CS 6400 A

Database Systems Concepts and Design

Lecture 14

10/13/25

Announcements

Assignment 2 due next Monday (Oct 20)

- Autograder run takes time.
- Reminder: extra credit for top 3 leaderboard entries

1	Nathan Braswell	786.8451136178284
2	Tarun	1019.521054953526
3	Yihao	2309.326552046627

Project milestone assignment released

- Due Nov 3

Project proposal feedback will be released this week

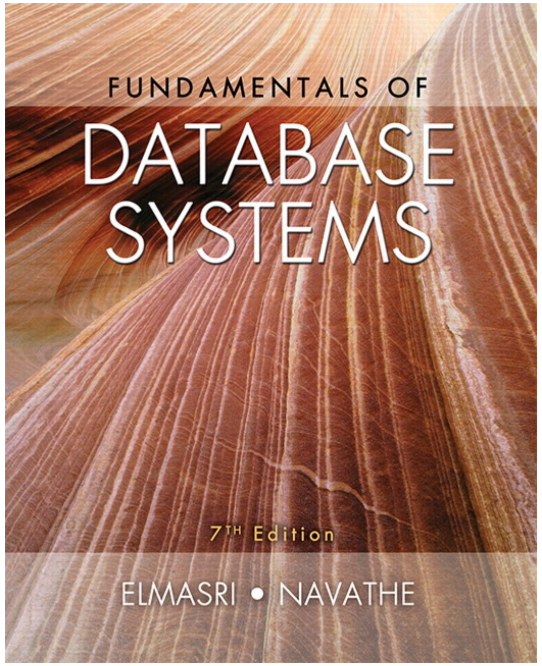
Agenda

1. The Buffer
2. External Merge Algorithm
3. External Merge Sort

Reading Materials

Fundamental of Database Systems (7th Edition)

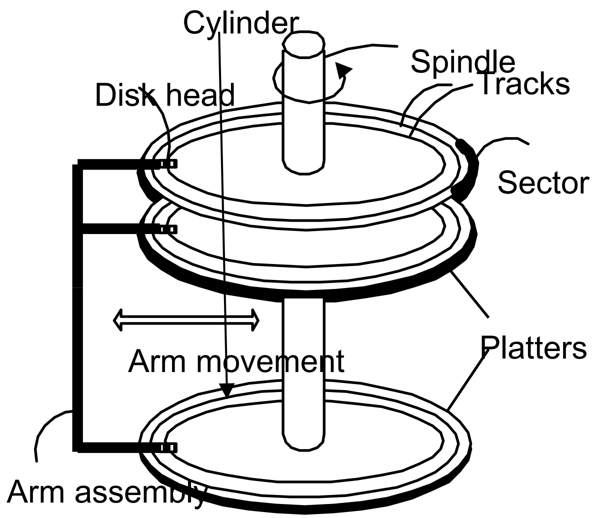
- Chapter 16.3 - Buffering of Blocks
- Chapter 18.2 - Algorithms for External Sorting



Acknowledgement: The following slides have been adapted from CS145 (Intro to Big Data Systems) taught by Peter Bailis.

1. The Buffer

Recall: Disk vs. Main Memory



Disk:

- ***Fast:*** 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:

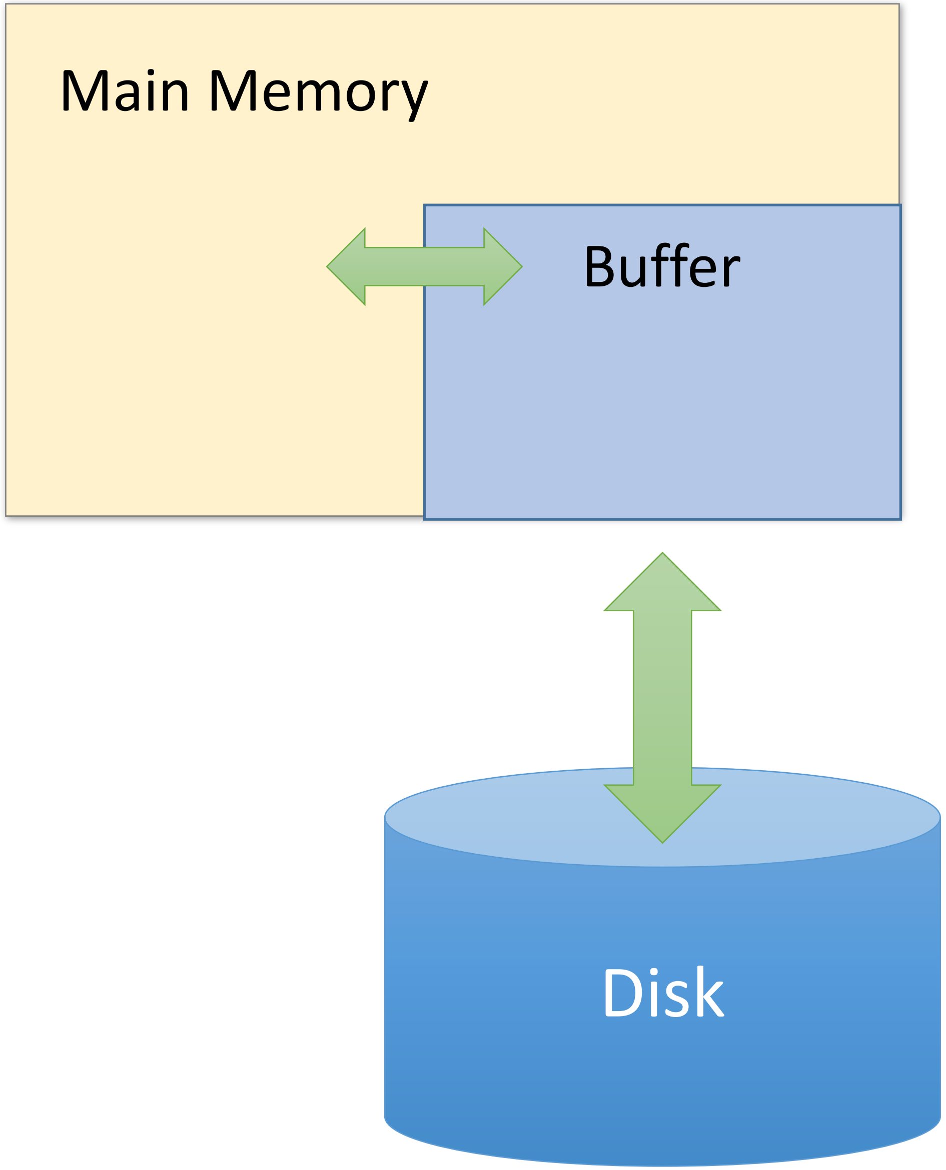
- ***Fast:*** Random access, byte addressable
    - ~10x faster for sequential access
    - ~100,000x faster for random access!
- ***Volatile:*** Data can be lost if e.g. crash occurs, power goes out, etc!
- ***Expensive:*** For $100, get 16GB of RAM vs. 2TB of disk!

The Buffer

A <u>buffer</u> 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**

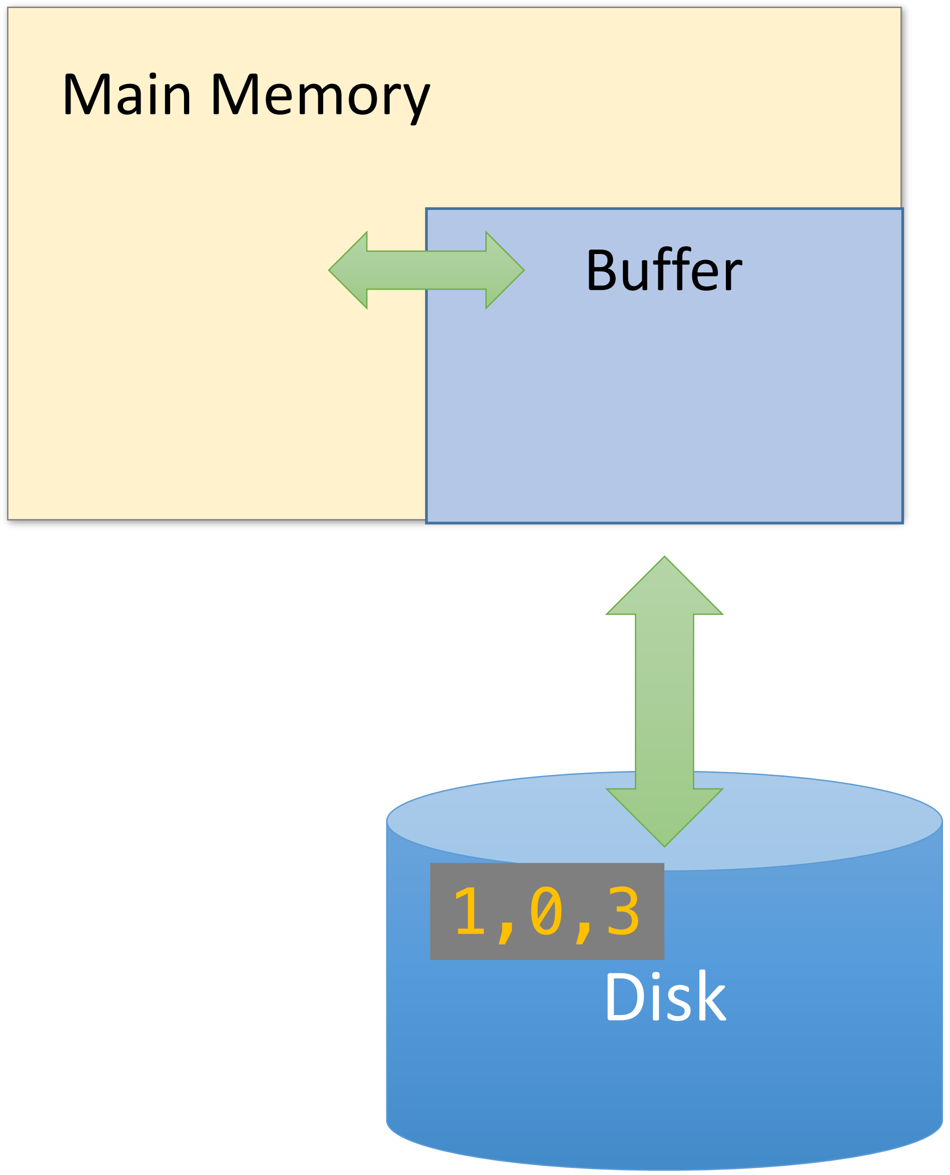
Key idea: Reading / writing to disk is slow- need to cache data!



The (Simplified) Buffer

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

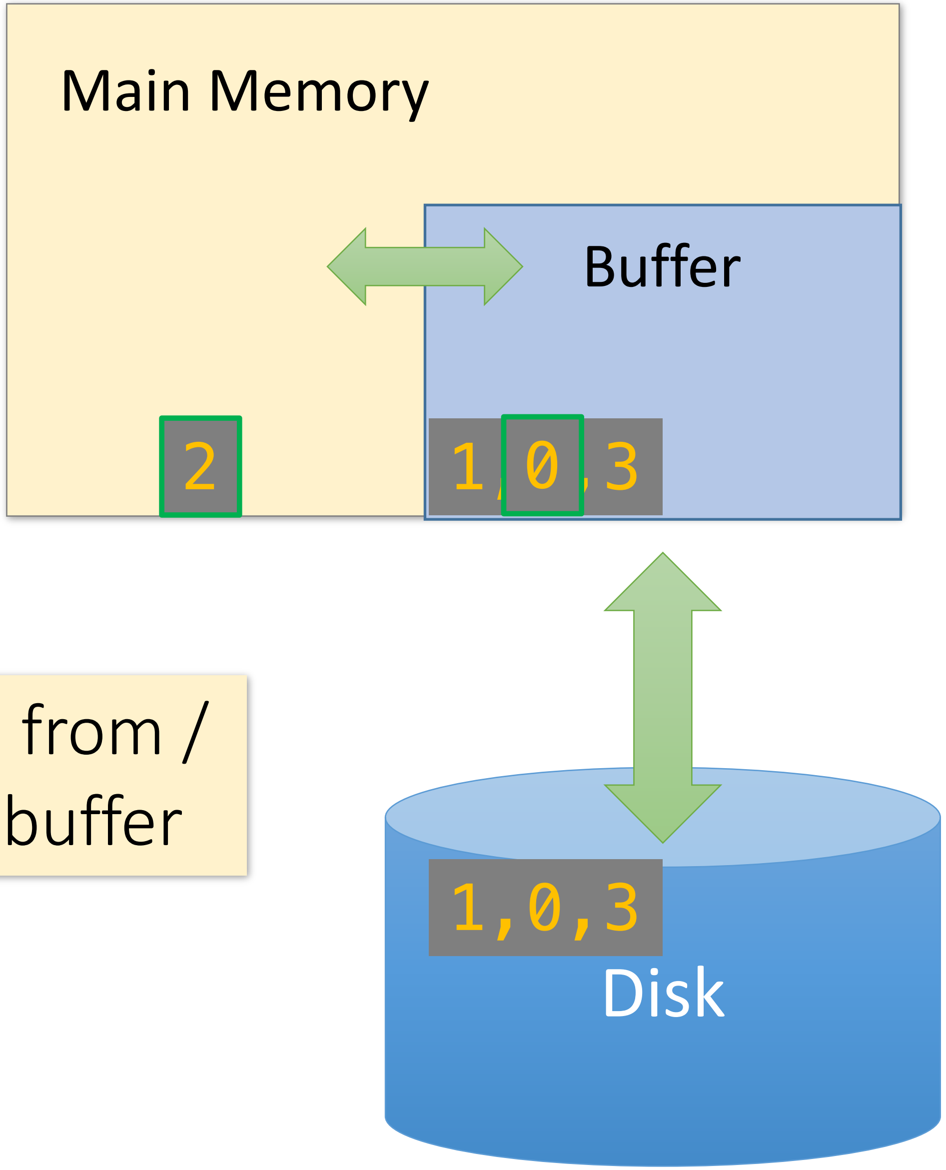
- Read(page): Read page from disk -> buffer if not already in buffer



The (Simplified) Buffer

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

- Read(page): Read page from disk -> buffer if not already in buffer

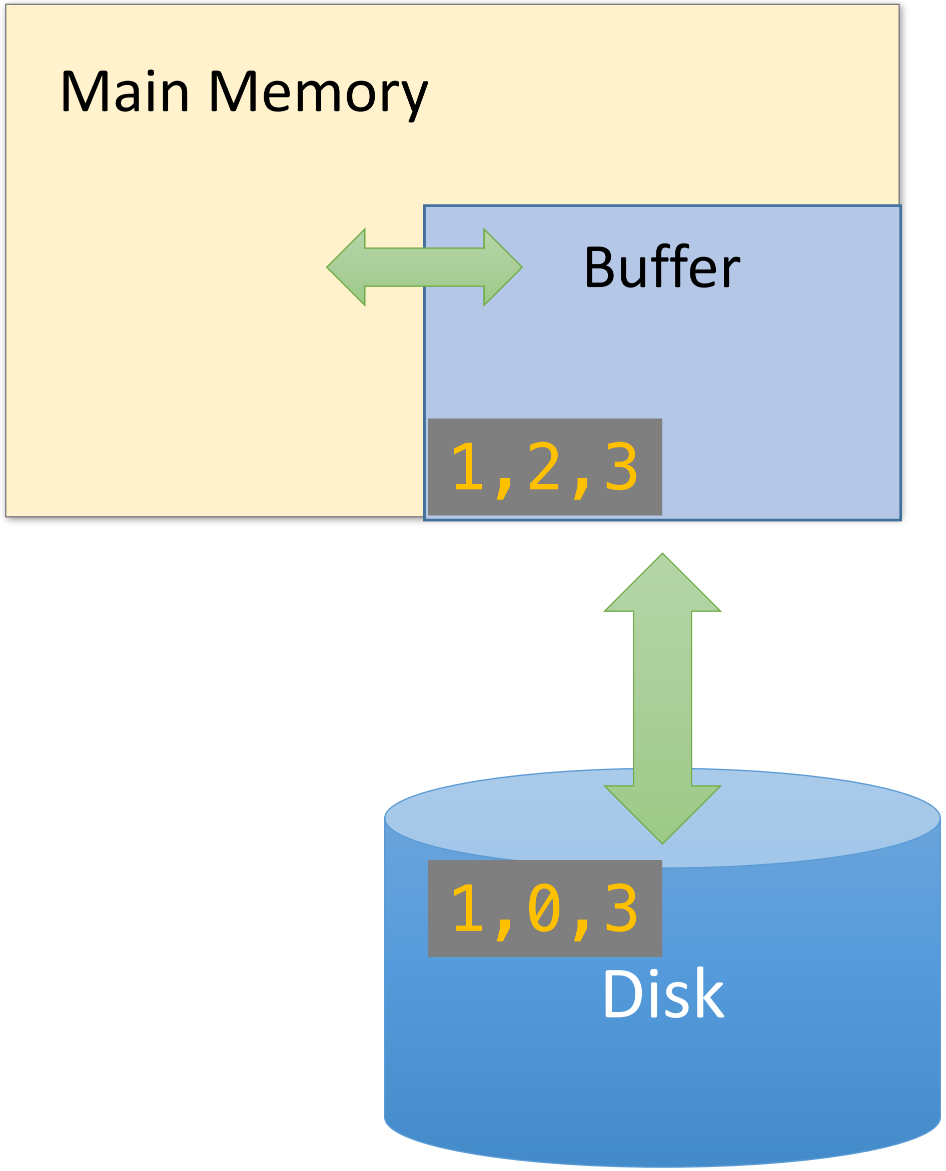


Processes can then read from / write to the page in the buffer

The (Simplified) Buffer

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

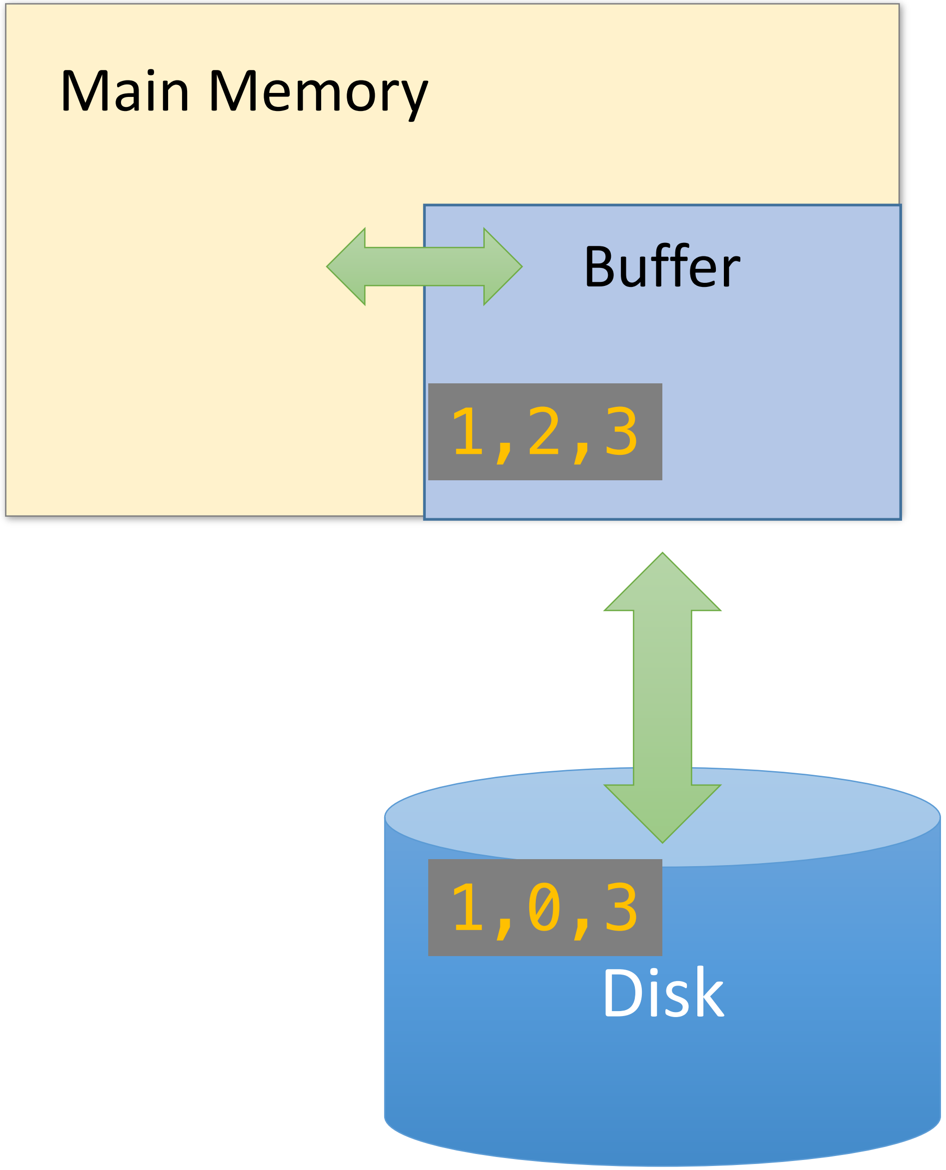
- Read(page): Read page from disk -> buffer if not already in buffer
- Flush(page): Evict page from buffer & write to disk



The (Simplified) Buffer

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

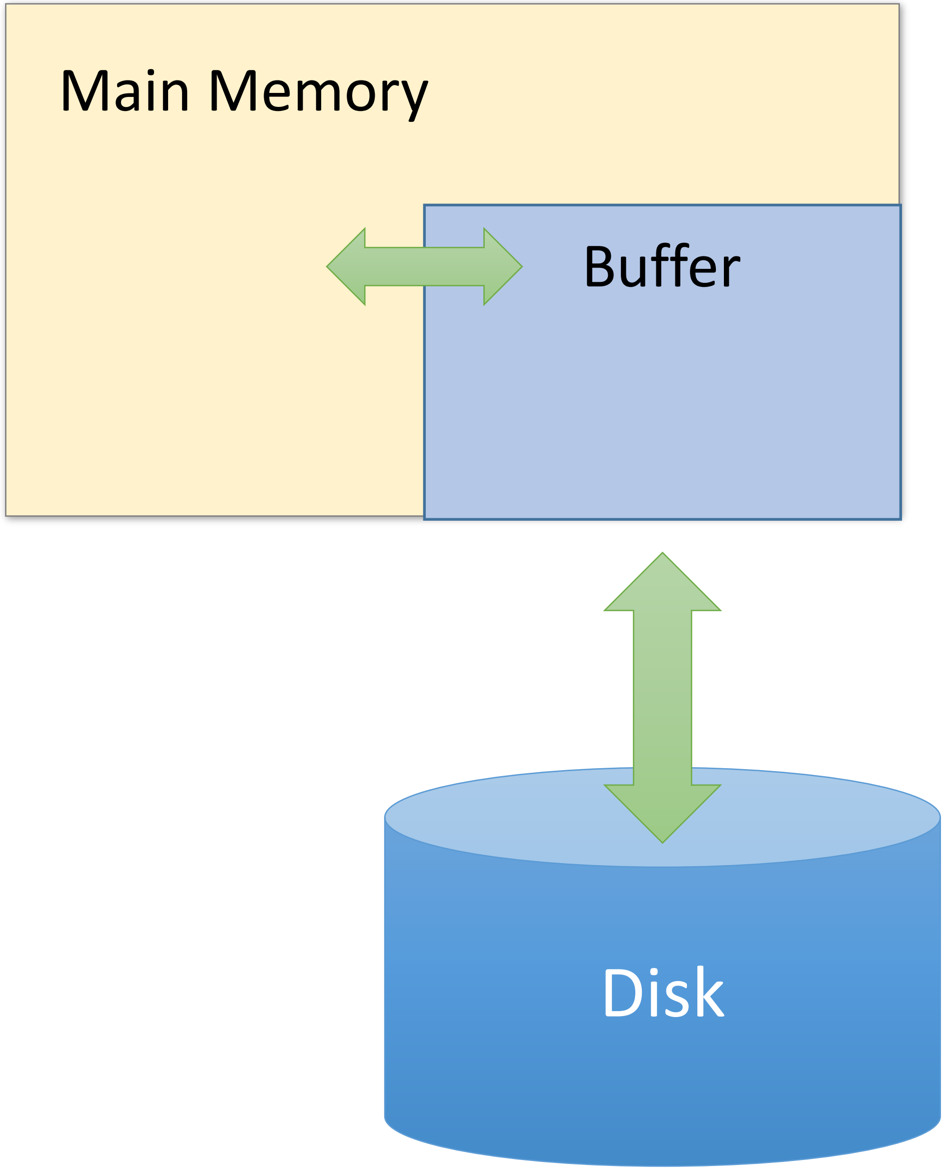
- Read(page): Read page from disk -> buffer if not already in buffer
- Flush(page): Evict page from buffer & write to disk
- Release(page): Evict page from buffer without writing to disk



The DBMS Buffer

Database maintains its own buffer

- Why? The OS already does this…
- DB knows more about access patterns.
- Recovery and logging require ability to **flush** to disk.



The Buffer Manager

A <u>buffer manager</u> 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
  - Examples: LRU, FIFO, Clock
- DBMSs typically implement their own buffer management routines

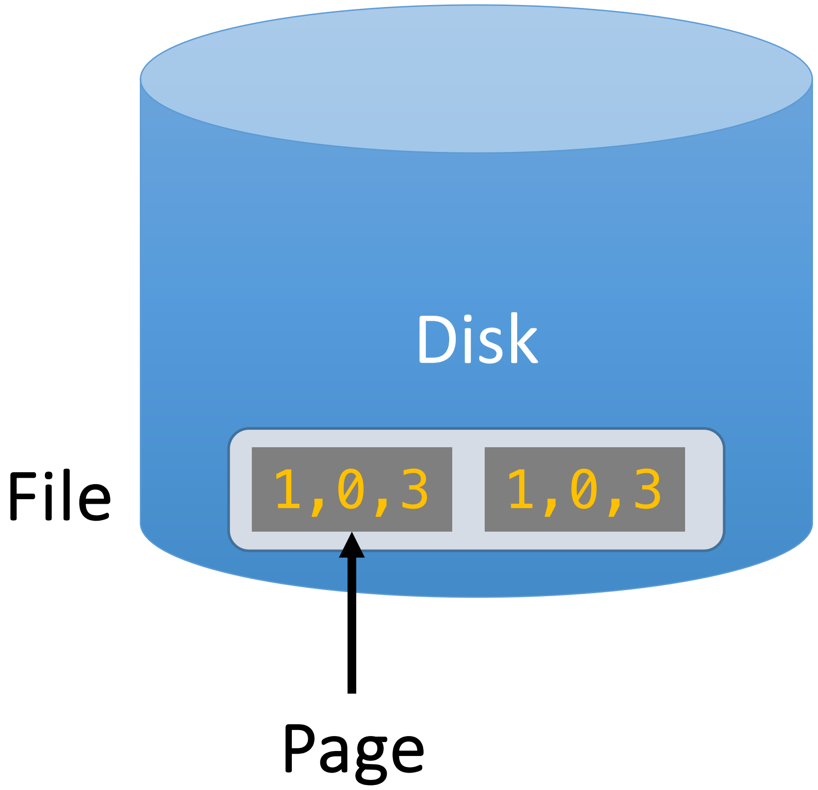
A Simplified Filesystem Model

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

And a <u>file</u> is a *variable-length list* of pages

- Interface: create / open / close; next_page(); etc.



2. External Merge Algorithm

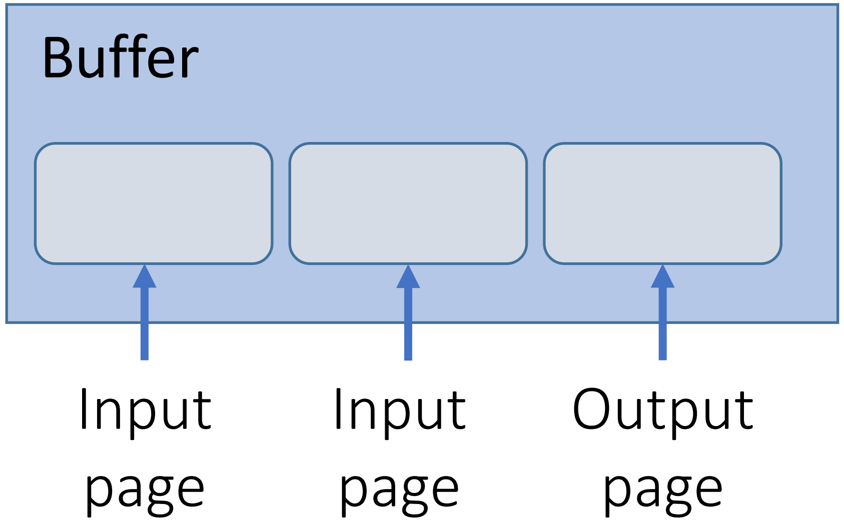
Challenge: Merging Big Files with Small Memory

- 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

Example of an **"IO aware"** algorithm / cost model

External Merge Algorithm: Summary

- **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:

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

Then:

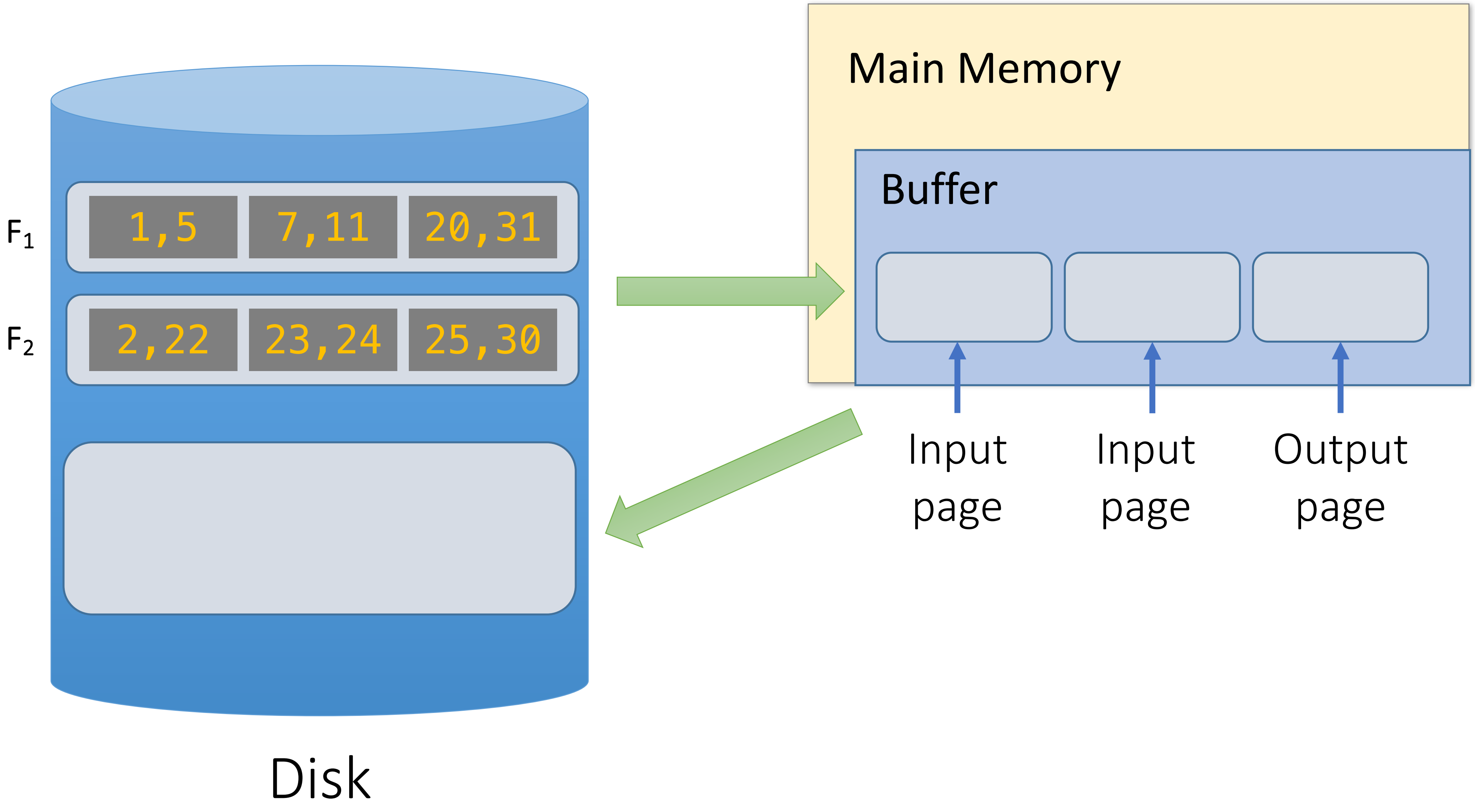
$$Min(A_1, B_1) \le A_i$$
$$Min(A_1, B_1) \le B_j$$

for i=1….N and j=1….M

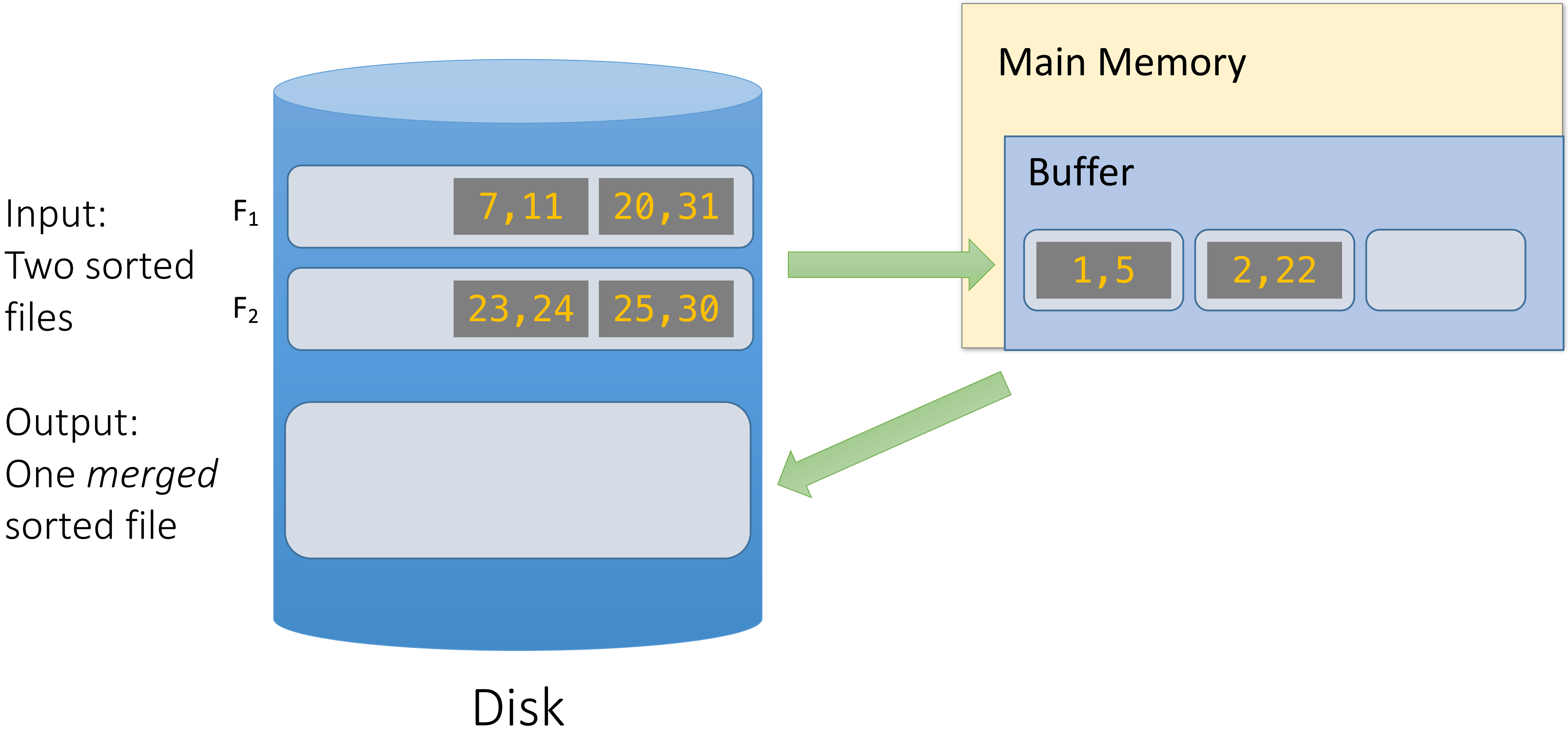
External Merge Algorithm

Input:
Two sorted files

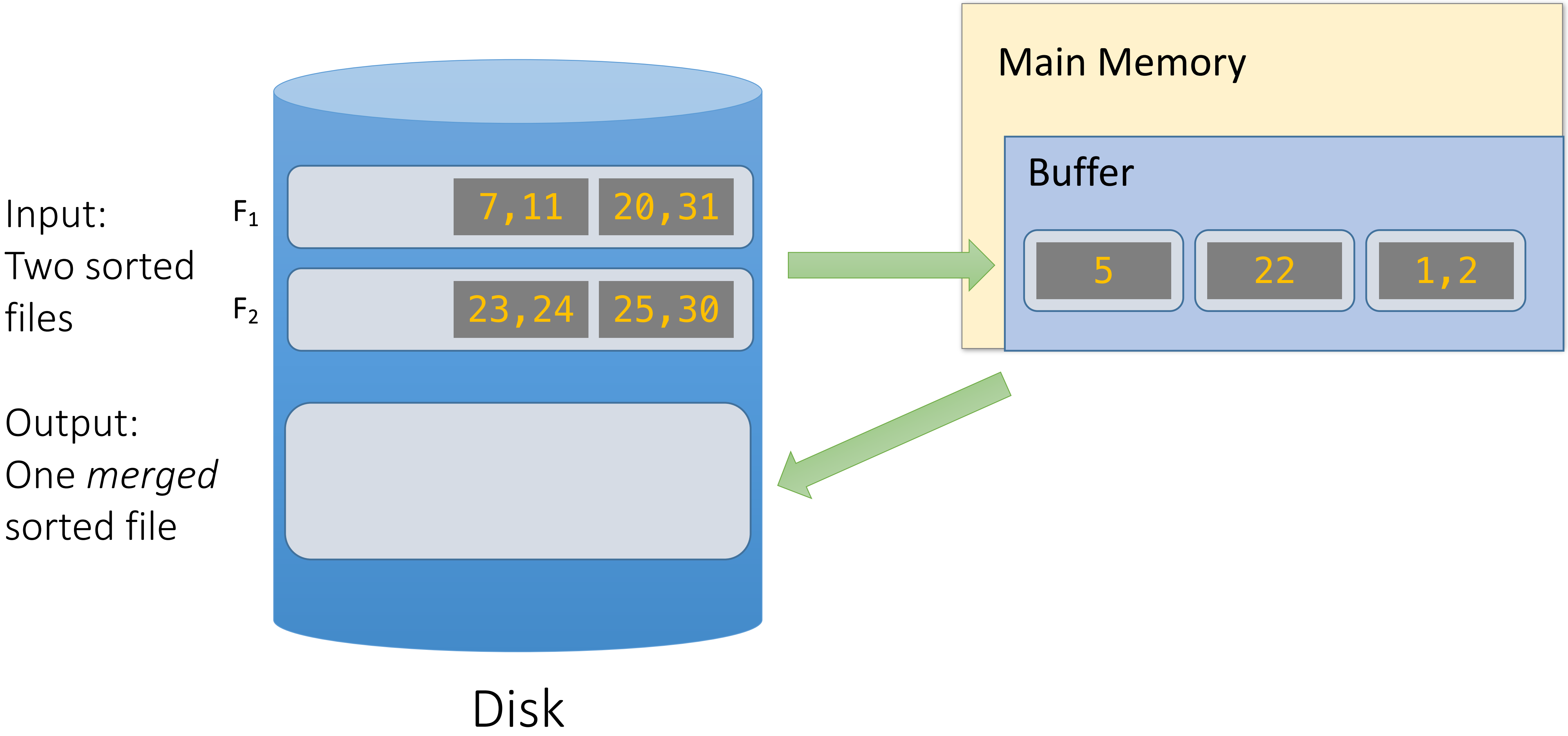
Output:
One *merged* sorted file



External Merge Algorithm



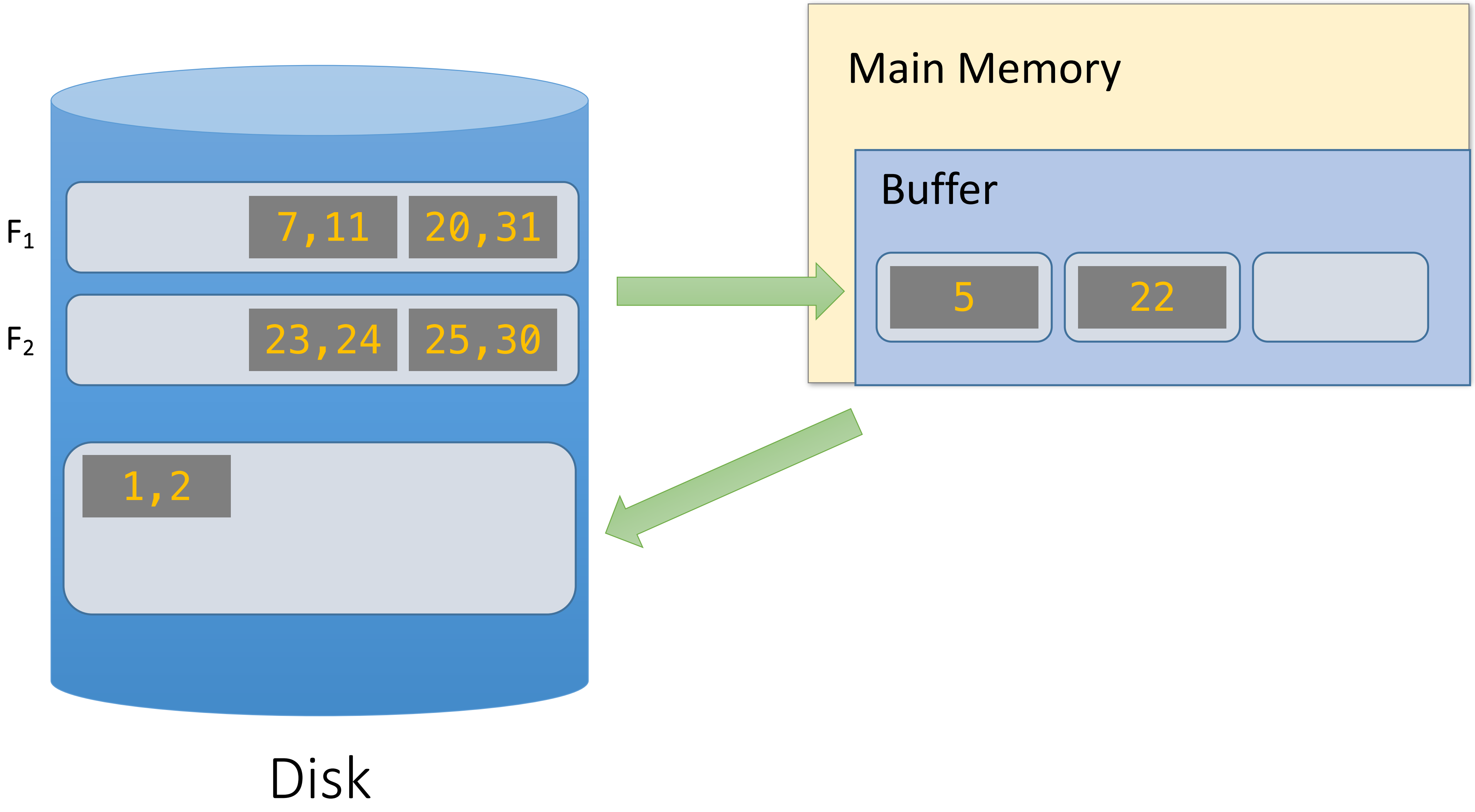
External Merge Algorithm



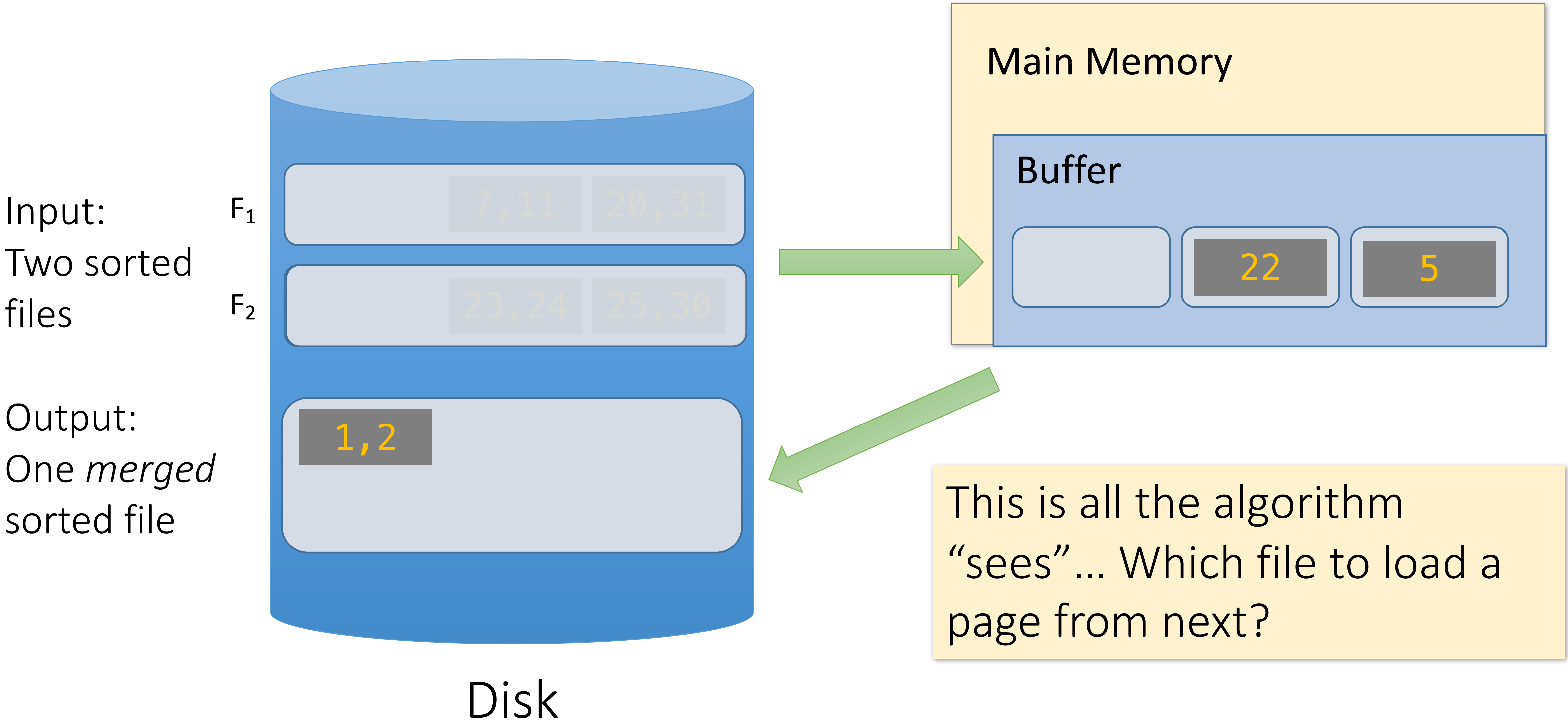
External Merge Algorithm

Input:
Two sorted
files

Output:
One *merged*
sorted file



External Merge Algorithm

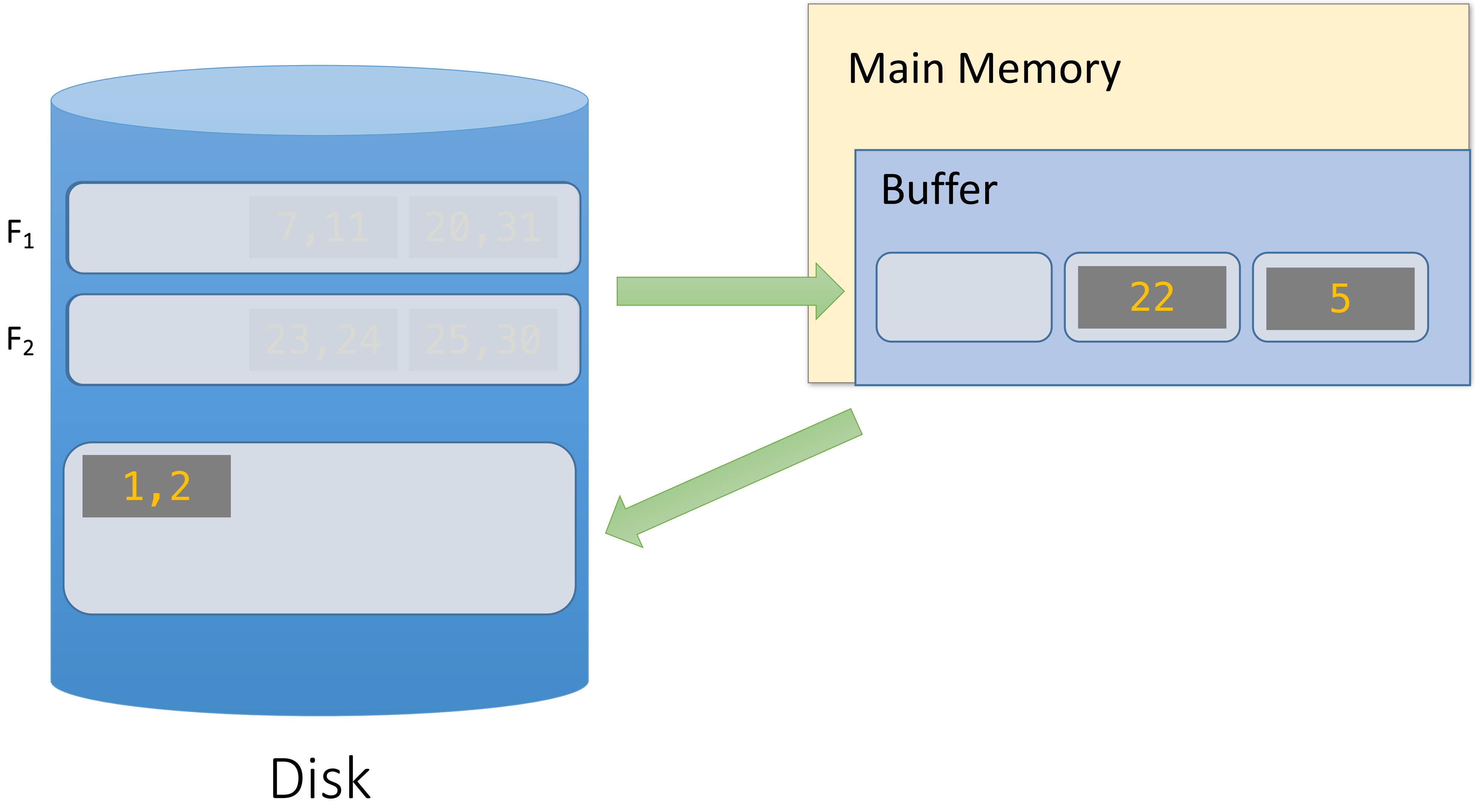


This is all the algorithm "sees"… Which file to load a page from next?

External Merge Algorithm

Input:
Two sorted files

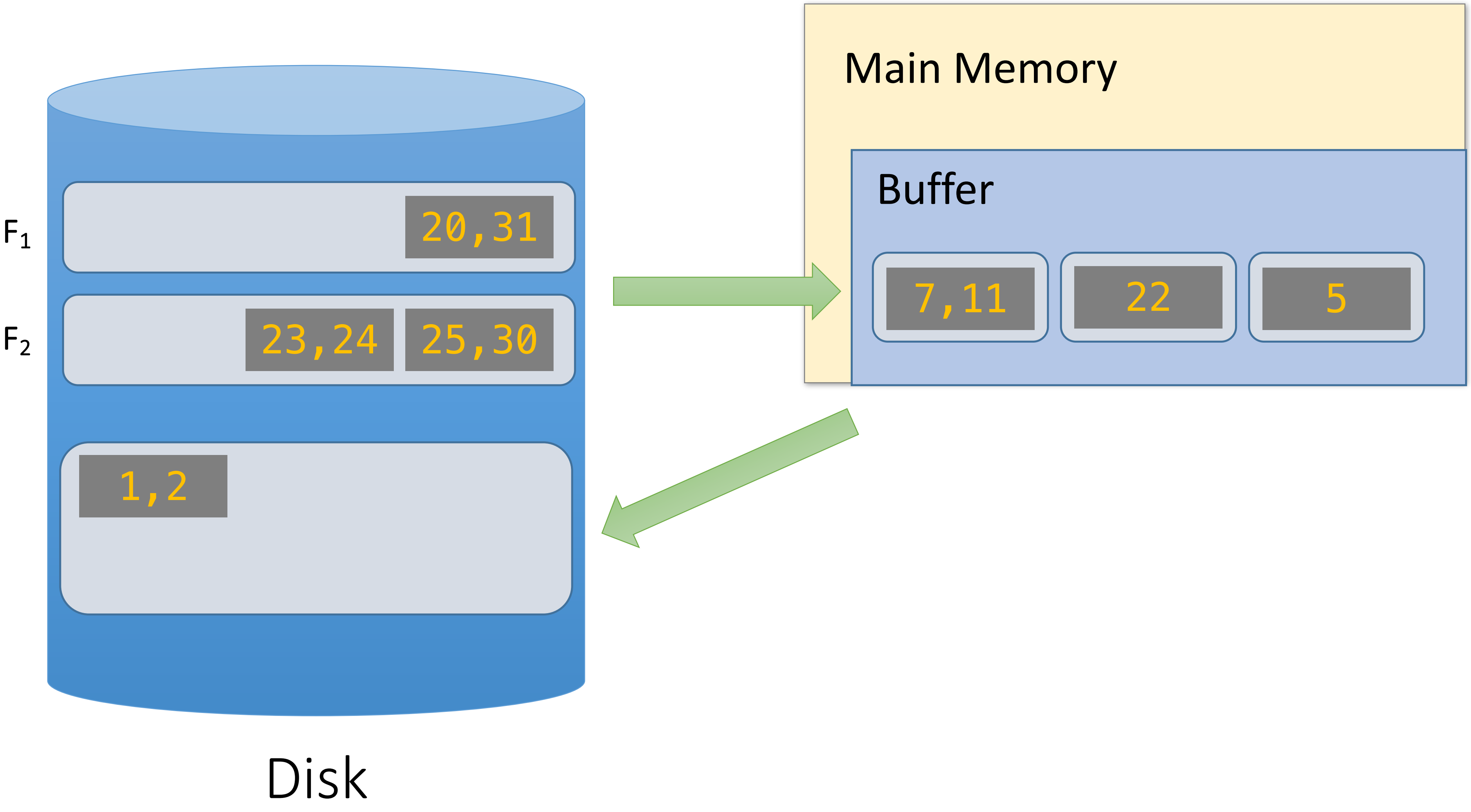
Output:
One *merged* sorted file



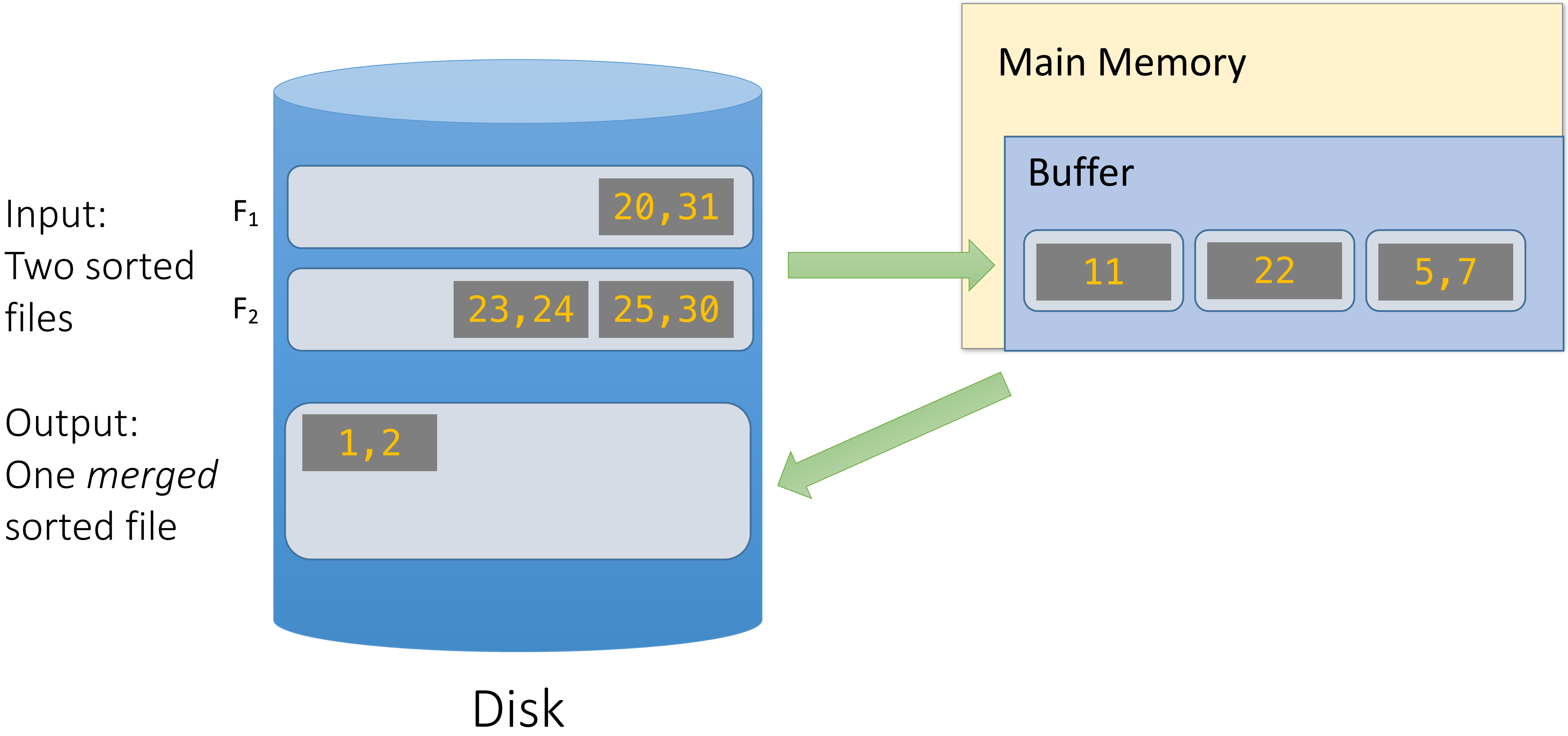
External Merge Algorithm

Input:
Two sorted
files

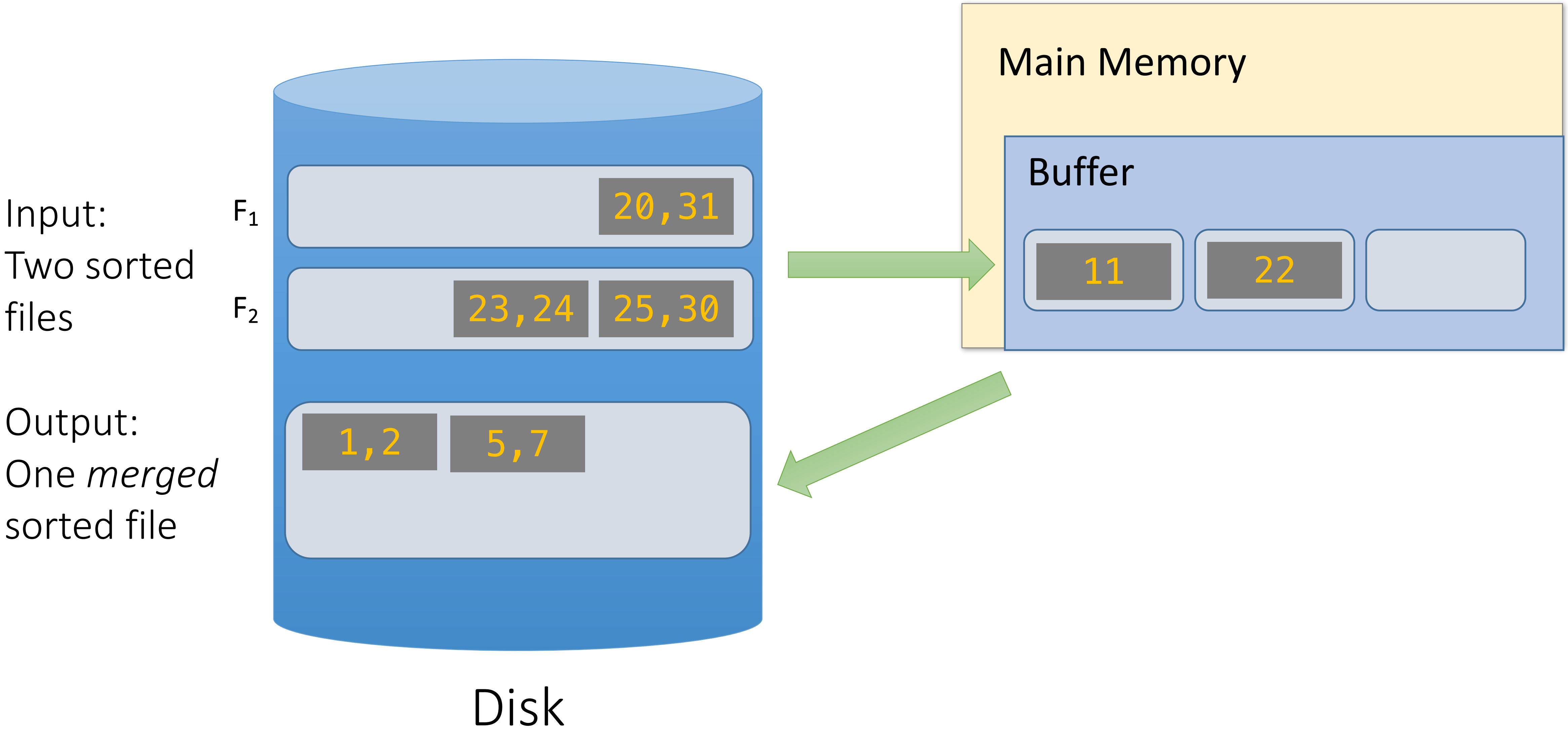
Output:
One *merged*
sorted file



External Merge Algorithm



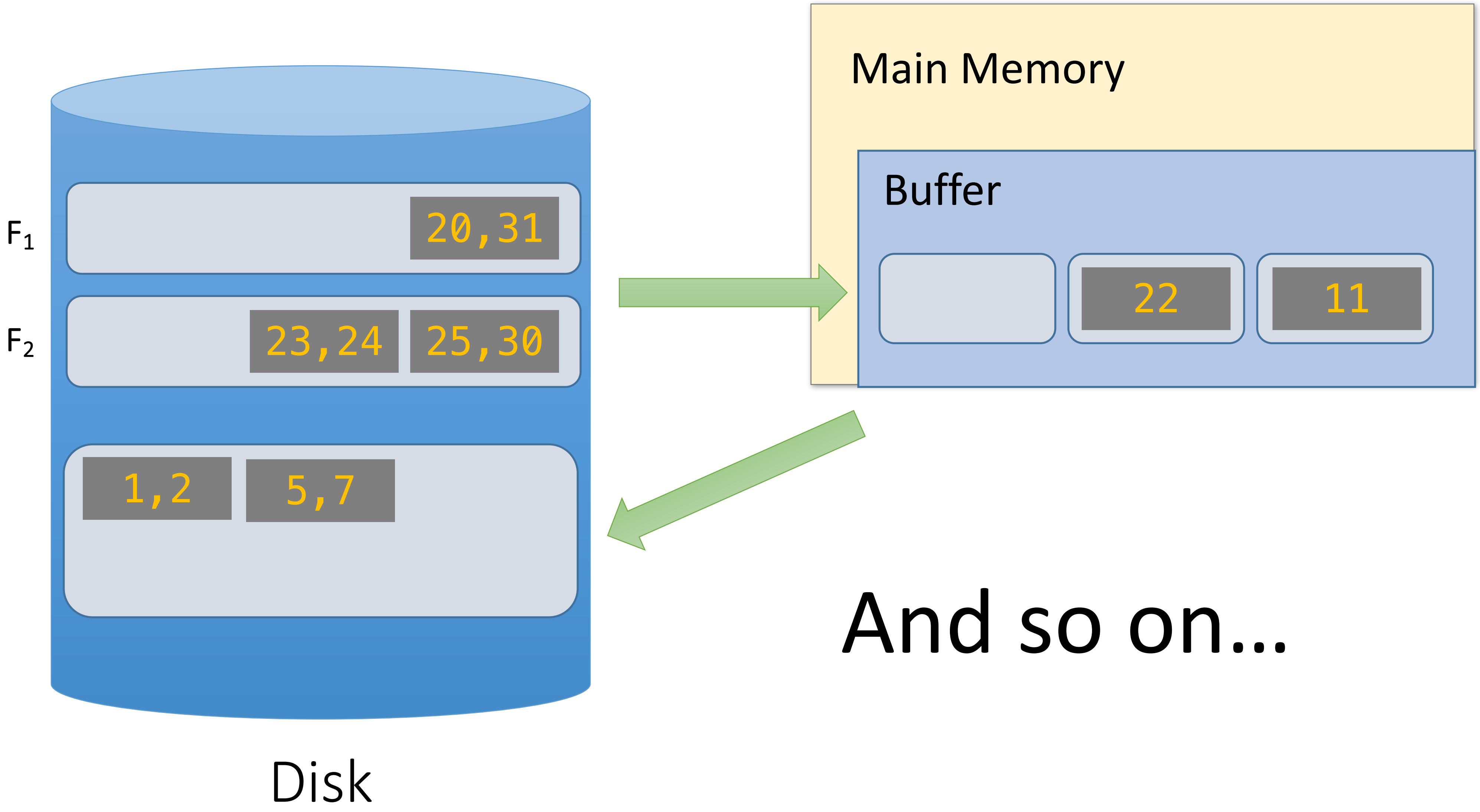
External Merge Algorithm



External Merge Algorithm

Input:
Two sorted files

Output:
One *merged* sorted file



And so on…

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?

3. External Merge Sort

Why are Sort Algorithms Important?

- Data requested from DB in sorted order is **extremely common**
  - e.g., find customer orders in increasing total amounts

- **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

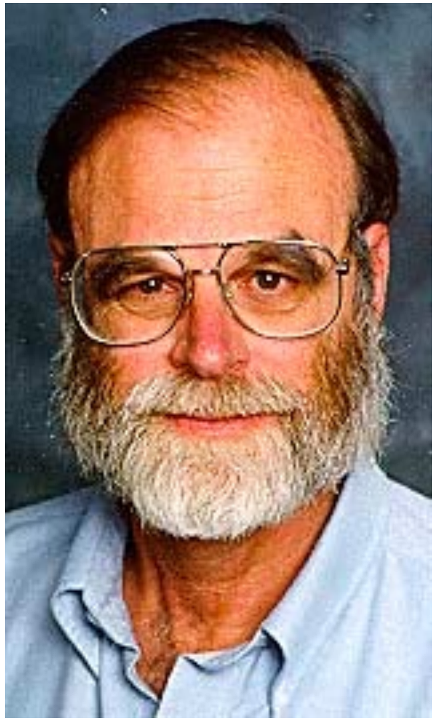
- Sorting is first step in *bulk loading* B+ tree index.

- *Sort-merge* join algorithm involves sorting

Next lecture

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”)*

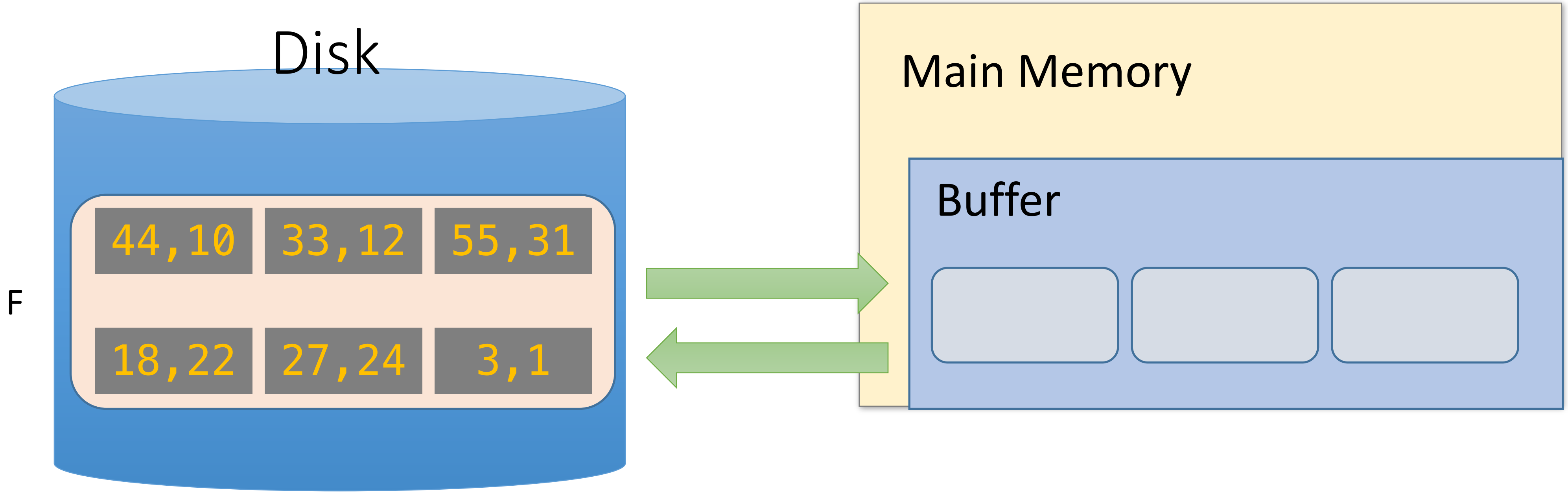
2. **Merge** pairs (or groups) of runs *using the external merge algorithm*

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

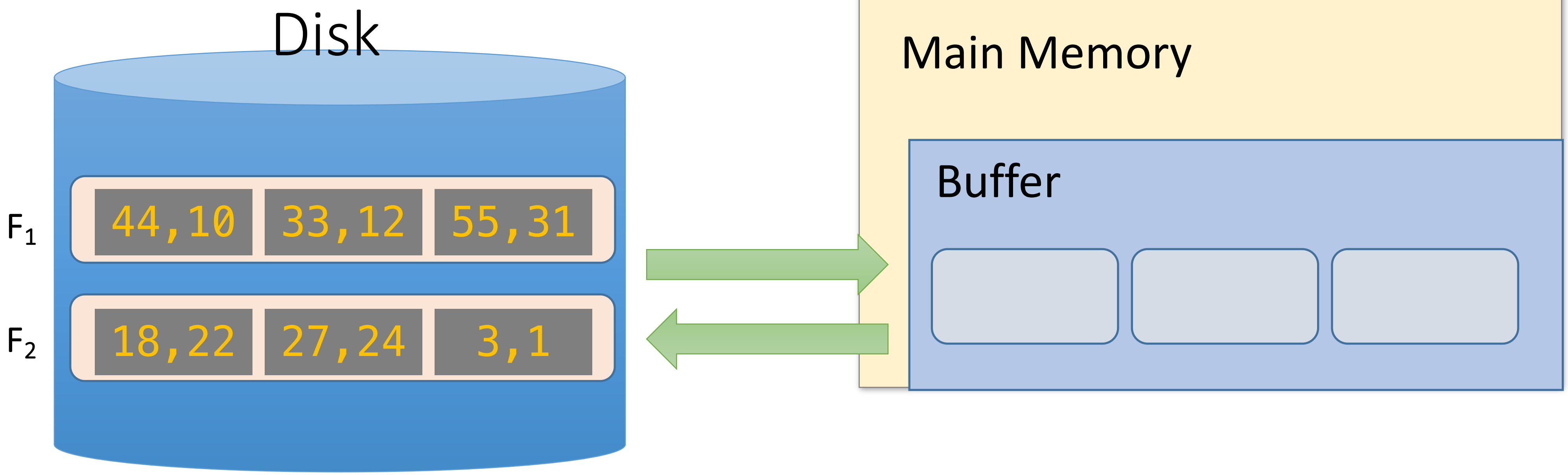


1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm

Example:
- 3 Buffer pages
- 6-page file

Orange file = unsorted

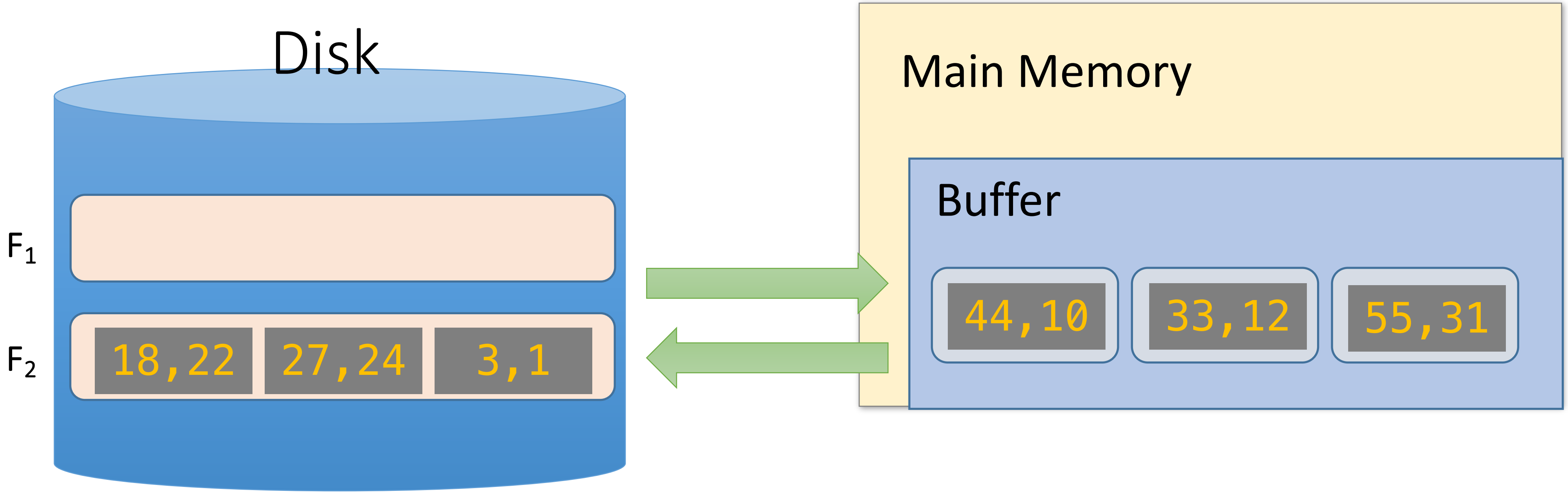


1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm

Example:
- 3 Buffer pages
- 6-page file

Orange file = unsorted

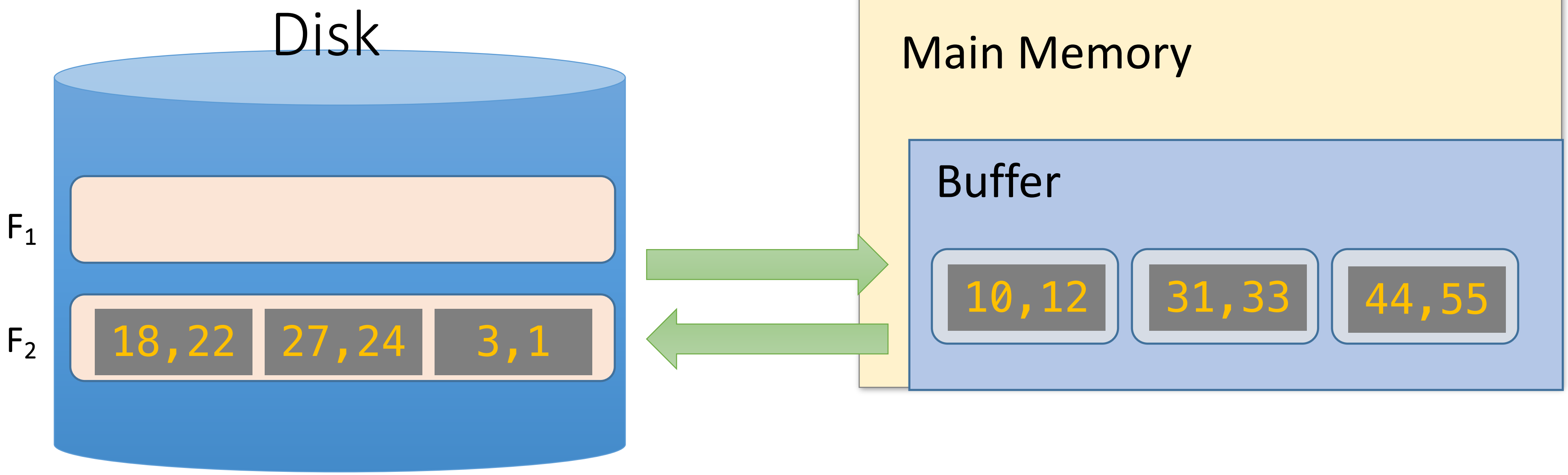


1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm

Example:
- 3 Buffer pages
- 6-page file

Orange file = unsorted



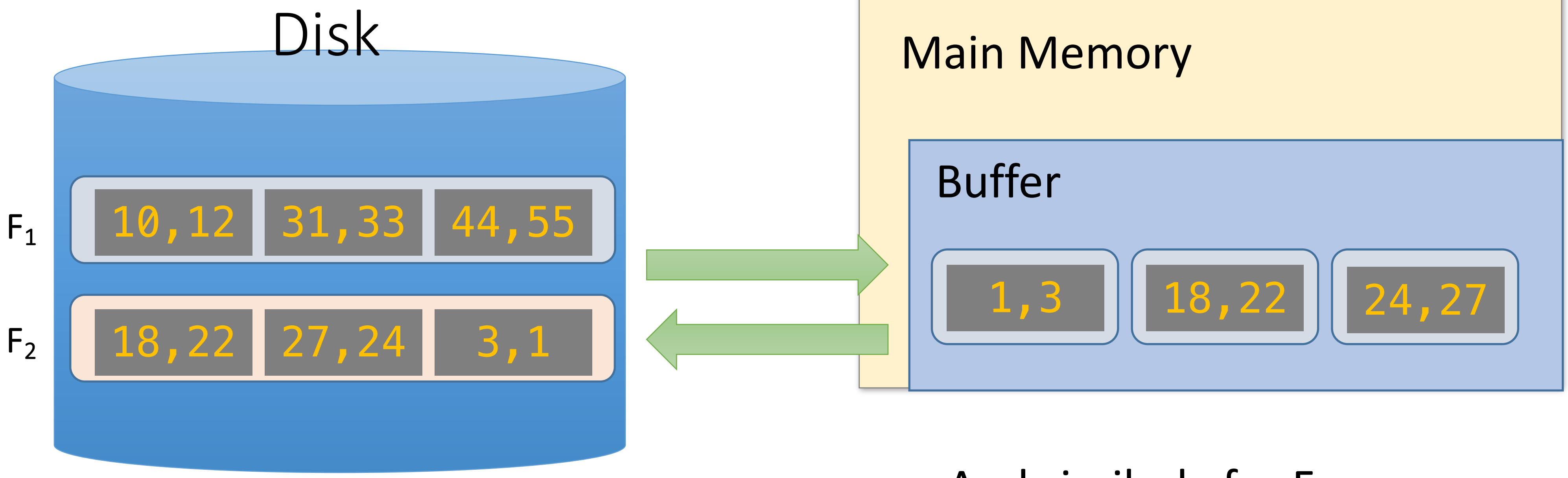
1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm

Example:

- 3 Buffer pages
- 6-page file

Each sorted file is a called a *run*



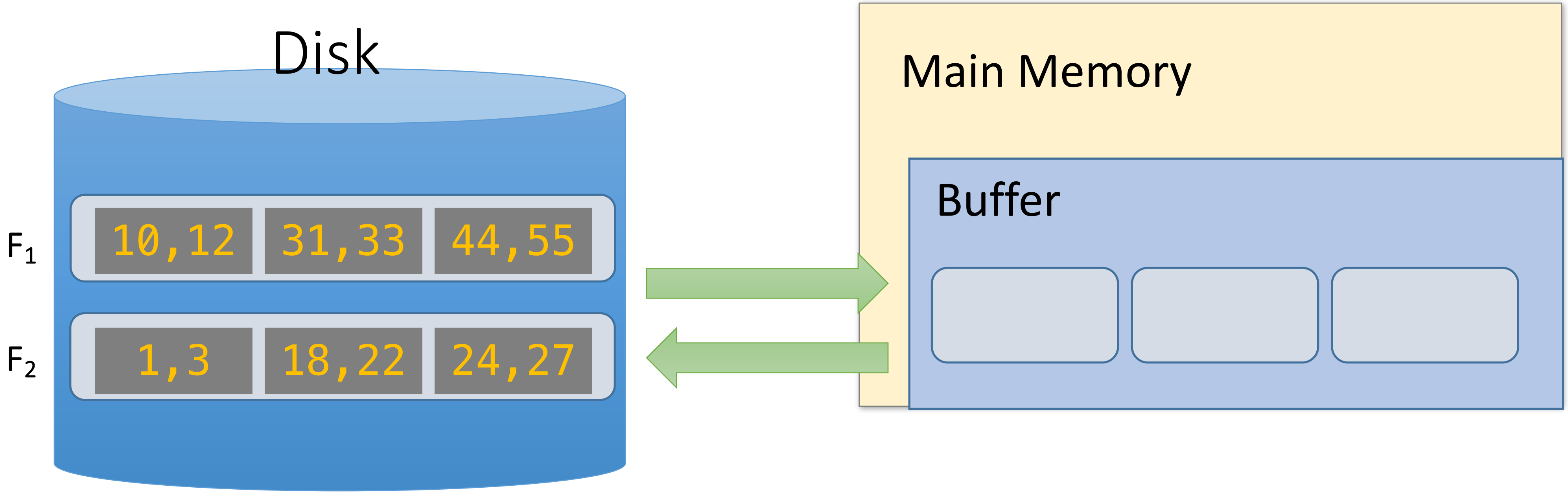
And similarly for F_2

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

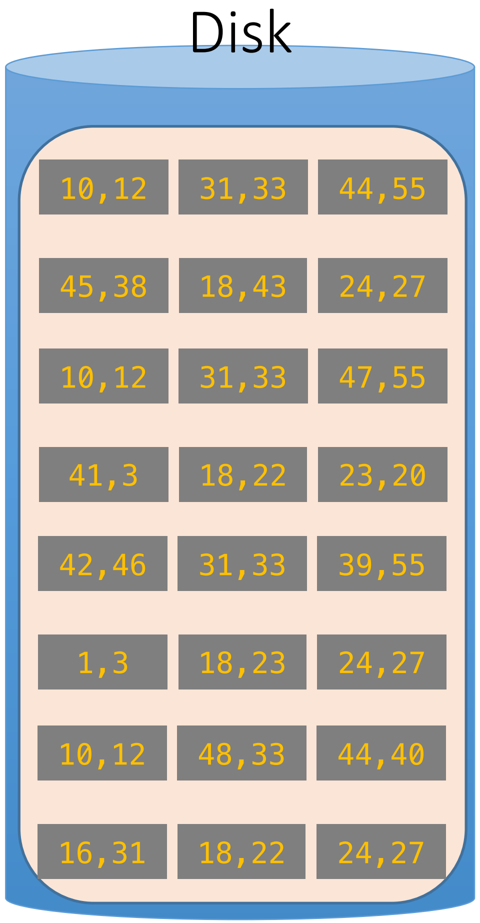
For 3 buffer pages, 6 page file:

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

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

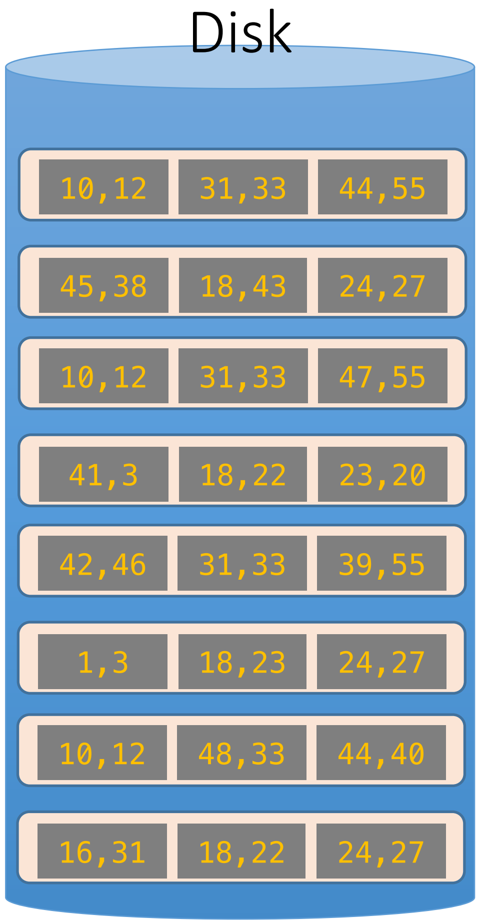
3. Total cost = 24 IO

Running External Merge Sort on Larger Files



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

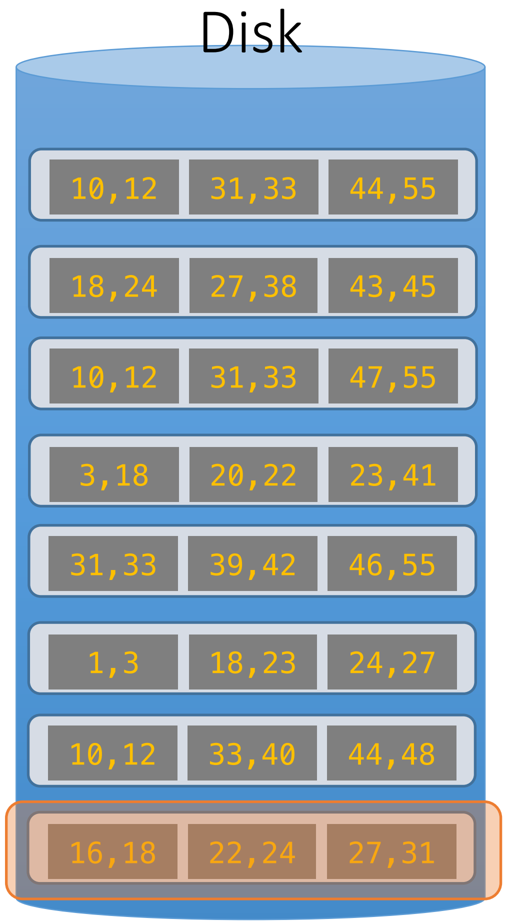
Running External Merge Sort on Larger Files



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

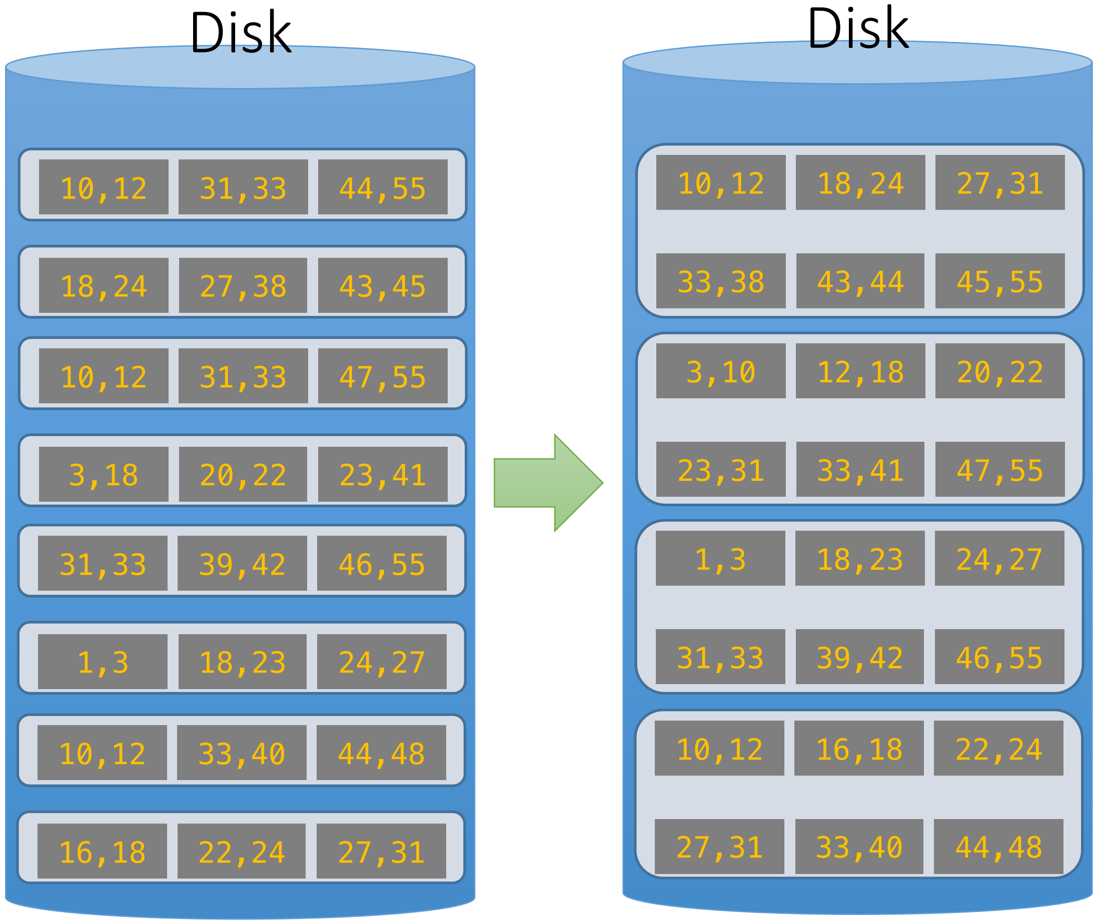


1. Split into files small enough to sort in buffer…

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

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

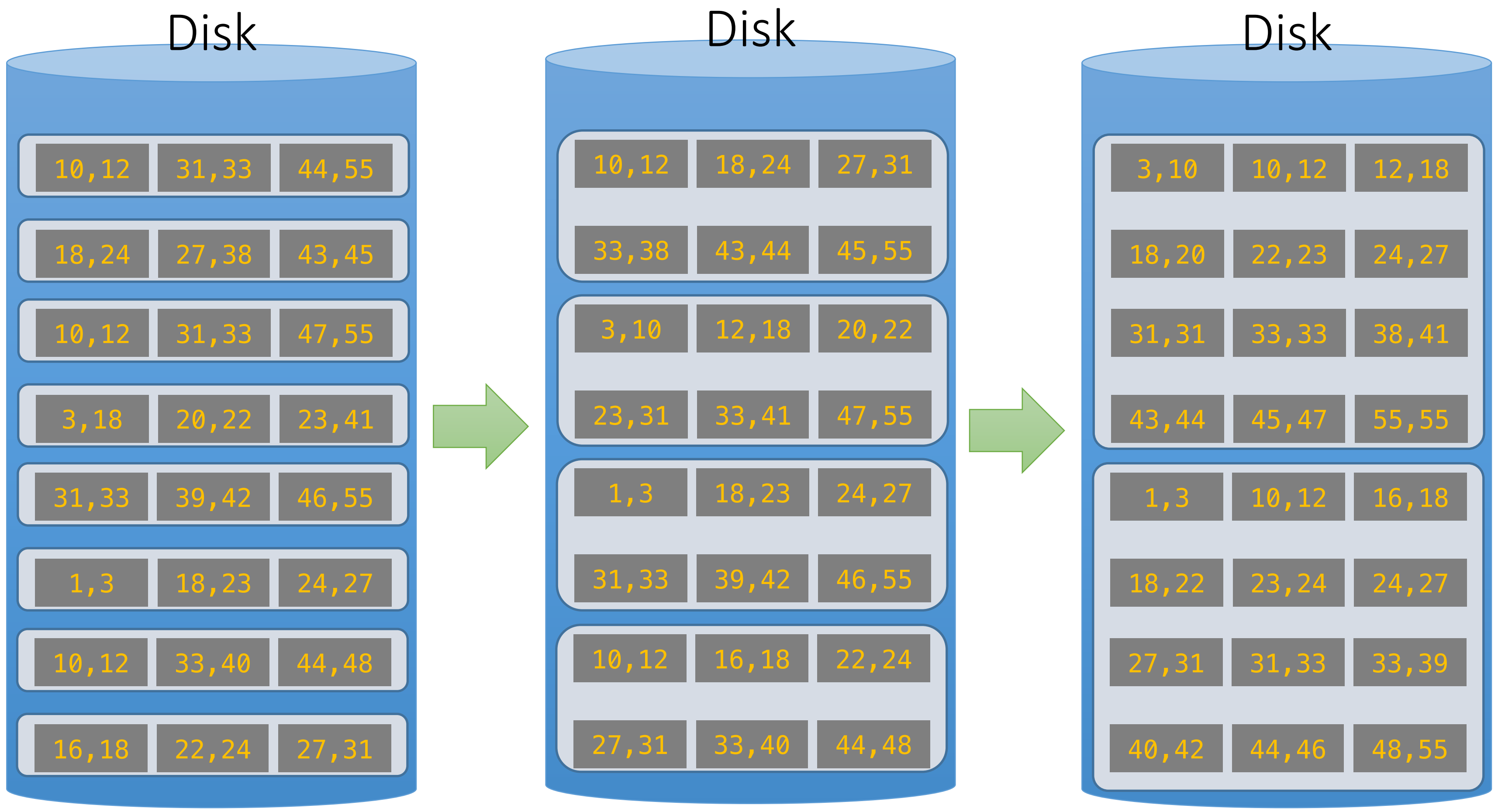
Running External Merge Sort on Larger Files



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

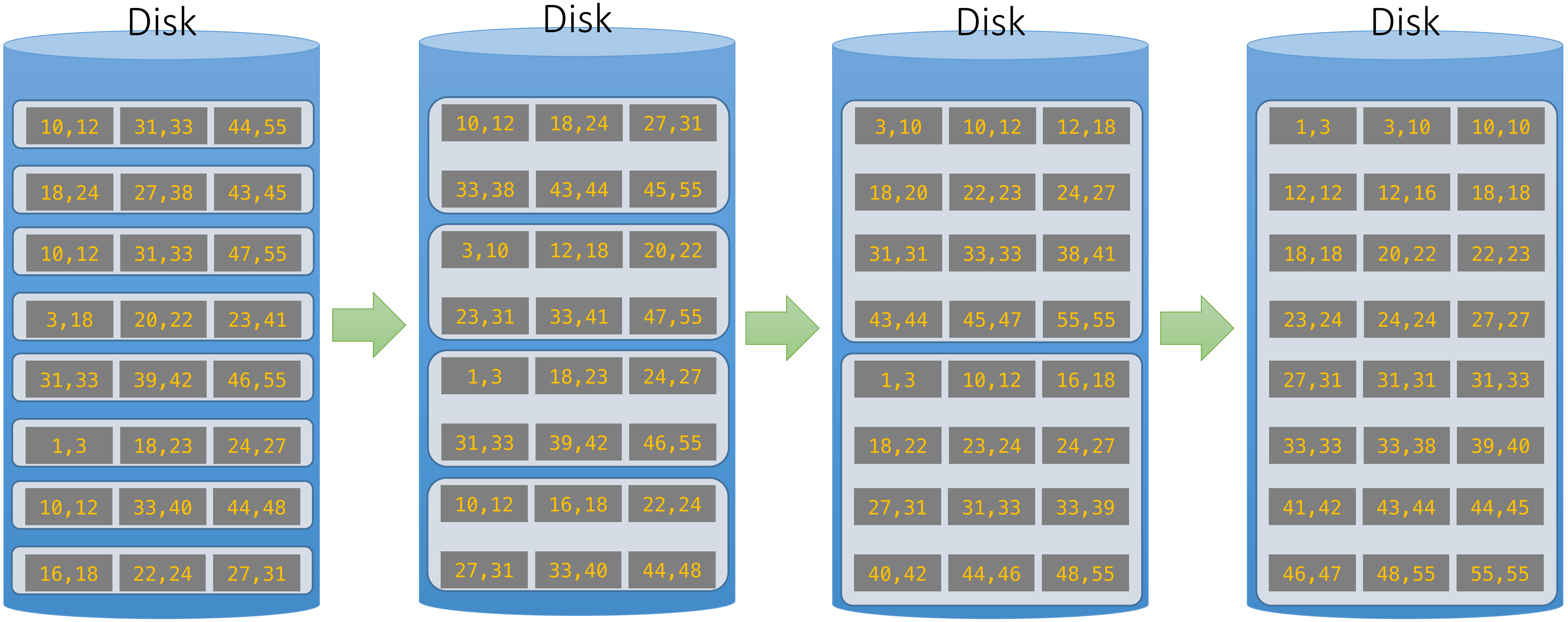


Assume we still only have *3* buffer pages *(Buffer not pictured)*

3. And repeat…

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

Running External Merge Sort on Larger Files

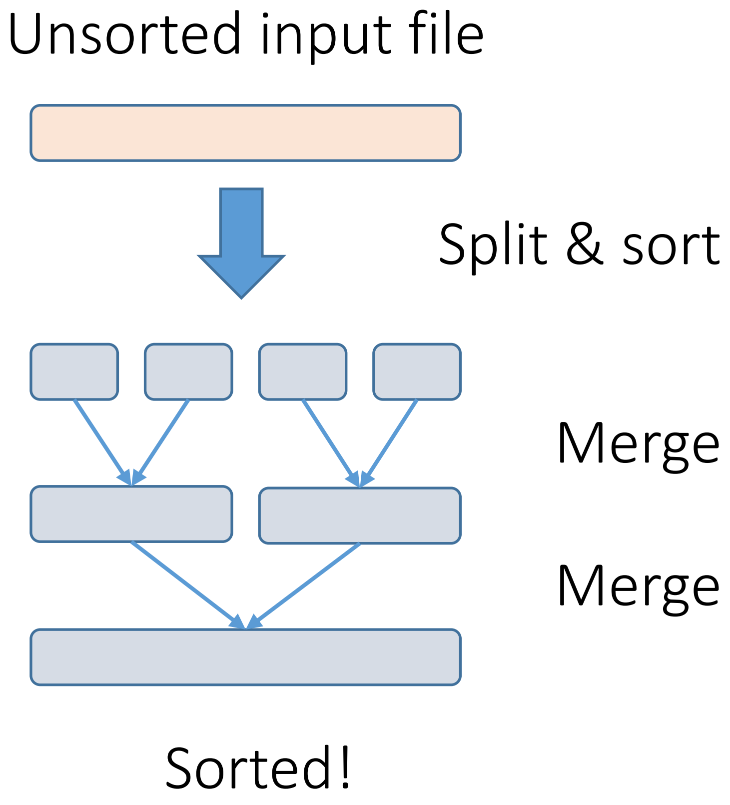


4. And repeat!

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 $\lceil \boldsymbol{log_2 N} \rceil$ passes
  - +1 for the initial split & sort
- Each pass involves reading in & writing out all the pages = *2N IO*



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

External Merge Sort: Optimizations

Now assume we have **B+1 buffer pages**; three optimizations:

1. Increase the length of initial runs
2. B-way merges
3. Repacking

Using B+1 buffer pages to reduce # of passes

Suppose we have B+1 buffer pages now; we can:

1. **Increase length of initial runs.** Sort B+1 at a time!

At the beginning, we can split the N pages into runs of length B+1 and sort these in memory

IO Cost:

$$2N(\lceil \log_2 N \rceil + 1) \Rightarrow 2N\left(\left\lceil \log_2 \frac{\boldsymbol{N}}{\boldsymbol{B+1}} \right\rceil + 1\right)$$

Starting with runs of length 1

Starting with runs of length ***B+1***

Using B+1 buffer pages to reduce # of passes

Suppose we have B+1 buffer pages now; we can:

2. Perform a B-way merge.

On each pass, we can merge groups of *B* runs at a time (vs. merging pairs of runs)!

<u>IO Cost:</u>

$$2N(\lceil \log_2 N \rceil + 1)$$

Starting with runs of length 1

$$2N\left(\left\lceil \log_2 \frac{\boldsymbol{N}}{\boldsymbol{B+1}} \right\rceil + 1\right)$$

Starting with runs of length ***B+1***

$$2N\left(\left\lceil \log_B \frac{\boldsymbol{N}}{\boldsymbol{B+1}} \right\rceil + 1\right)$$

Performing ***B***-way merges

Repacking for even longer initial runs

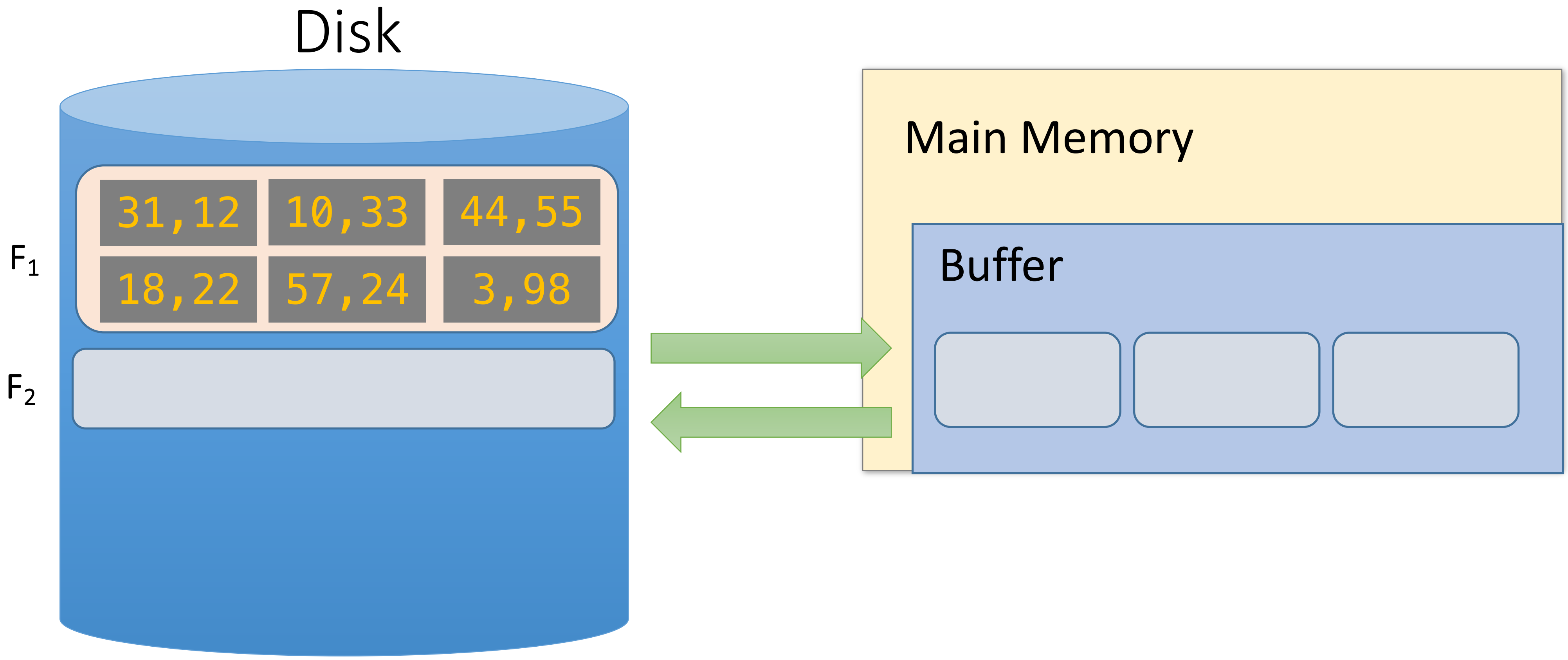
- With B+1 buffer pages, we can now start with ***B+1-length initial runs*** (and use ***B-way merges***) to get $2N(\left\lceil \log_B \frac{\boldsymbol{N}}{\boldsymbol{B+1}} \right\rceil + 1)$ IO cost…

- Can we reduce this cost more by getting even longer initial runs?

- Use <u>repacking</u>- produce longer initial runs by "merging" in buffer as we sort at initial stage

Repacking Example: 3 page buffer

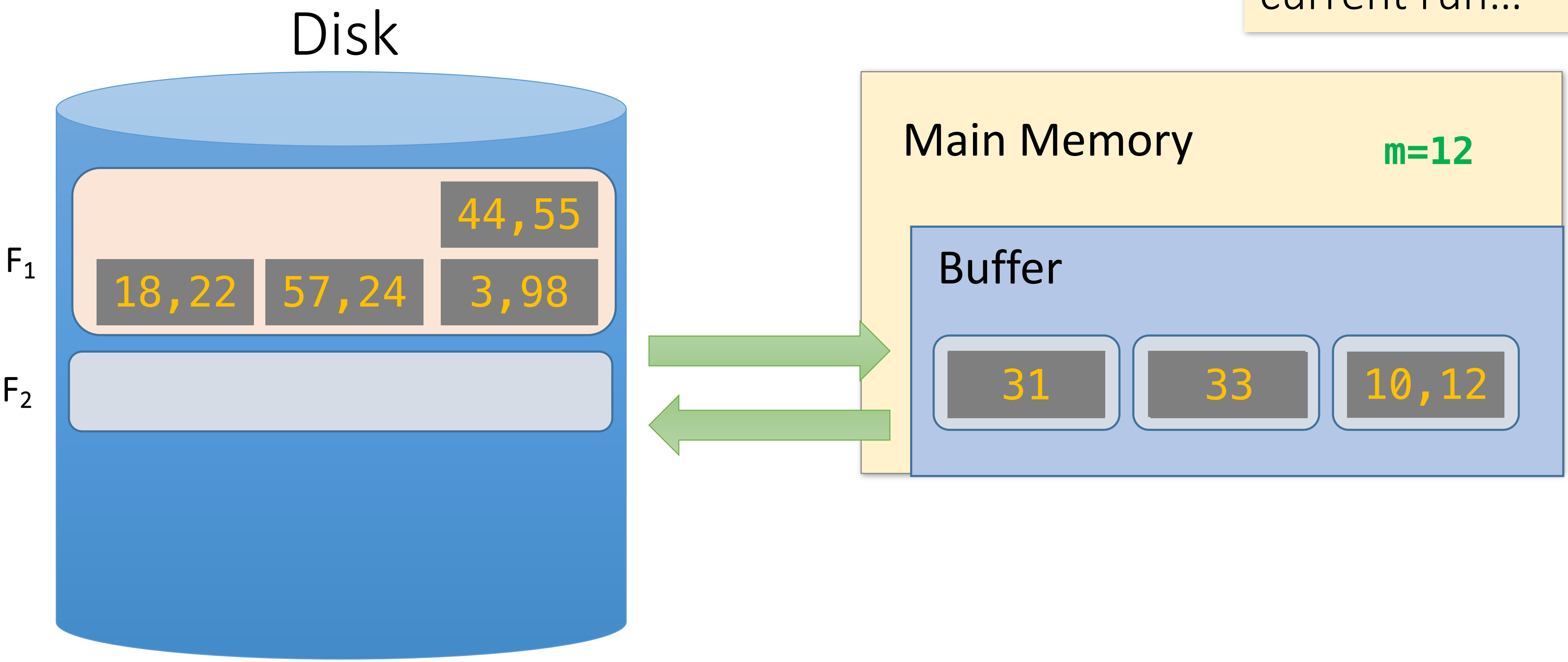
- Start with unsorted single input file, and load 2 pages



Repacking Example: 3 page buffer

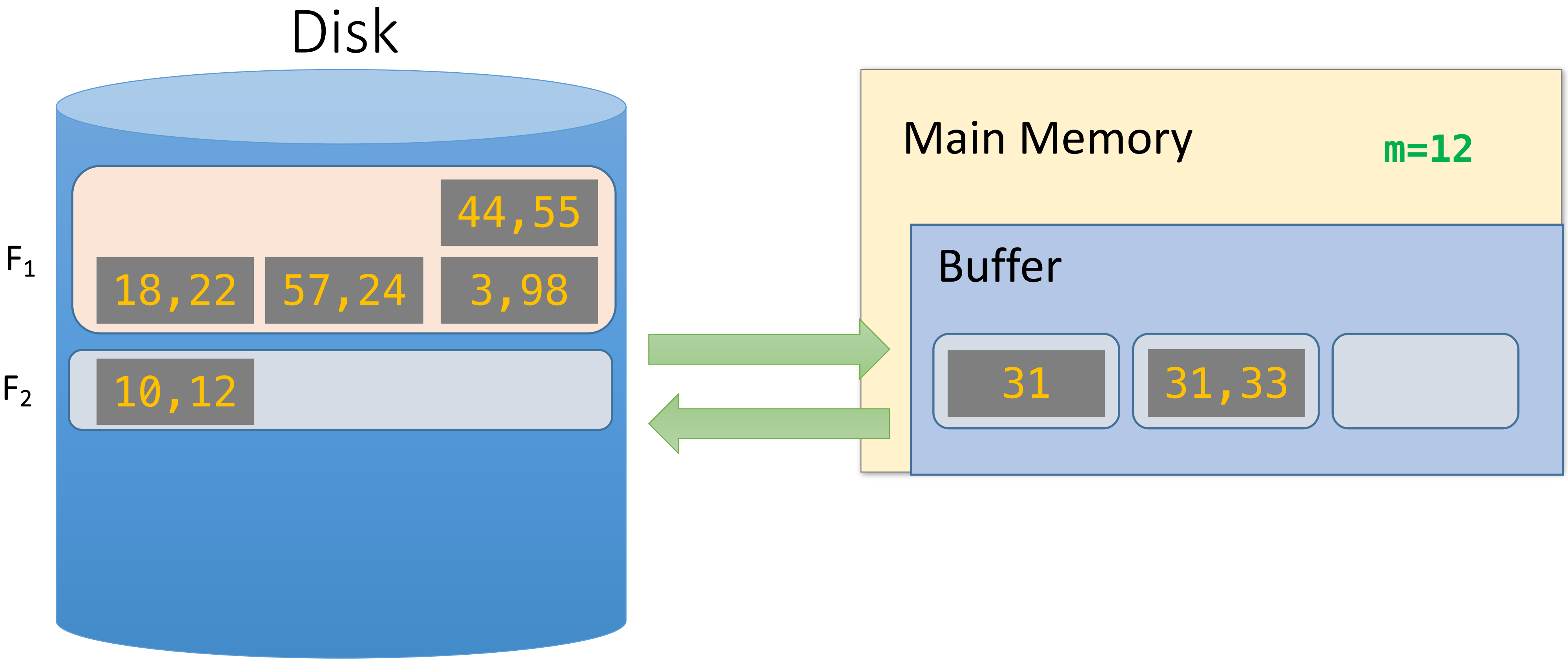
- Take the minimum two values, and put in output page

Also keep track of max (last) value in current run...



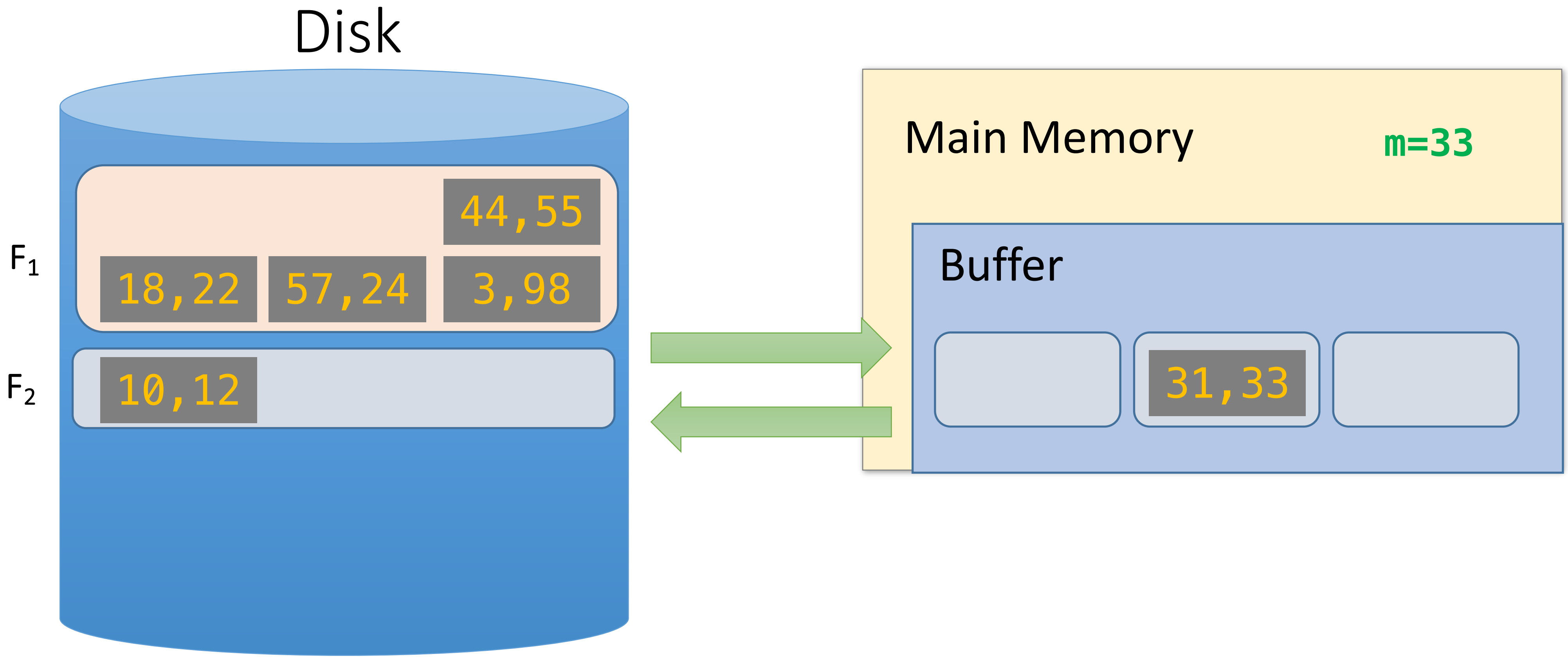
Repacking Example: 3 page buffer

- Next, *repack*



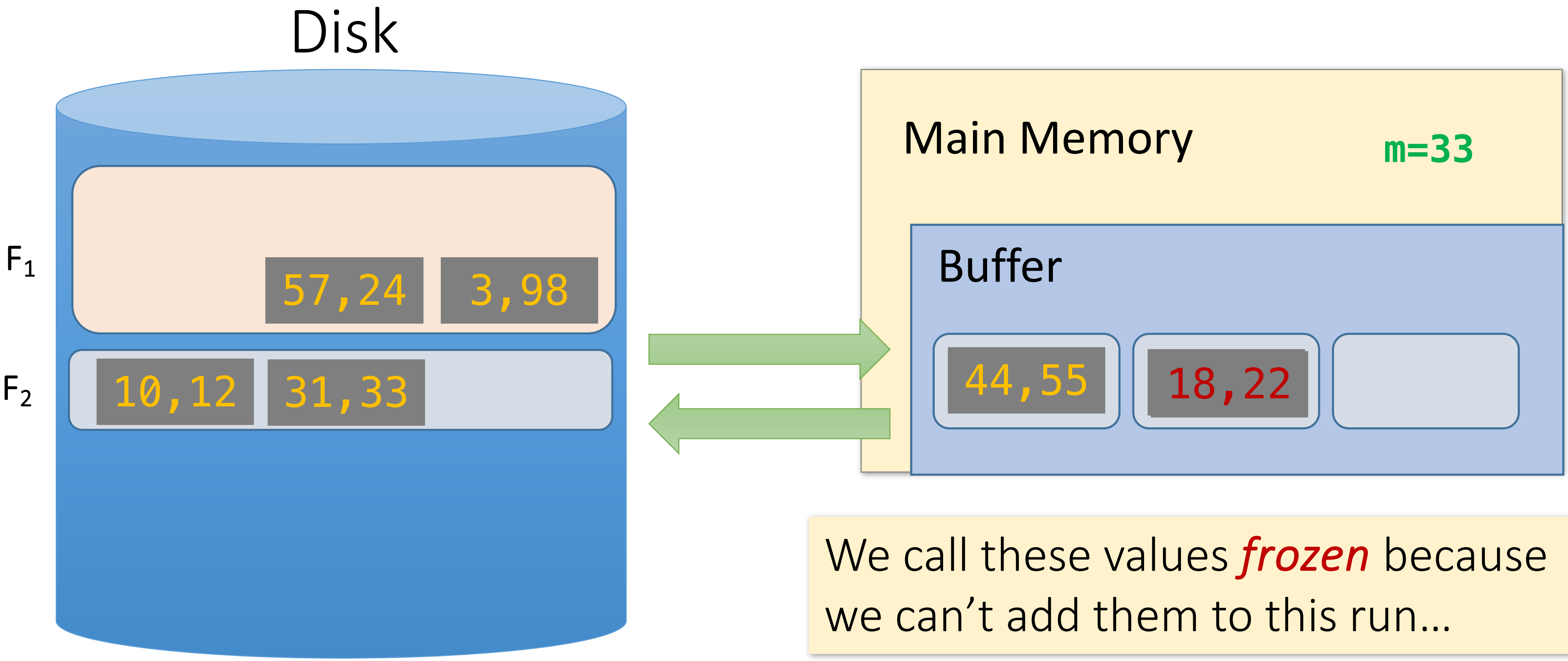
Repacking Example: 3 page buffer

- Next, *repack*, then load another page and continue!



Repacking Example: 3 page buffer

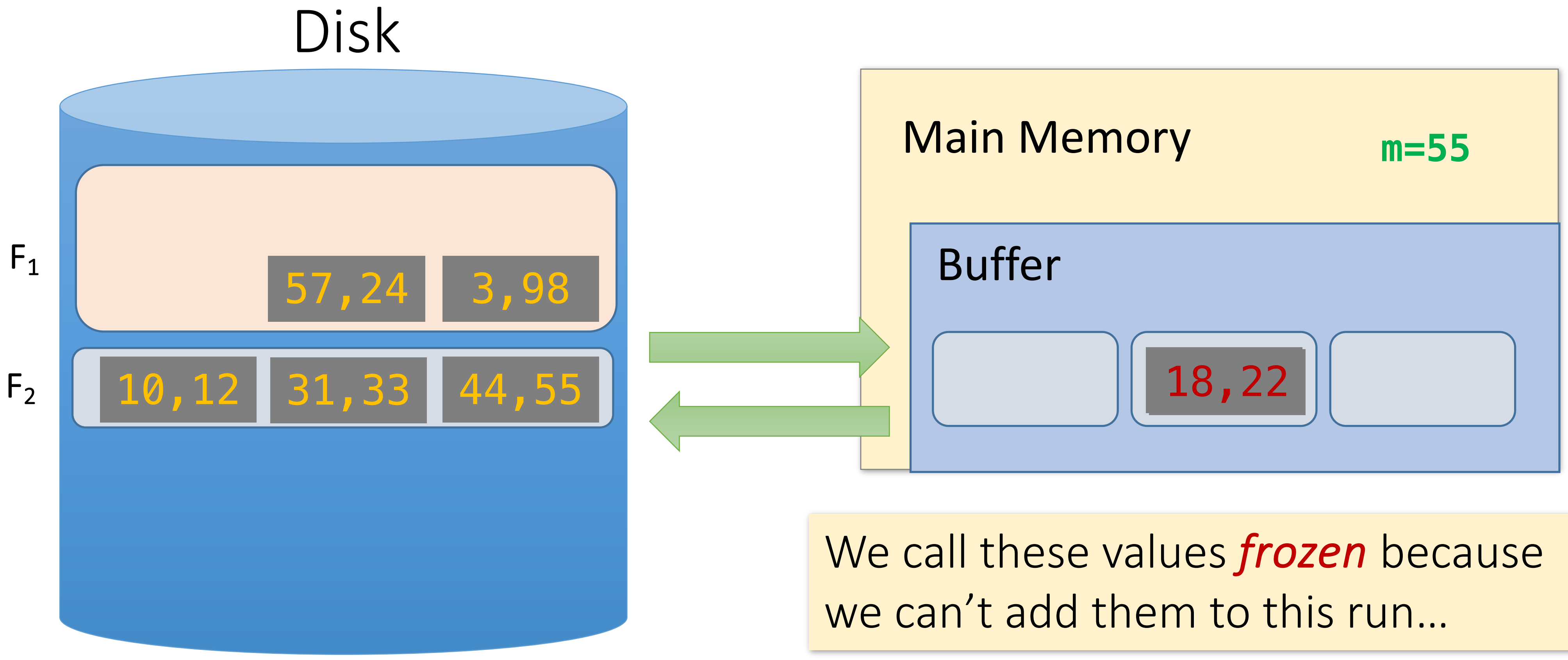
- Now, however, *the smallest values are less than the largest (last) in the sorted run…*



We call these values ***frozen*** because we can't add them to this run…

Repacking Example: 3 page buffer

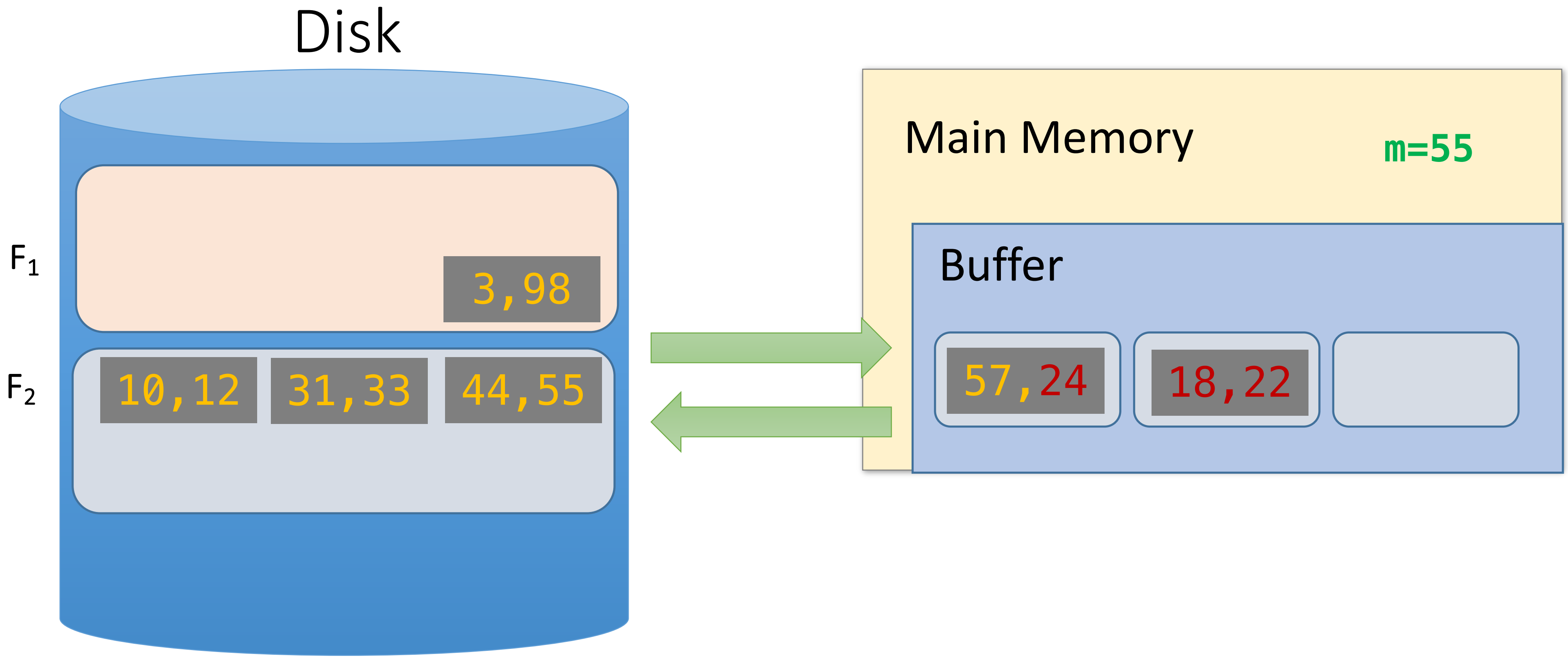
- Now, however, *the smallest values are less than the largest (last) in the sorted run…*



We call these values ***frozen*** because we can't add them to this run…

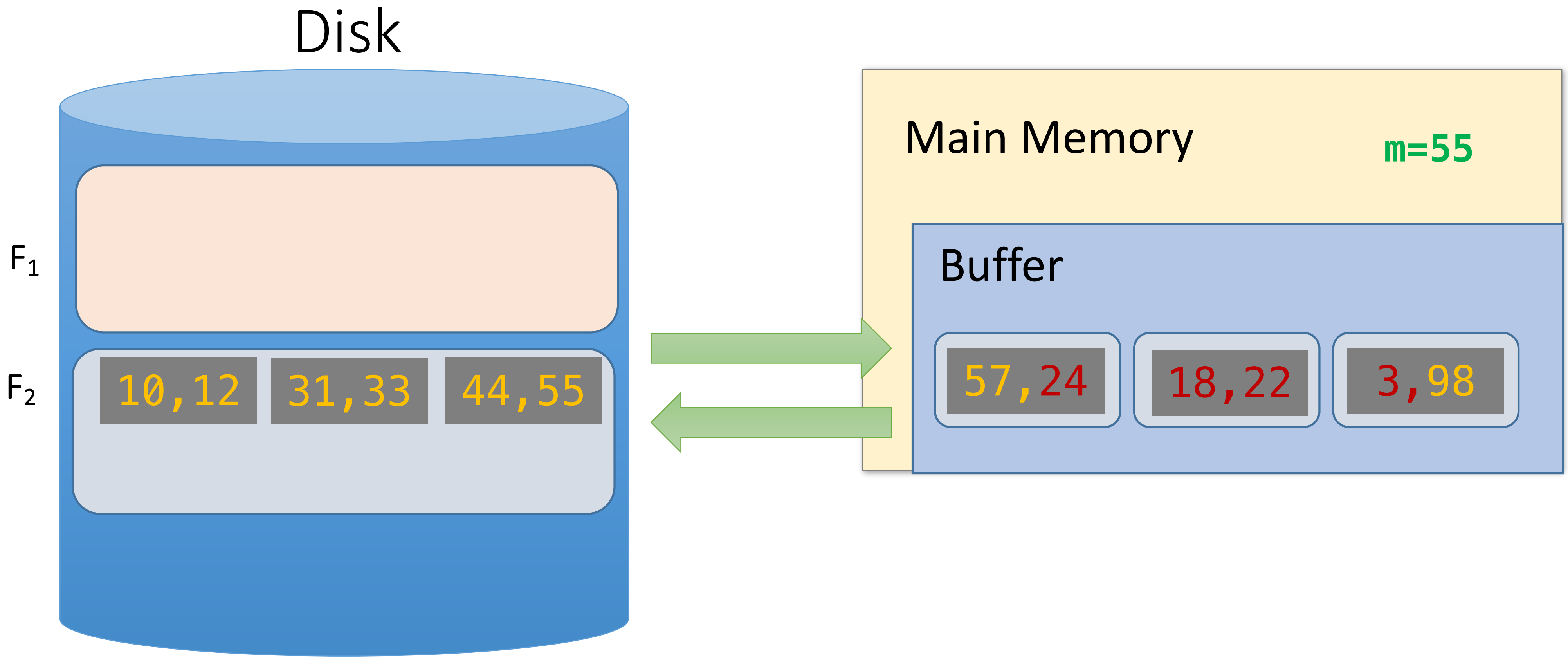
Repacking Example: 3 page buffer

- Now, however, *the smallest values are less than the largest (last) in the sorted run…*



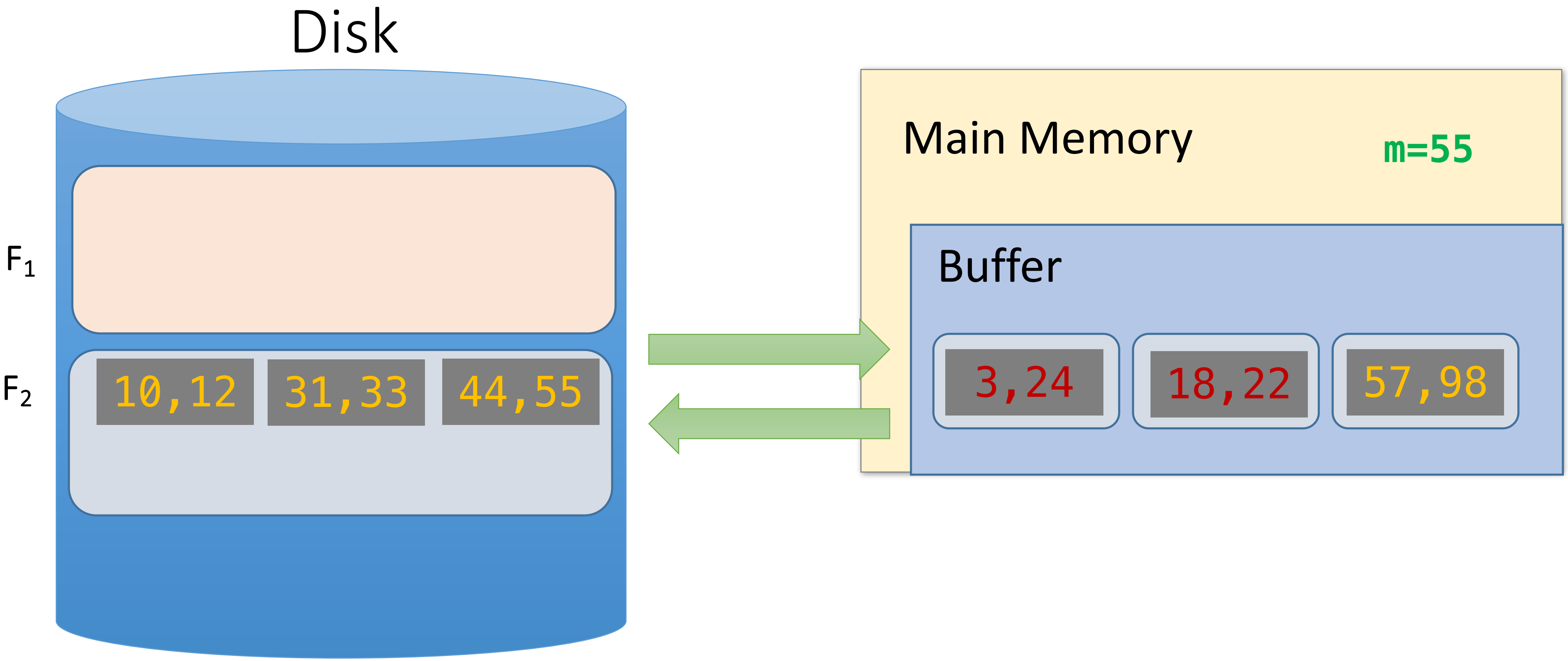
Repacking Example: 3 page buffer

- Now, however, *the smallest values are less than the largest (last) in the sorted run…*



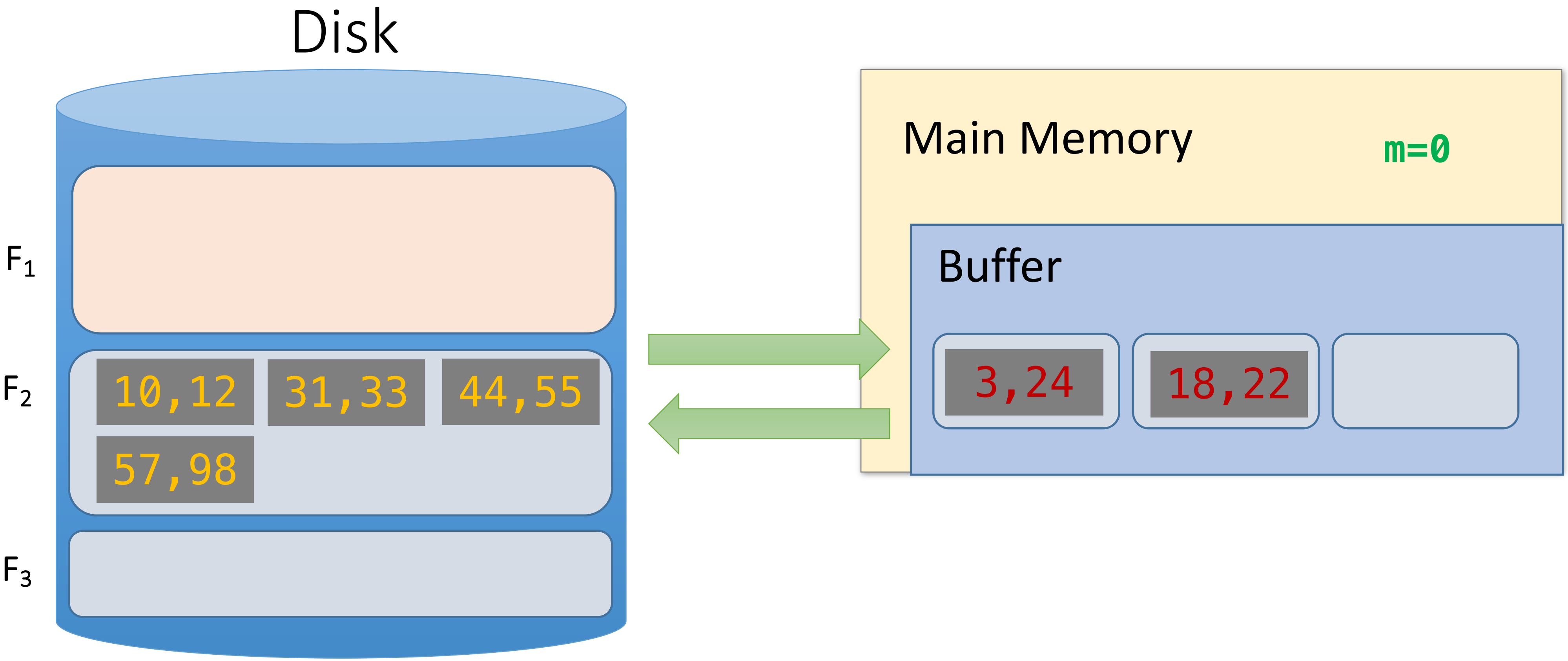
Repacking Example: 3 page buffer

- Now, however, *the smallest values are less than the largest (last) in the sorted run…*



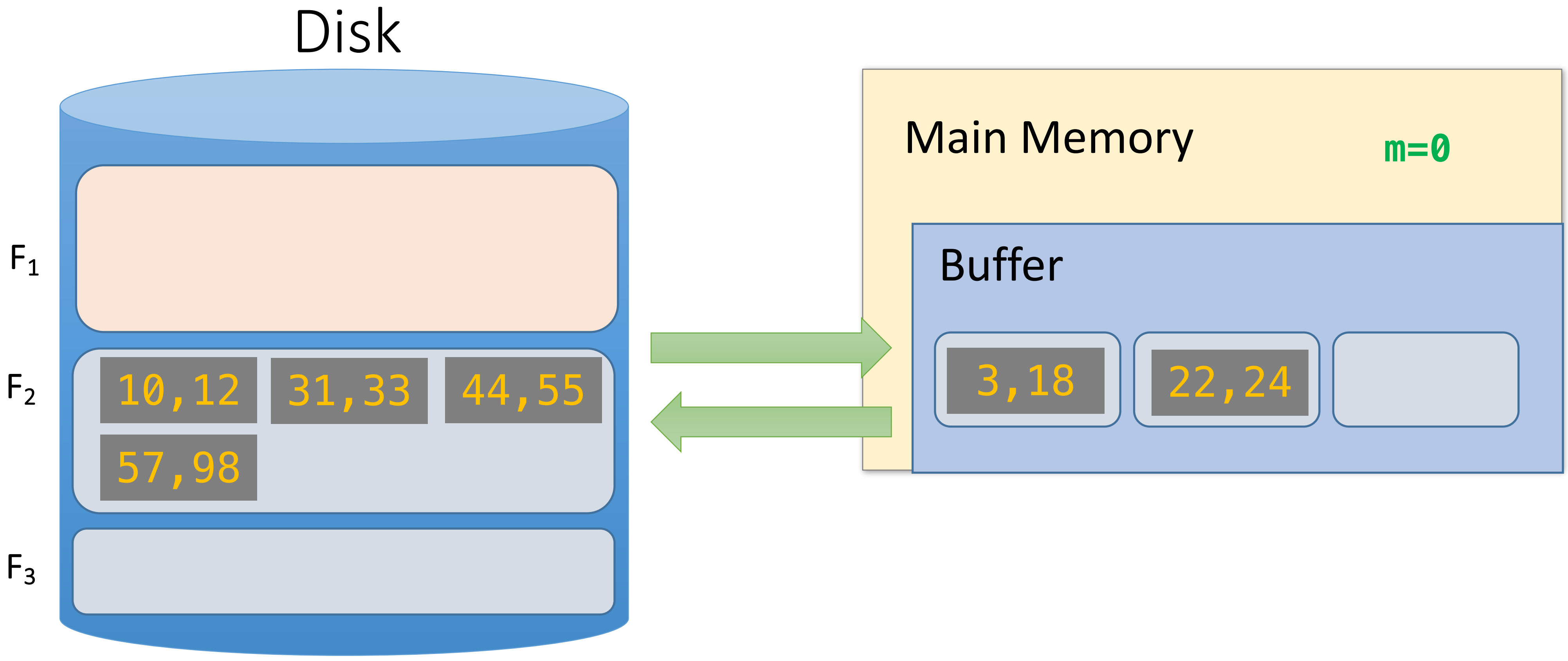
Repacking Example: 3 page buffer

- Once ***all buffer pages have a frozen value,*** or input file is empty, start new run with the frozen values



Repacking Example: 3 page buffer

- Once ***all buffer pages have a frozen value,*** or input file is empty, start new run with the frozen values



Repacking

- Note that, for buffer with B+1 pages:
  - **Best case:** If input file is sorted → nothing is frozen → we get **a single** run!
  - **Worst case:** If input file is reverse sorted → everything is frozen → we get runs of length **B+1**
- In general, with repacking we do <u>**no worse**</u> than without it!
- Engineer's approximation: runs will have **~2(B+1)** length

$$\sim 2N\left(\left\lceil \log_B \frac{\boldsymbol{N}}{\mathbf{2}(\boldsymbol{B}+\mathbf{1})} \right\rceil + 1\right)$$

Summary

- Basics of IO and buffer management.
- We introduced the IO cost model using **sorting**.
  - Saw how to do merges with few IOs,
  - Works better than main-memory sort algorithms.
- Described a few optimizations for sorting