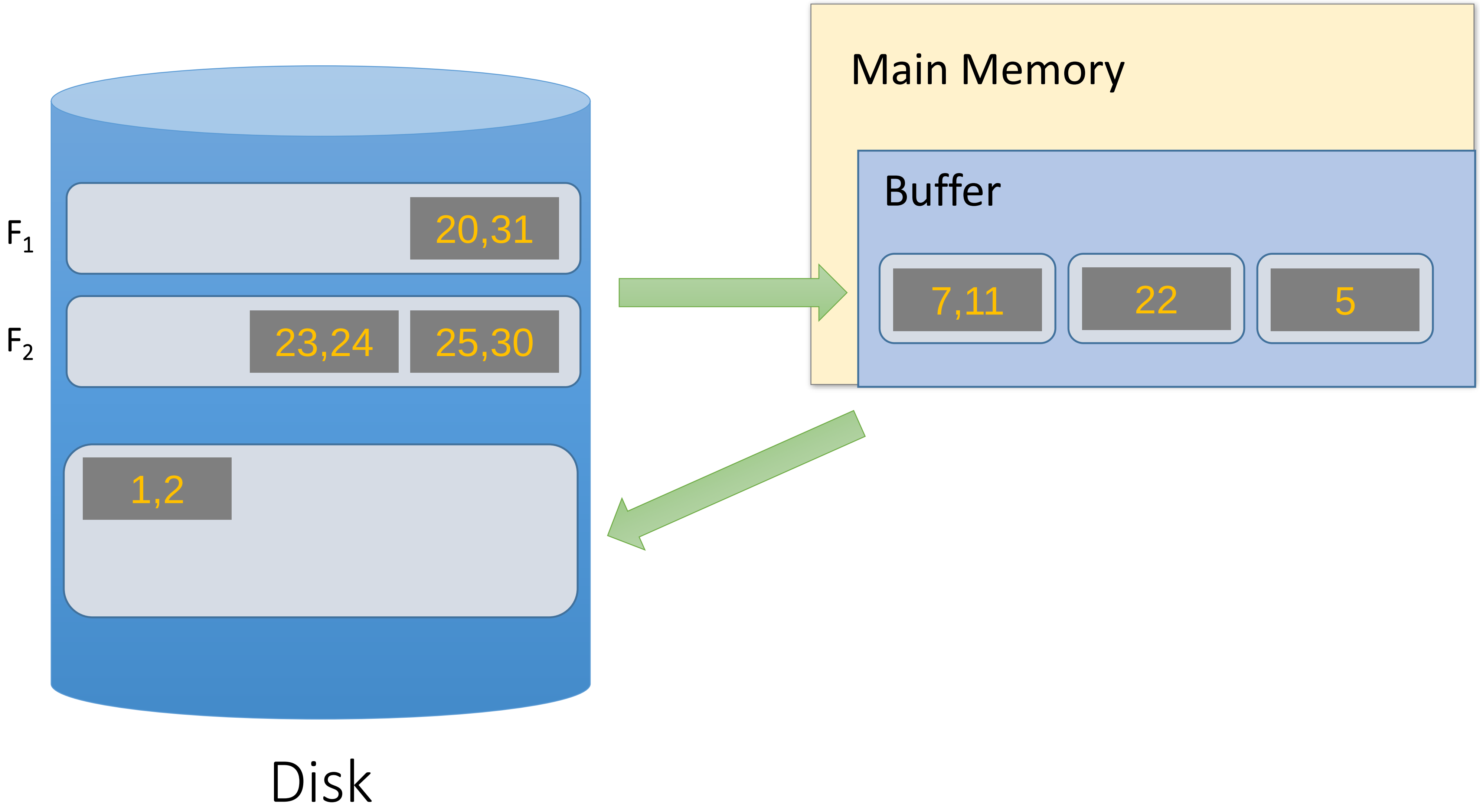


We know that F$_2$ only contains values ≥ 22… so we should load from F$_1$!

External Merge Algorithm

Input:
Two sorted
files

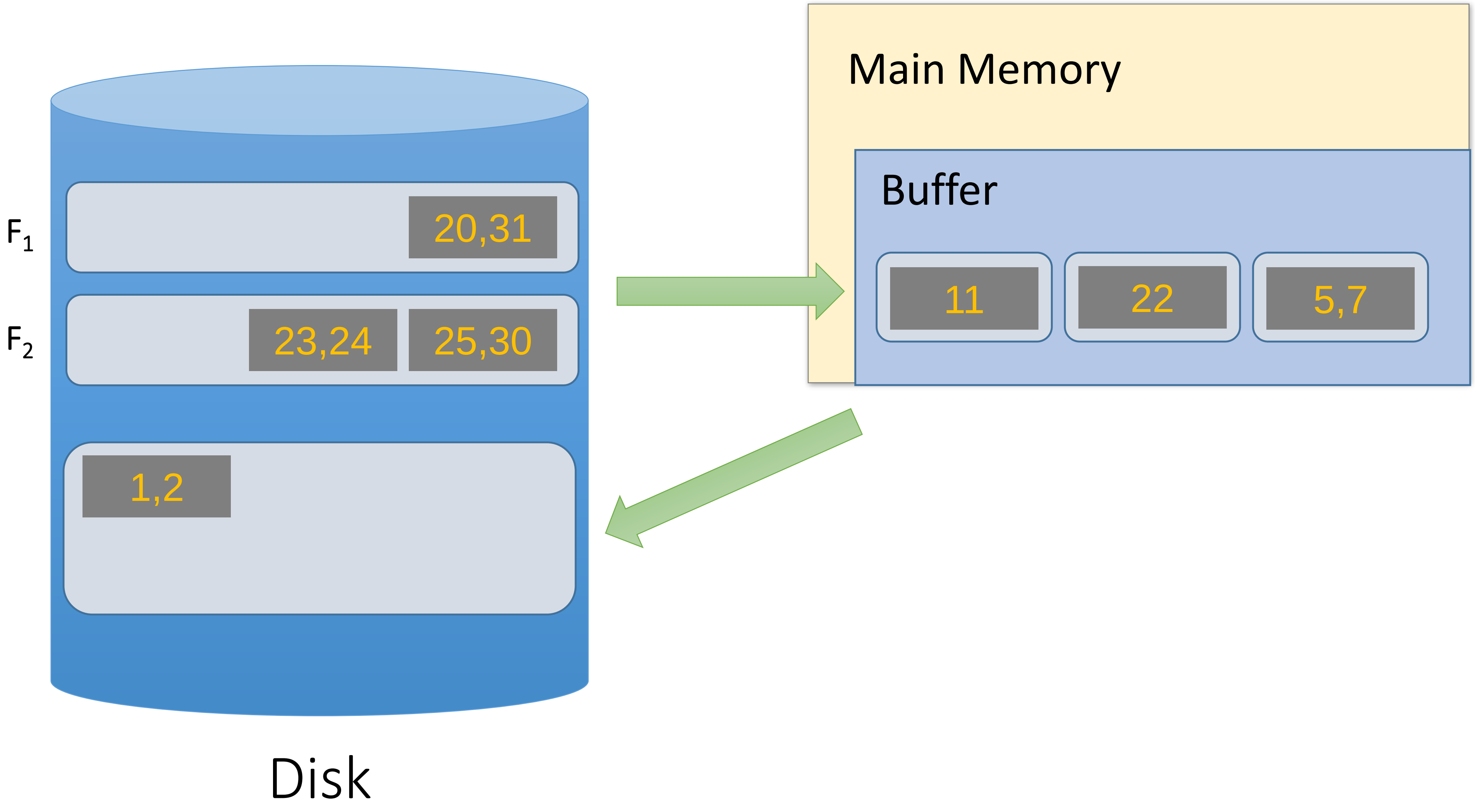
Output:
One *merged*
sorted file



External Merge Algorithm

Input:
Two sorted
files

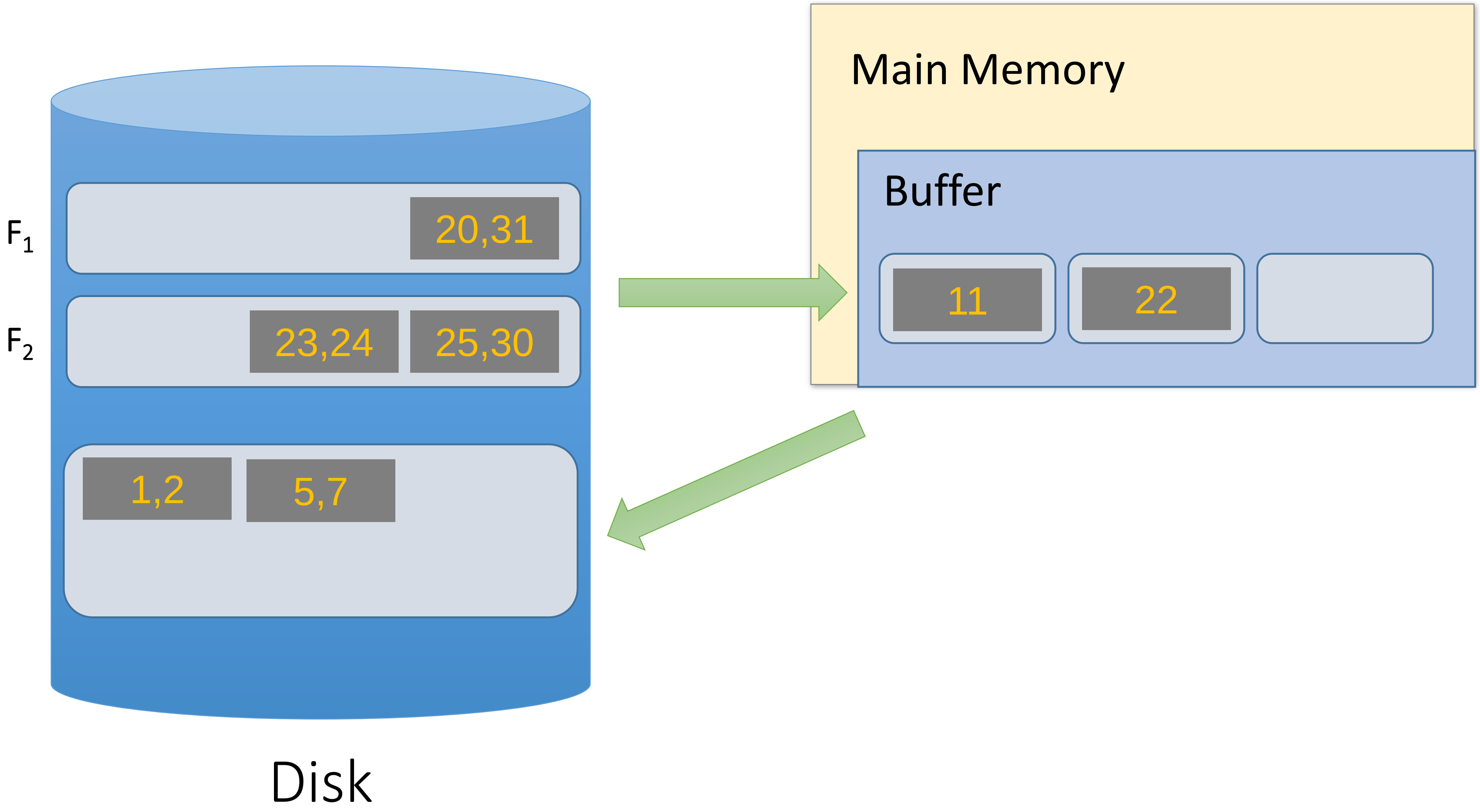
Output:
One *merged*
sorted file



External Merge Algorithm

Input:
Two sorted
files

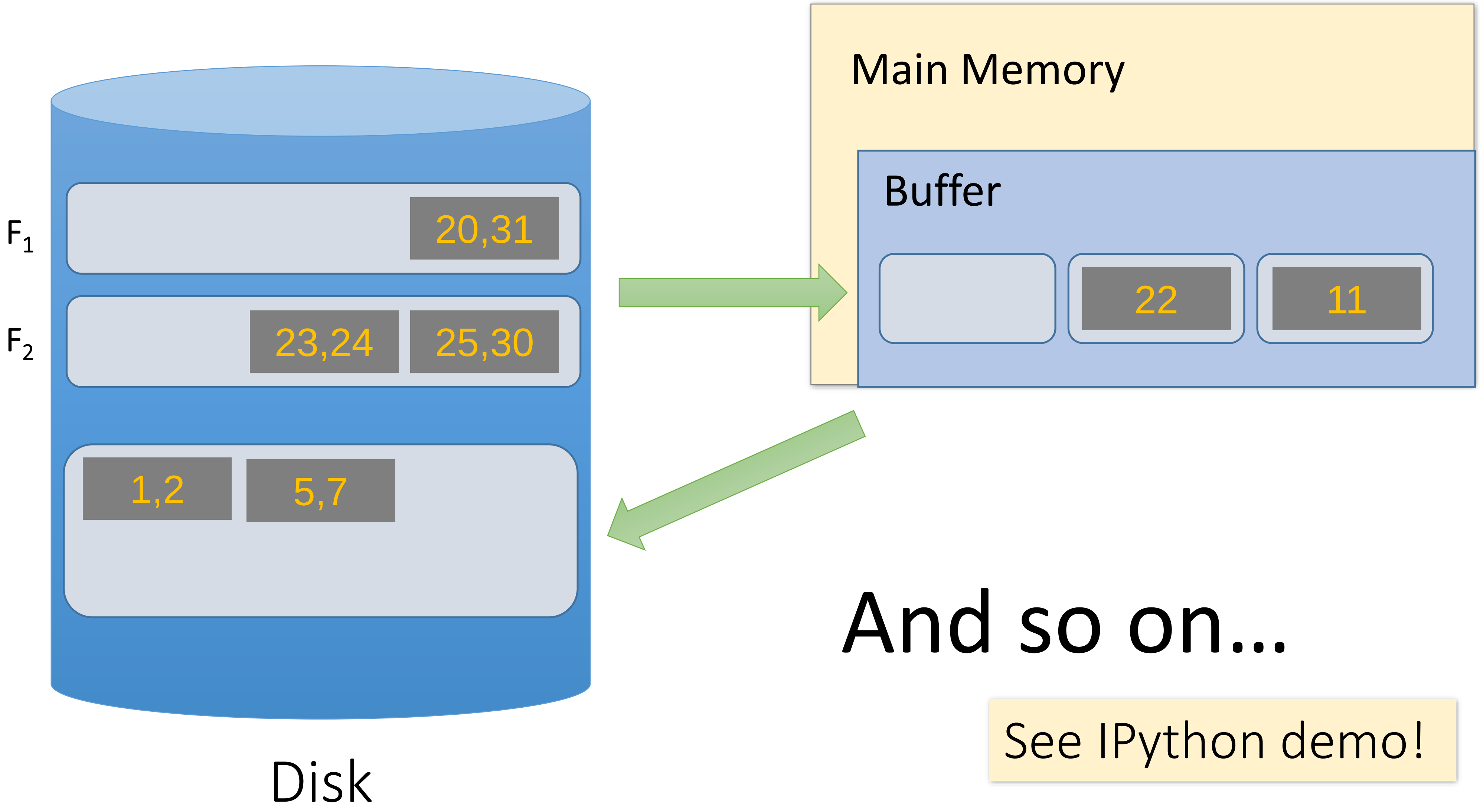
Output:
One *merged*
sorted file



External Merge Algorithm

Input:
Two sorted
files

Output:
One *merged*
sorted file



And so on…

See IPython demo!

We can merge lists of **arbitrary length** with *only* 3 buffer pages.

If lists of size M and N, then

Cost: 2(M+N) IOs

Each page is read once, written once

With B+1 buffer pages, can merge B lists. How?

Recap: External Merge Algorithm

- Suppose we want to merge two **sorted** files both much larger than main memory (i.e. the buffer)

- We can use the **external merge algorithm** to merge files of ***arbitrary length*** in **2*(N+M) IO** operations with only **3 buffer pages**!

Our first example of an "IO aware" algorithm / cost model

4. External Merge Sort

Why are Sort Algorithms Important?

- Data requested from DB in sorted order is **extremely common**
  - e.g., find students in increasing GPA order

- **Why not just use quicksort in main memory??**
  - What about if we need to sort 1TB of data with 1GB of RAM…

A classic problem in computer science!

More reasons to sort…

- Sorting useful for eliminating *duplicate copies* in a collection of records (Why?)

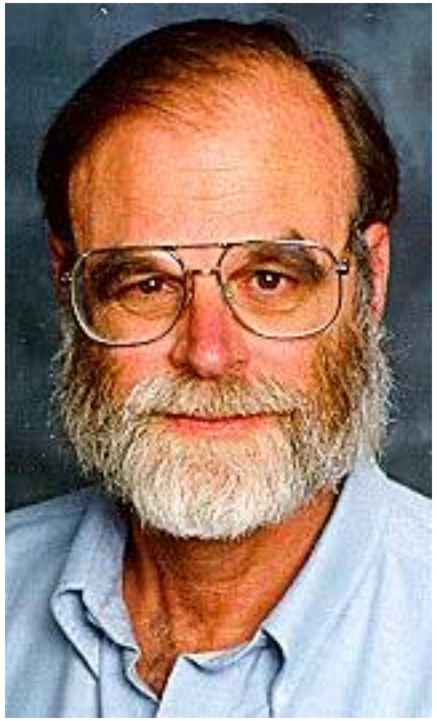
- Sorting is first step in *bulk loading* B+ tree index.

- *Sort-merge* join algorithm involves sorting

Next lectures

Do people care?

http://sortbenchmark.org



Sort benchmark bears his name

So how do we sort big files?

1. Split into chunks small enough to **sort in memory *("runs")***

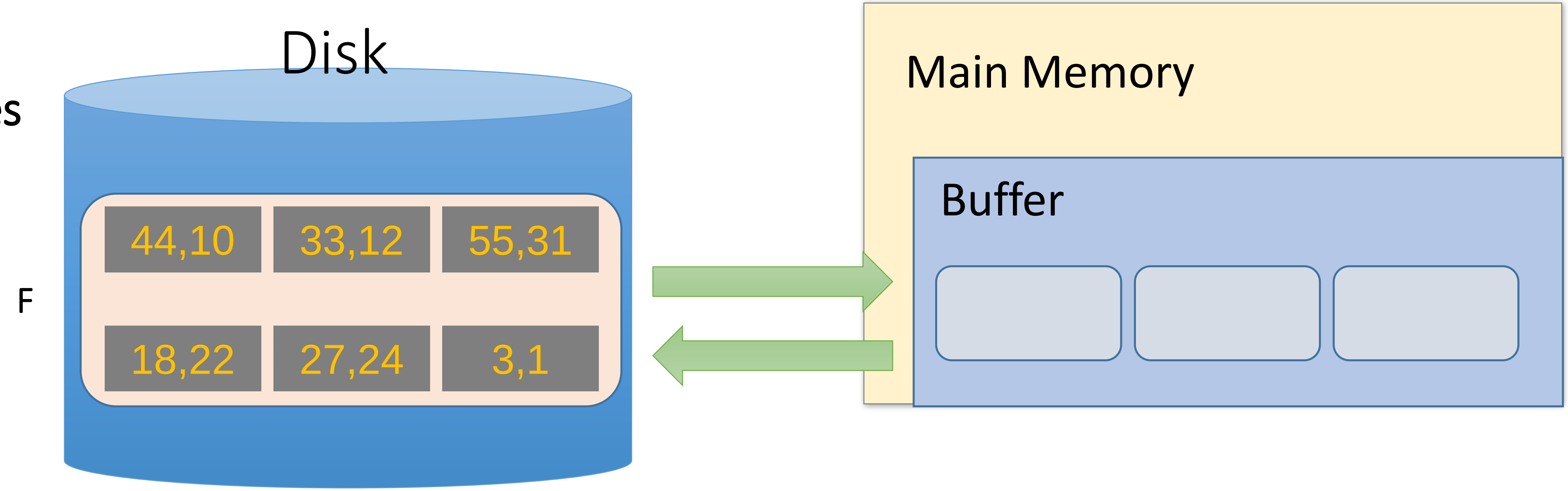
2. **Merge** pairs (or groups) of runs ***using the external merge algorithm***

3. **Keep merging** the resulting runs ***(each time = a "pass")*** until left with one sorted file!

External Merge Sort Algorithm

Example:
- 3 Buffer pages
- 6-page file

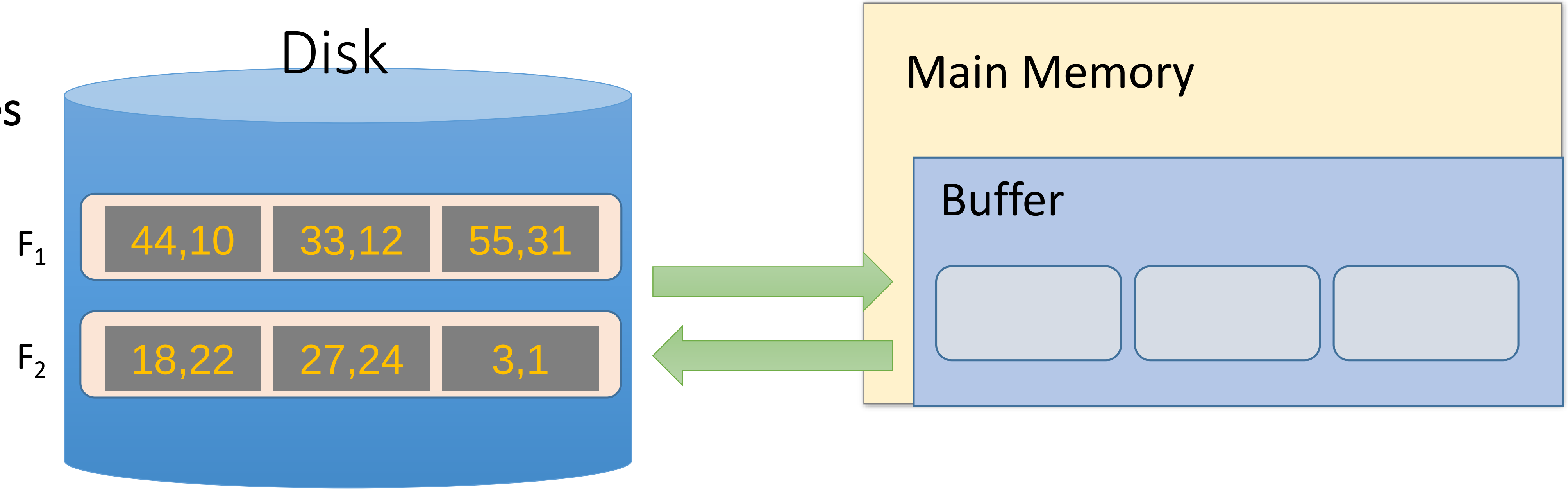
Orange file = unsorted



1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm

Example:
- 3 Buffer pages
- 6-page file



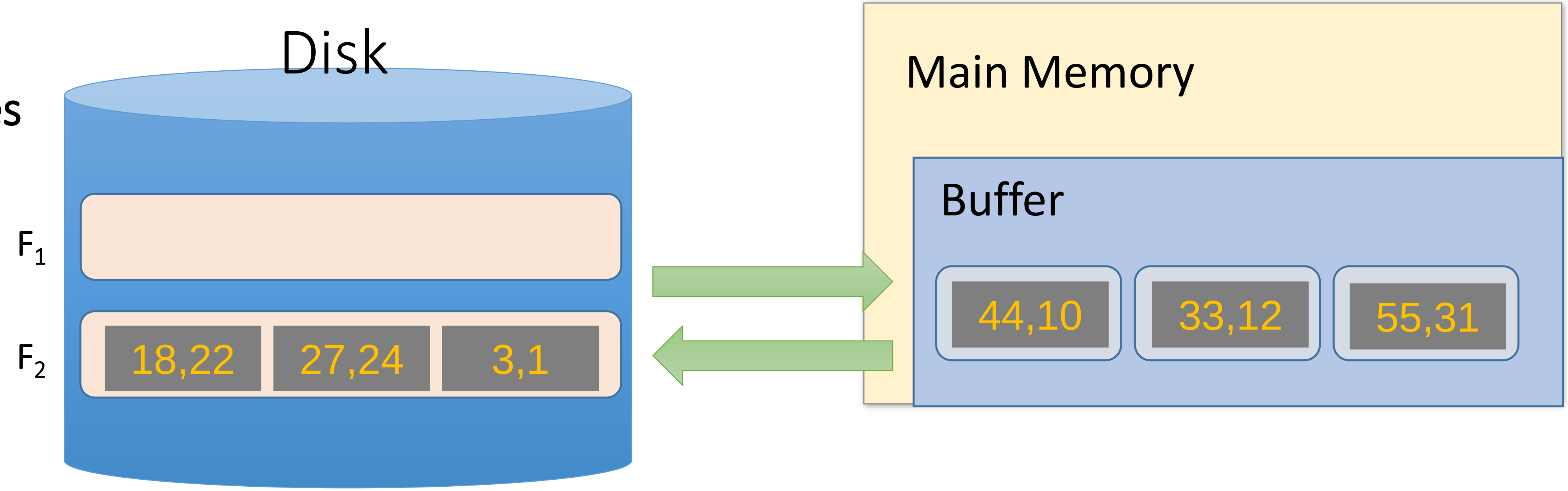
Orange file = unsorted

1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm

Example:
- 3 Buffer pages
- 6-page file

Orange file = unsorted

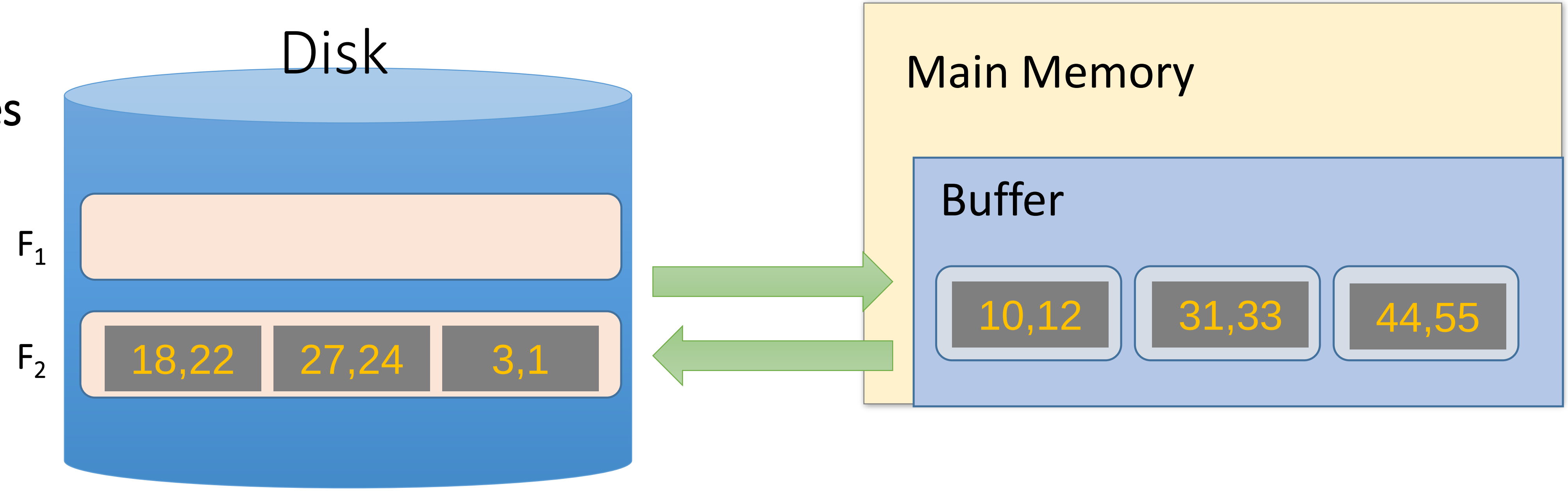


1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm

Example:

- 3 Buffer pages
- 6-page file



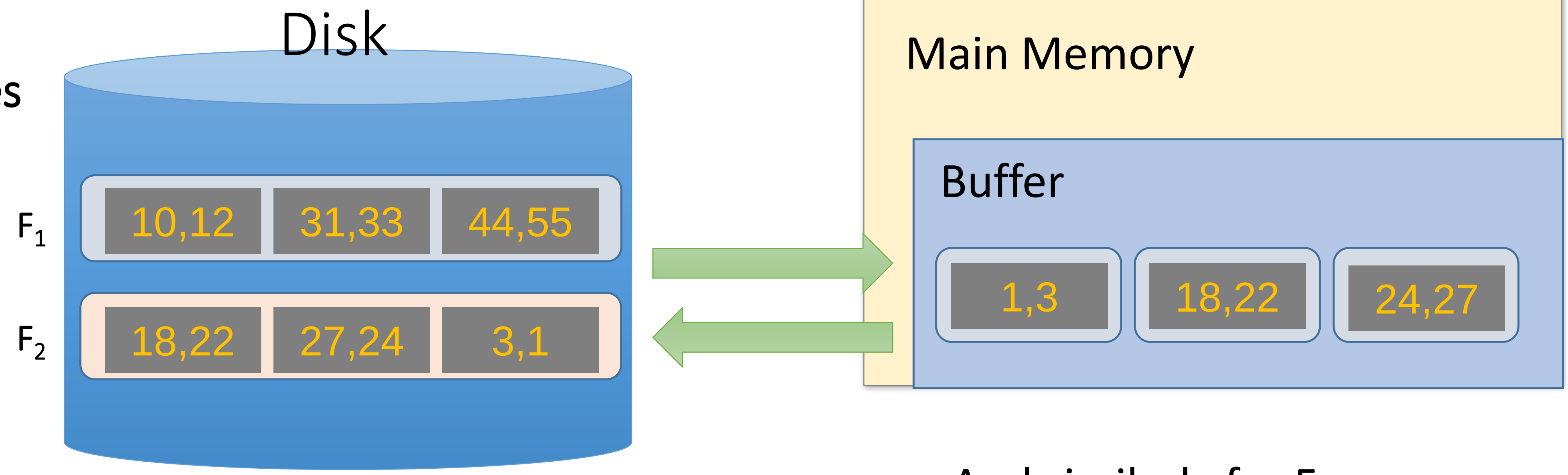
Orange file = unsorted

1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm

Example:
- 3 Buffer pages
- 6-page file

Each sorted file is a called a *run*

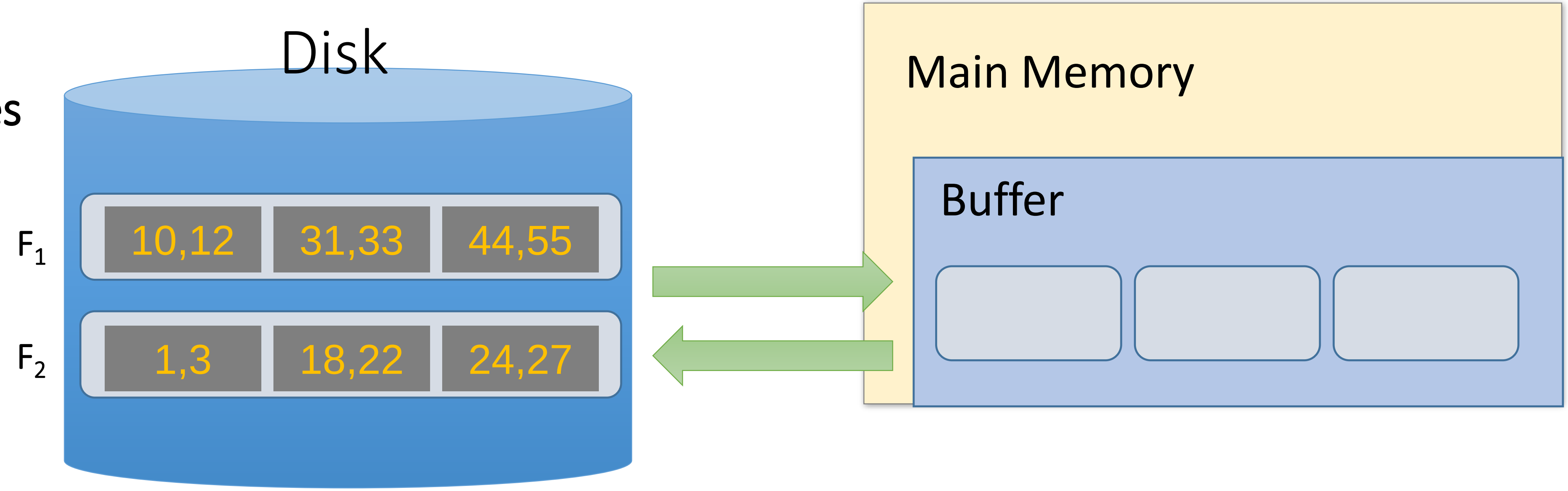


And similarly for F_2

1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm

Example:
- 3 Buffer pages
- 6-page file



2. Now just run the **external merge** algorithm & we're done!

Calculating IO Cost

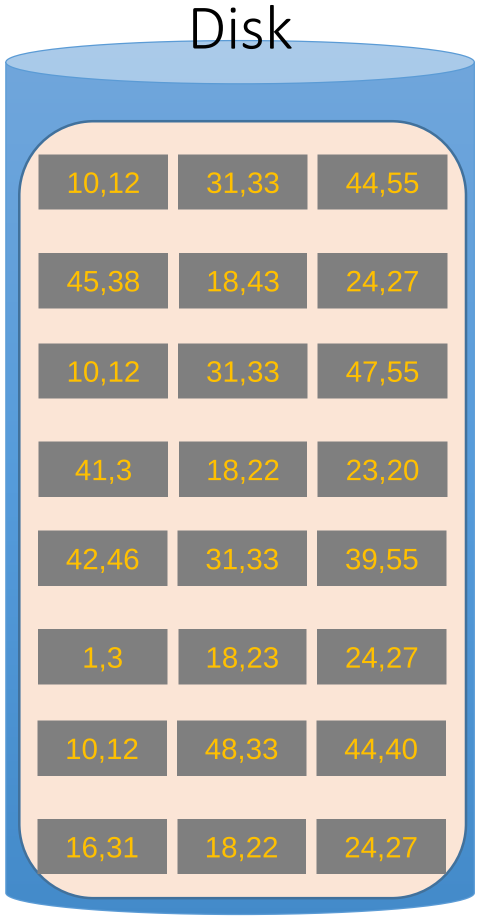
For 3 buffer pages, 6 page file:

1. Split into **<u>two 3-page files</u>** and **sort in memory**
   1. **= 1 R + 1 W for each page = 2*(3 + 3) = 12 IO operations**

2. **Merge** each pair of sorted chunks ***using the external merge algorithm***
   1. **= 2*(3 + 3) = 12 IO operations**

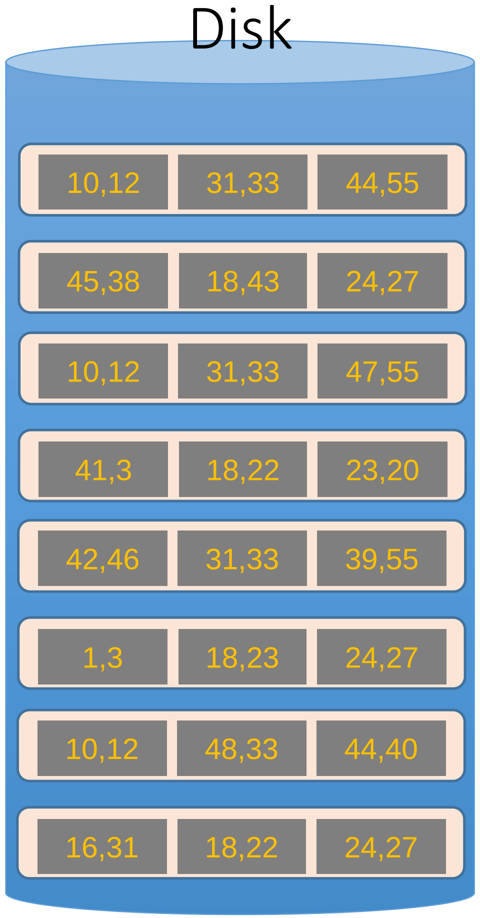
3. **Total cost = 24 IO**

Running External Merge Sort on Larger Files



Assume we still only have *3* buffer pages *(Buffer not pictured)*

Running External Merge Sort on Larger Files



1. Split into files small enough to sort in buffer…

Assume we still only have *3* buffer pages *(Buffer not pictured)*

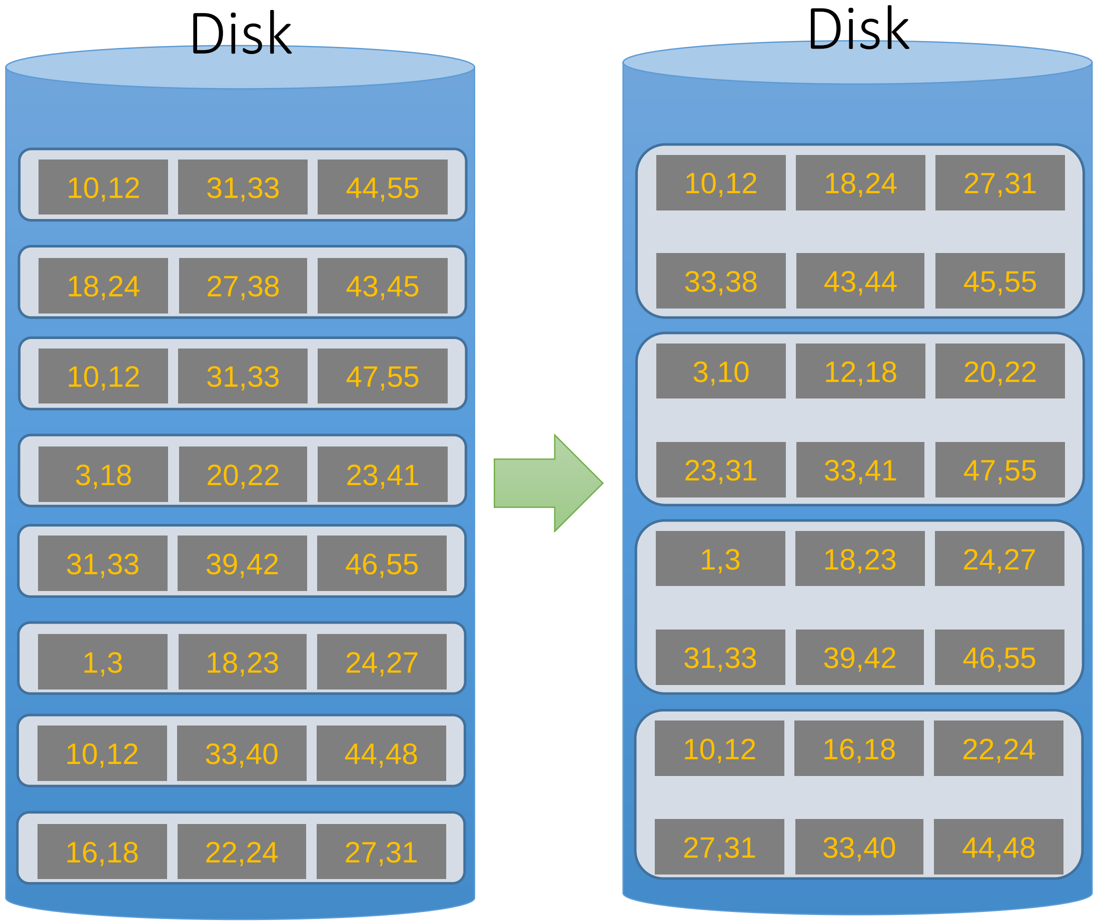
Running External Merge Sort on Larger Files

1. Split into files small enough to sort in buffer… and sort

Assume we still only have *3* buffer pages *(Buffer not pictured)*

Call each of these sorted files a ***run***

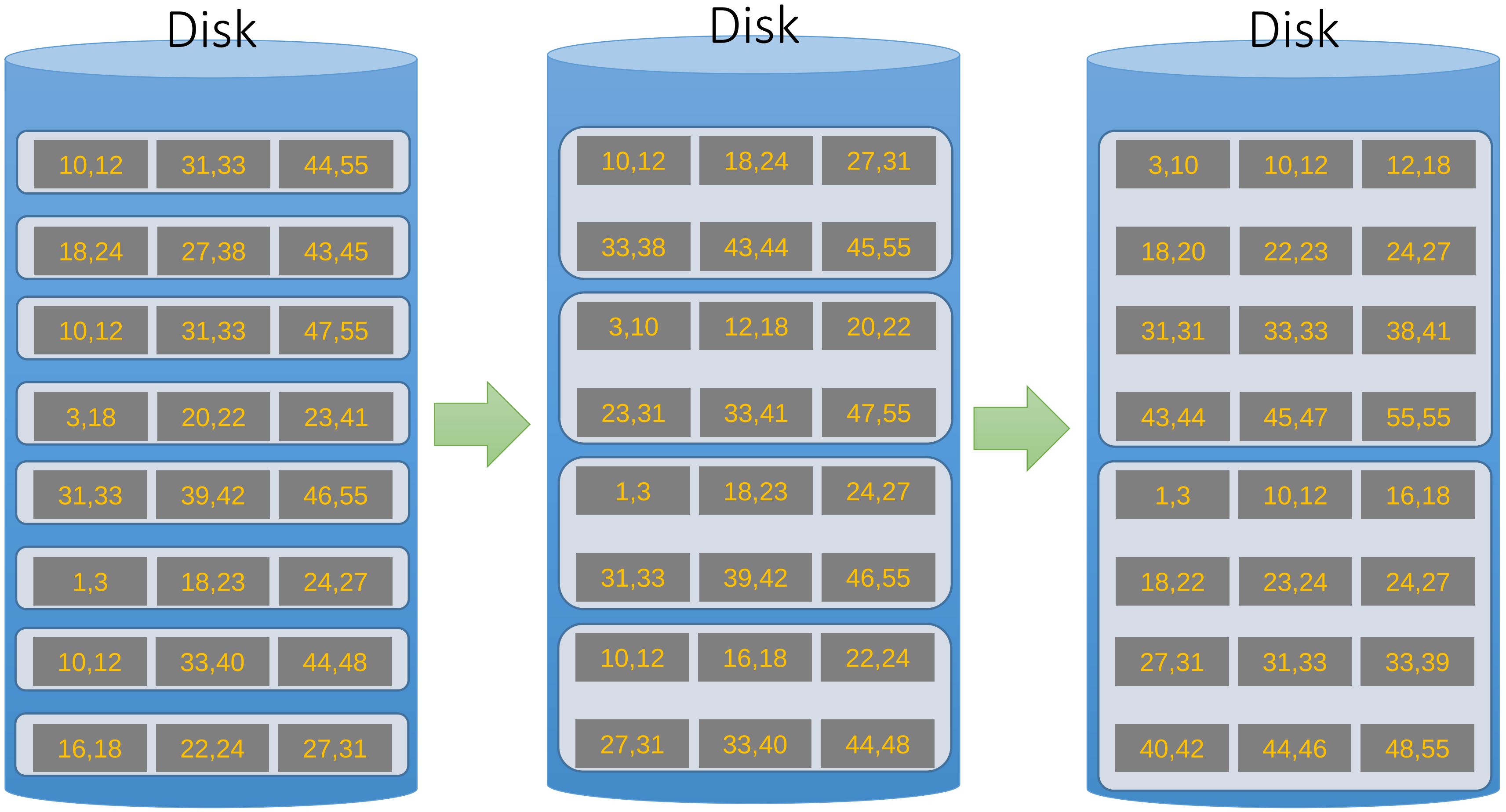
Running External Merge Sort on Larger Files



Assume we still only have *3* buffer pages *(Buffer not pictured)*

2. Now merge pairs of (sorted) files… **the resulting files will be sorted!**

Running External Merge Sort on Larger Files

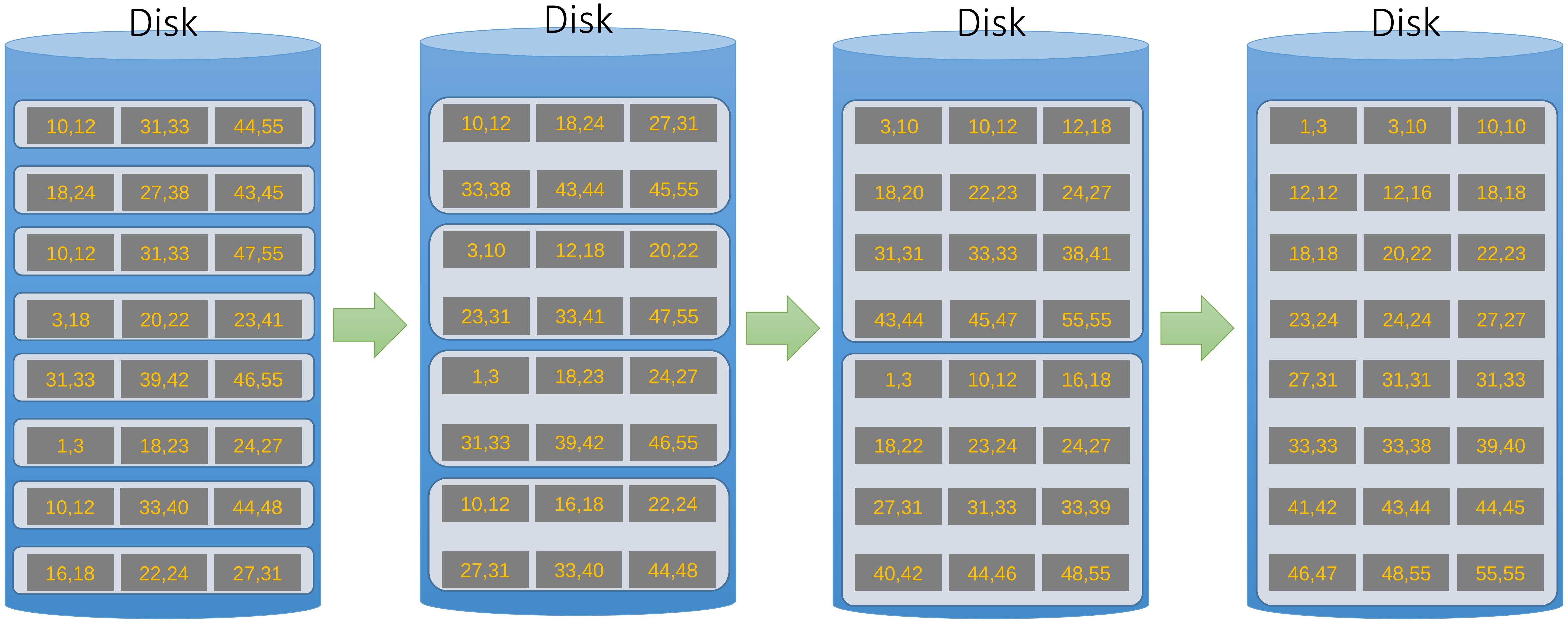


Assume we still only have *3* buffer pages *(Buffer not pictured)*

3. And repeat…

Call each of these steps a ***pass***

Running External Merge Sort on Larger Files

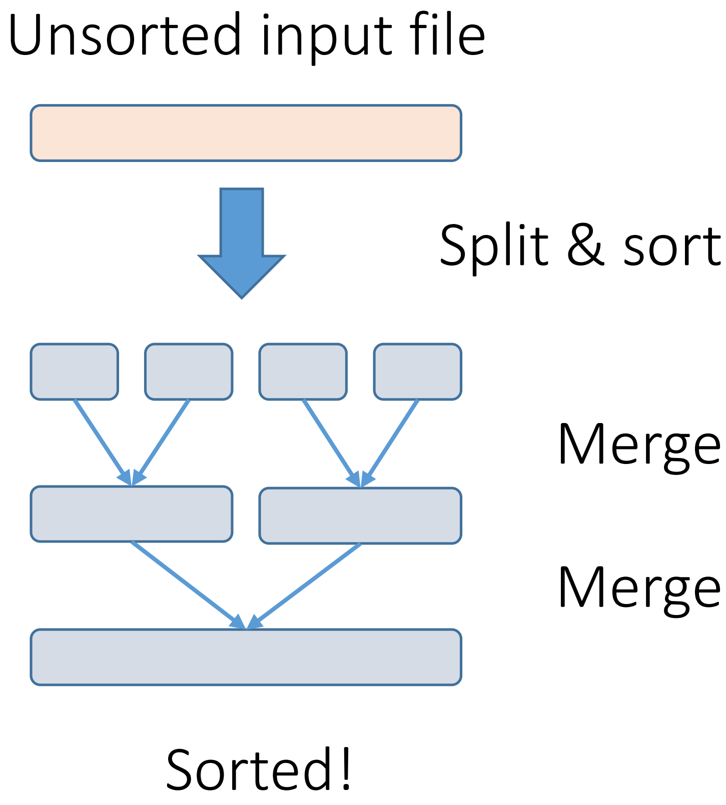


4. And repeat!

Simplified 3-page Buffer Version

Assume for simplicity that we split an N-page file into N single-page ***runs*** and sort these; then:

- First pass: Merge **N/2 *pairs* of runs** each of length **1 page**
- Second pass: Merge **N/4 *pairs* of runs** each of length **2 pages**
- In general, for **N** pages, we do $\lceil \log_2 N \rceil$ passes
  - +1 for the initial split & sort
- Each pass involves reading in & writing out all the pages = ***2N IO***



→ 2N*($\lceil \log_2 N \rceil$+1) total IO cost!