CS 4440 A

Emerging Database Technologies

Lecture 11

02/23/26

Announcements

- Assignment 2 due next Wednesday (March 4)
- Assignment 3 will be released today
    - Presentation groups: People -> Presentation Groups

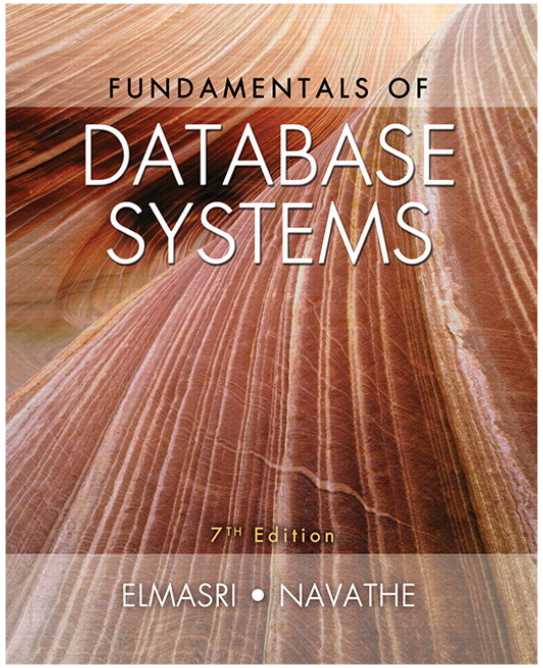
Agenda

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.

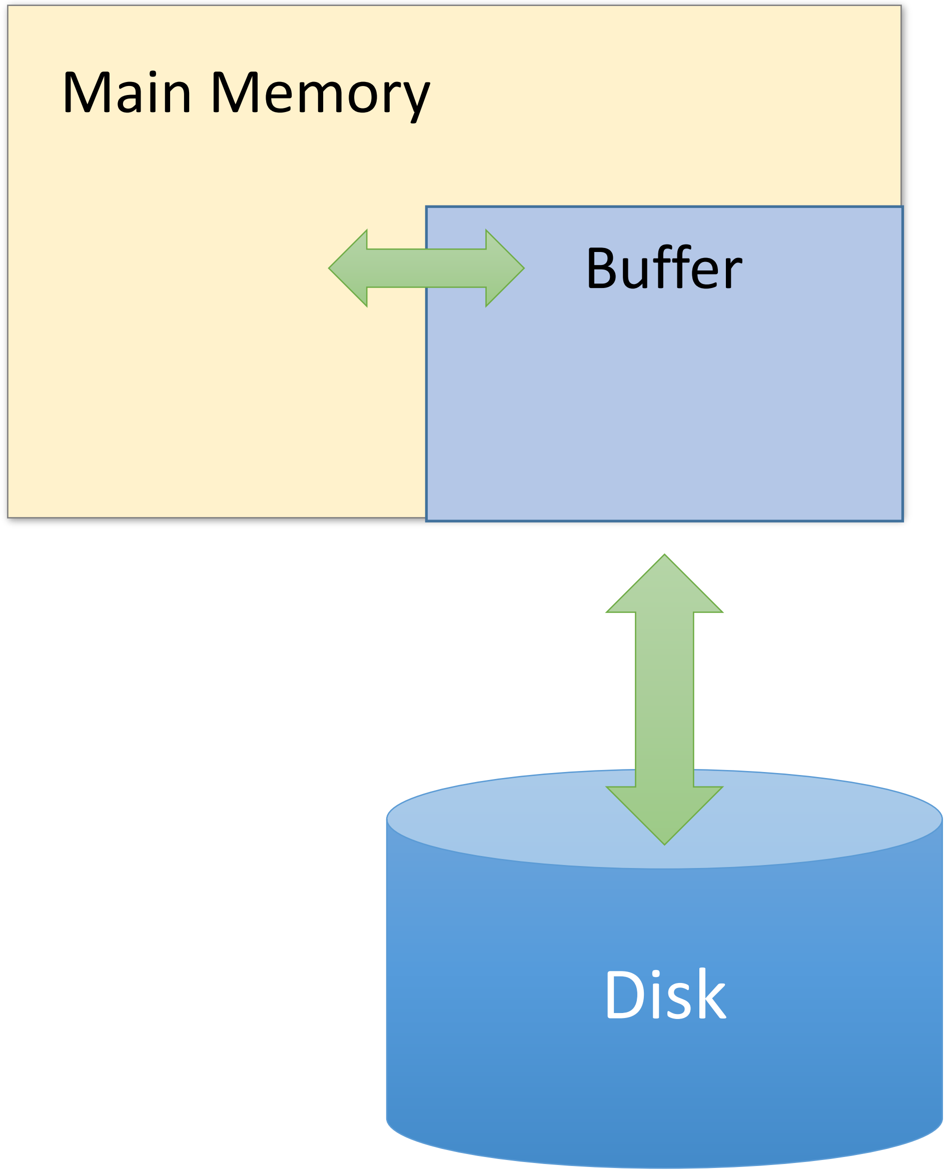
1. The Buffer

The Buffer

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

- A region in main memory used to store **intermediate data between disk and processes**

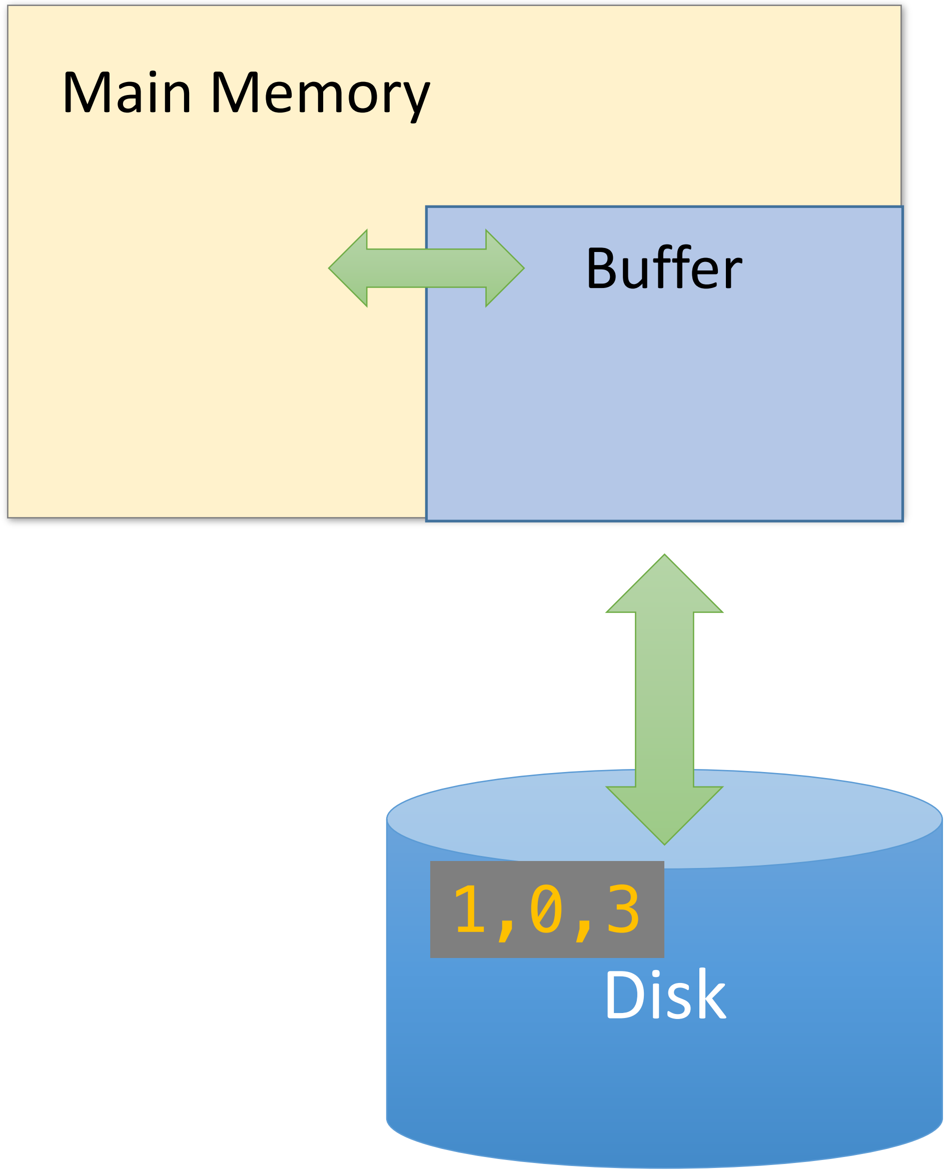
Key idea: Reading / writing to disk is slow- need to cache data!



The (Simplified) Buffer

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

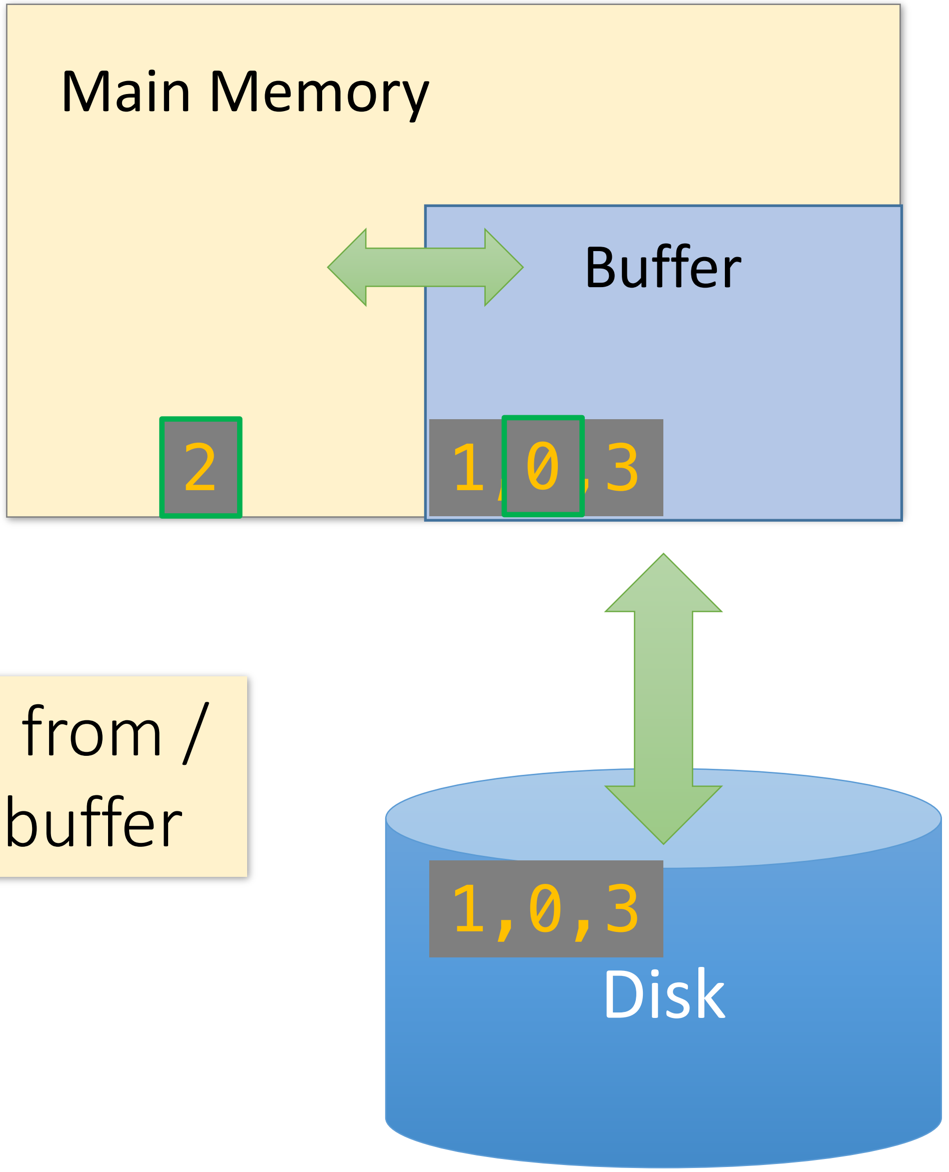
- Read(page): Read page from disk -> buffer if not already in buffer



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

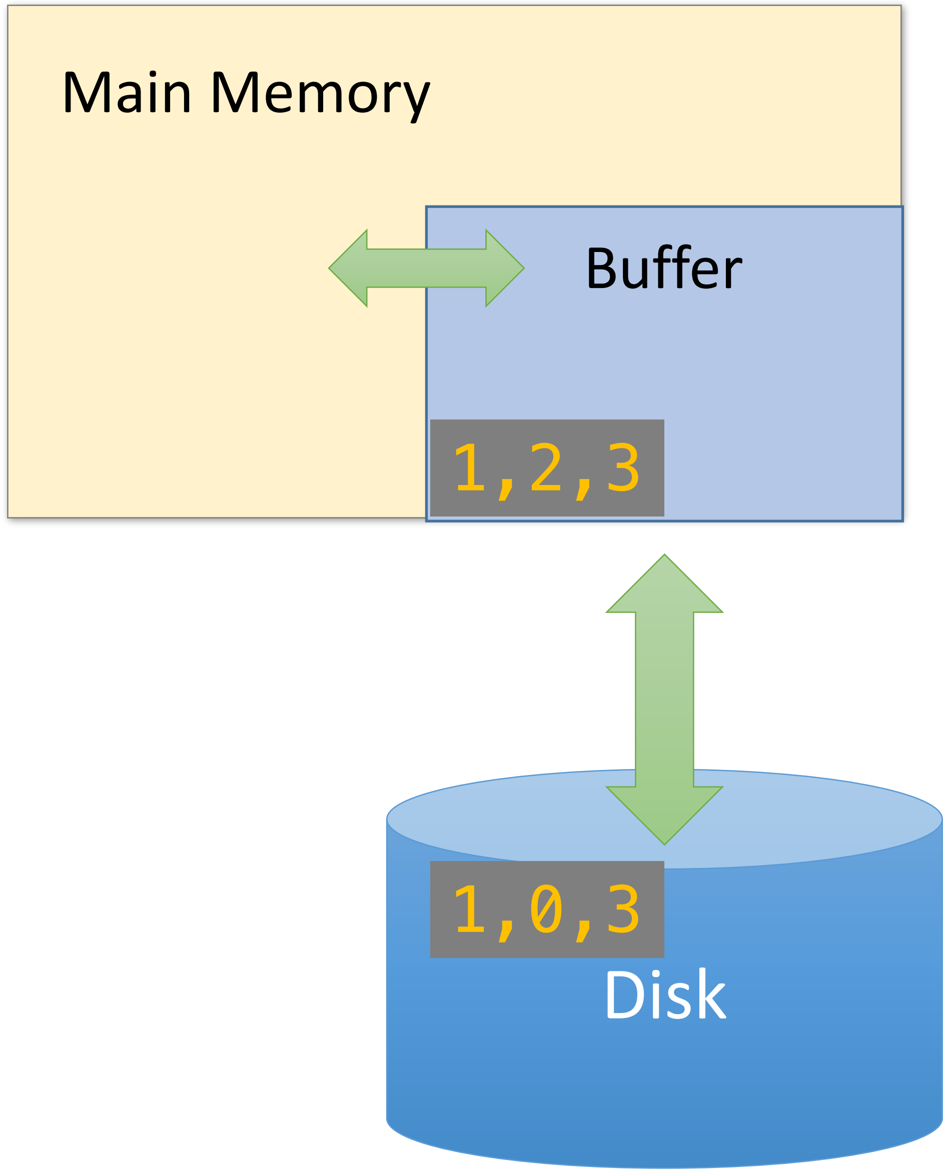


Processes can then read from / write to the page in the buffer

The (Simplified) Buffer

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

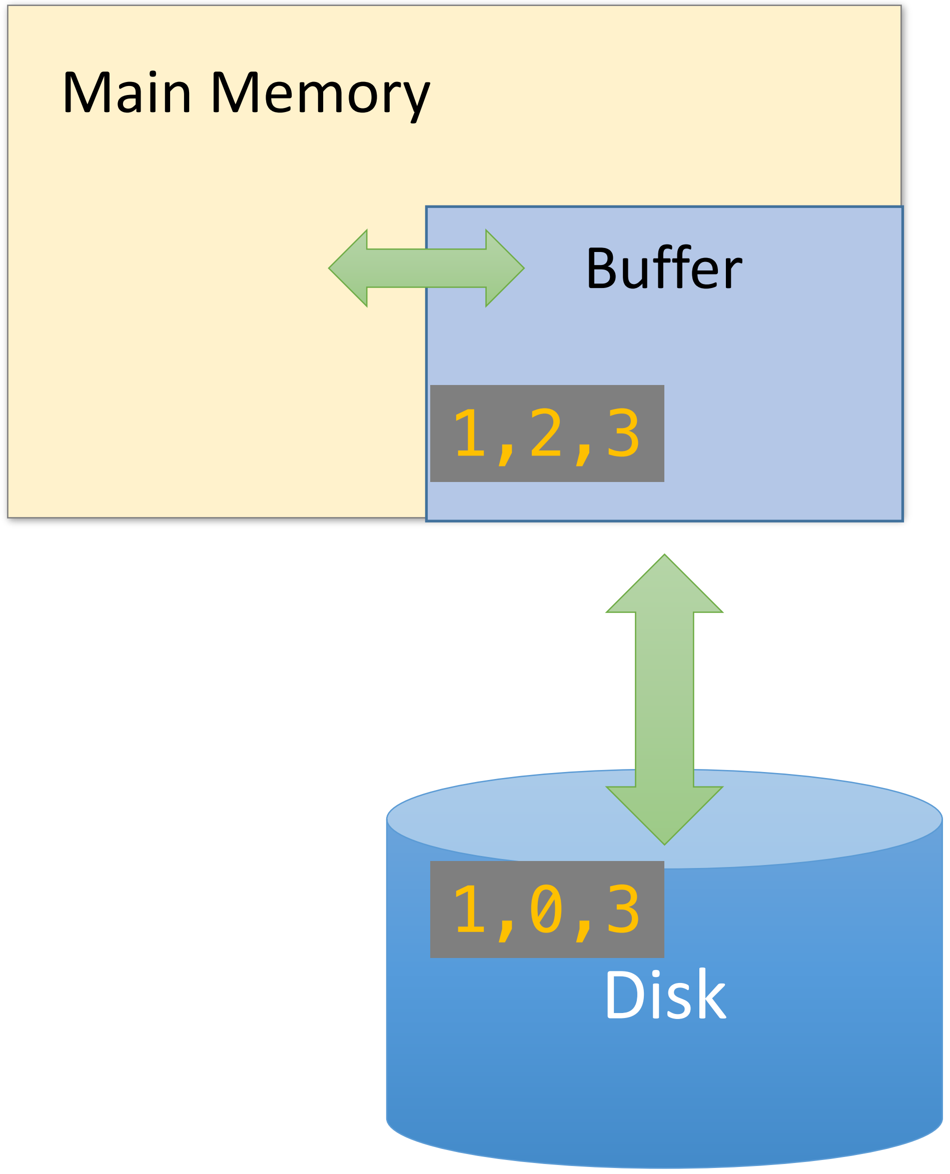
- Read(page): Read page from disk -> buffer if not already in buffer
- Flush(page): Evict page from buffer & write to disk



The (Simplified) Buffer

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

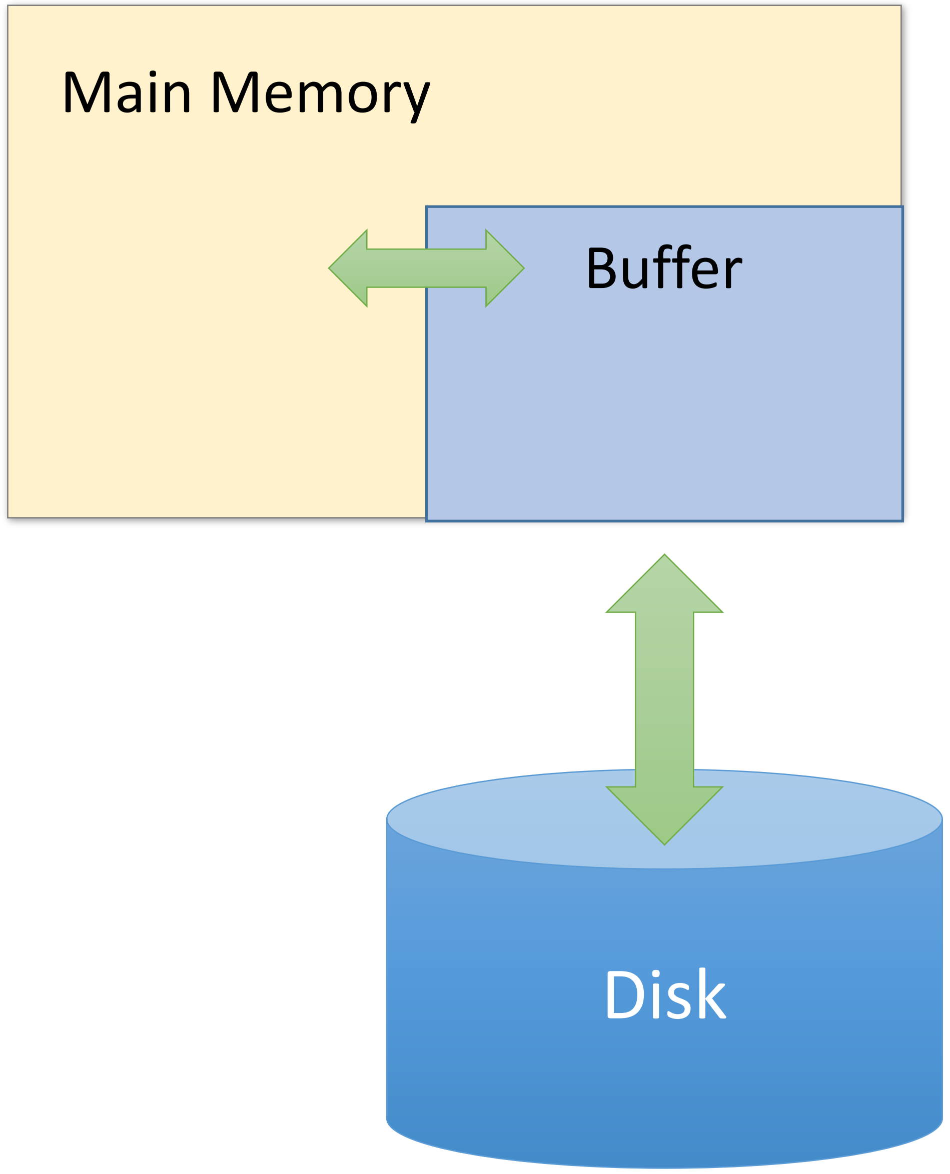
- <u>Read(page):</u> Read page from disk -> buffer if not already in buffer
- <u>Flush(page):</u> Evict page from buffer & write to disk
- <u>Release(page):</u> Evict page from buffer without writing to disk



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
  - DBAs can configure the appropriate buffer size

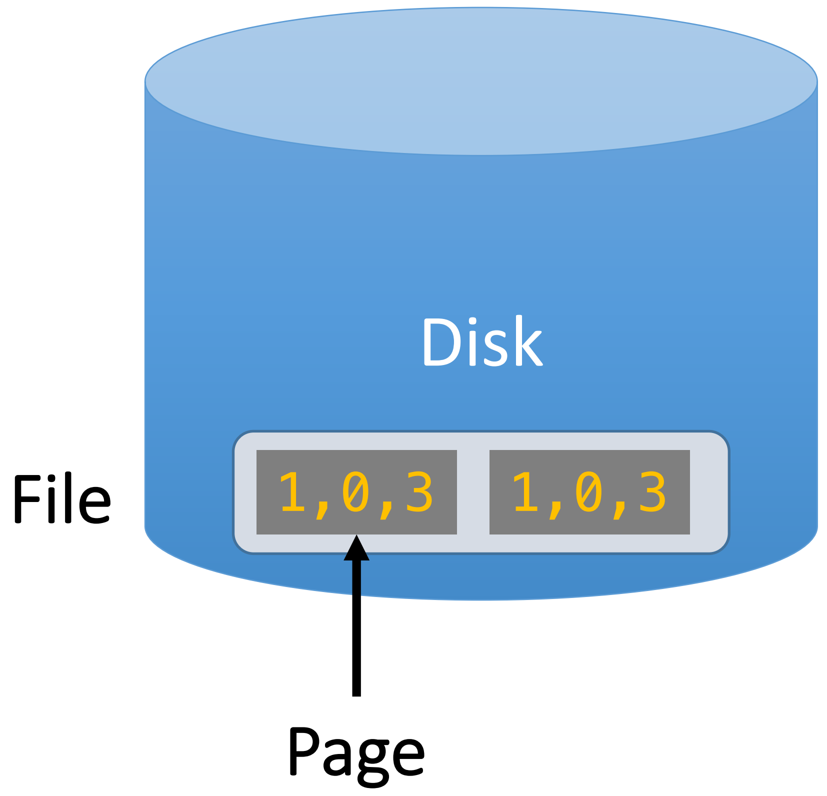
A Simplified Filesystem Model

For us, a page 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 file 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

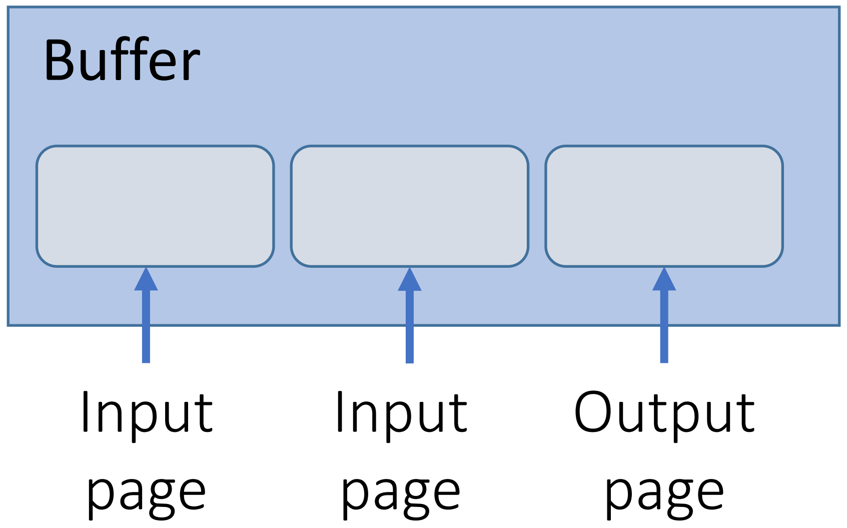
- How do we *efficiently* merge two sorted files when both are much larger than our main memory buffer?

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

Our first 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:

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

Then:

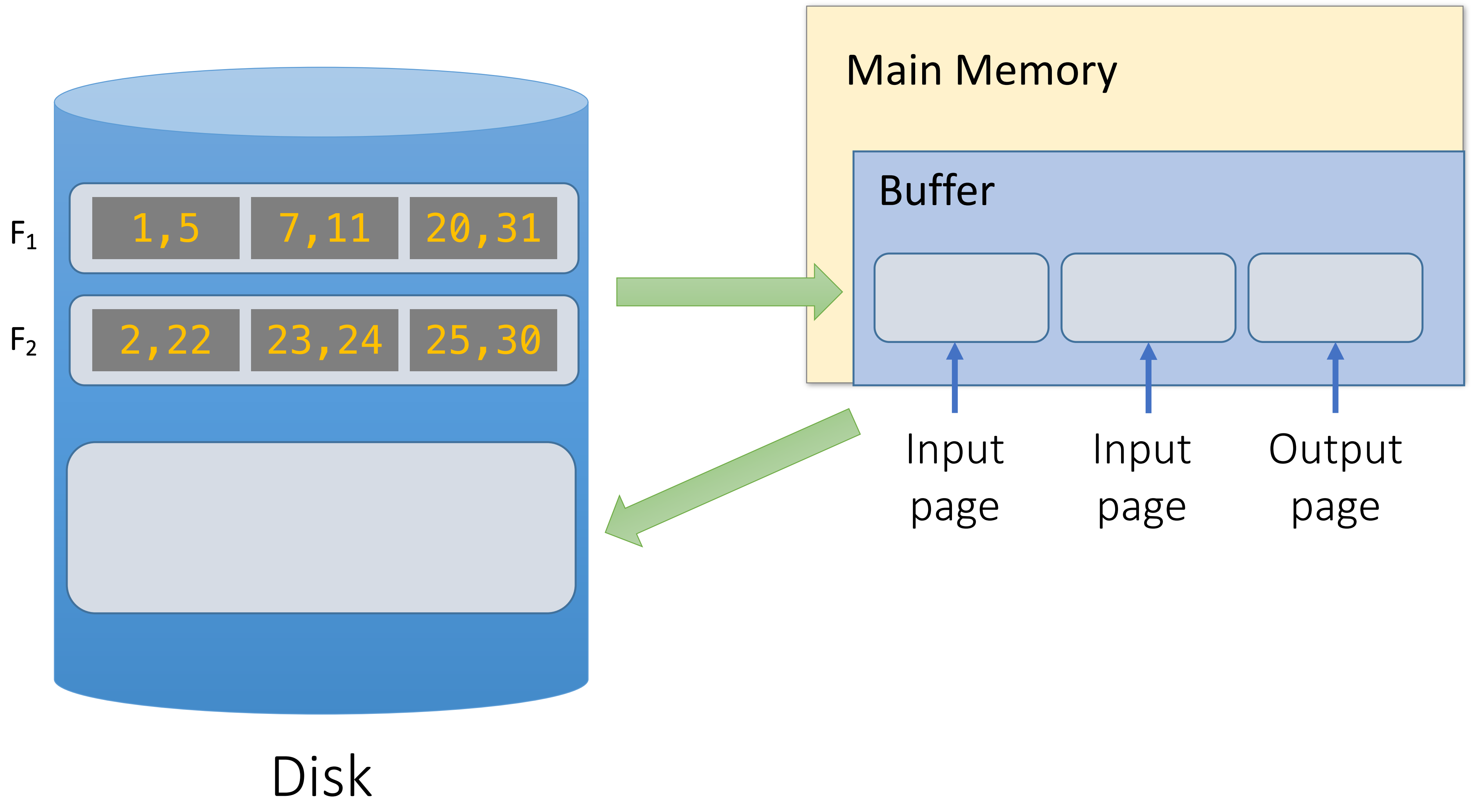
$$Min(A_1, B_1) \leq A_i$$
$$Min(A_1, B_1) \leq B_j$$

for i=1….N and j=1….M

External Merge Algorithm

Input:
Two sorted files

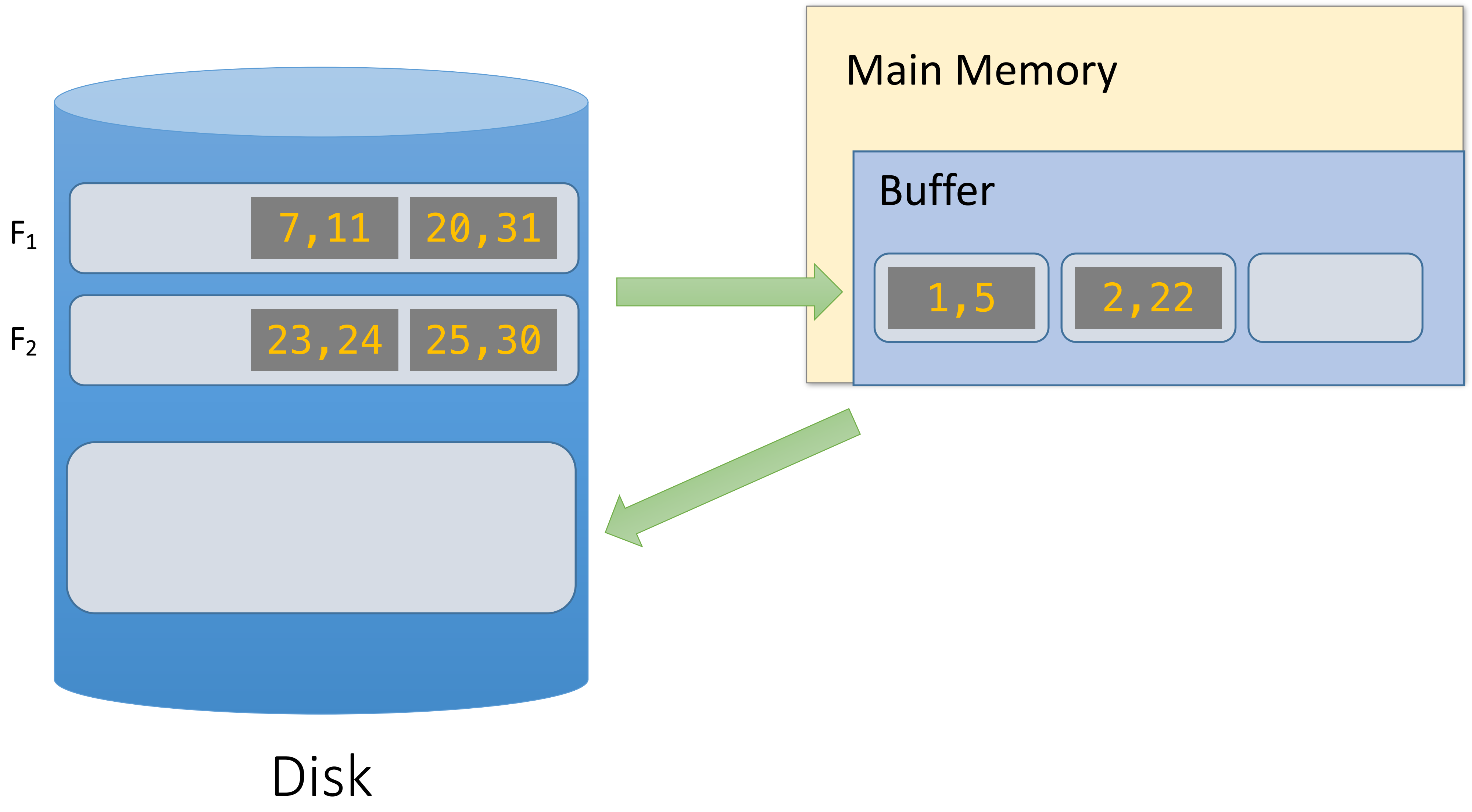
Output:
One *merged* sorted file



External Merge Algorithm

Input:
Two sorted files

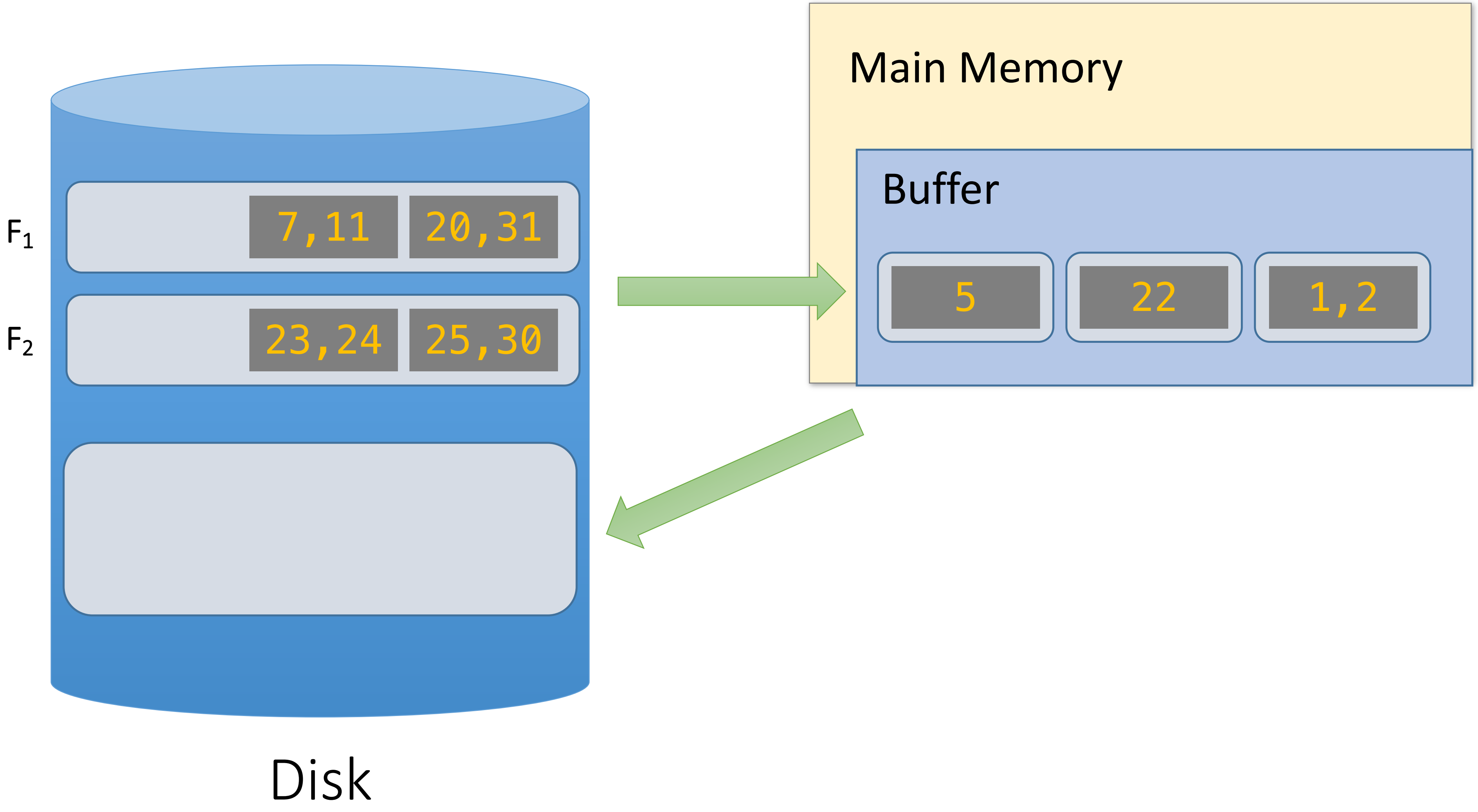
Output:
One *merged* sorted file



External Merge Algorithm

Input:
Two sorted
files

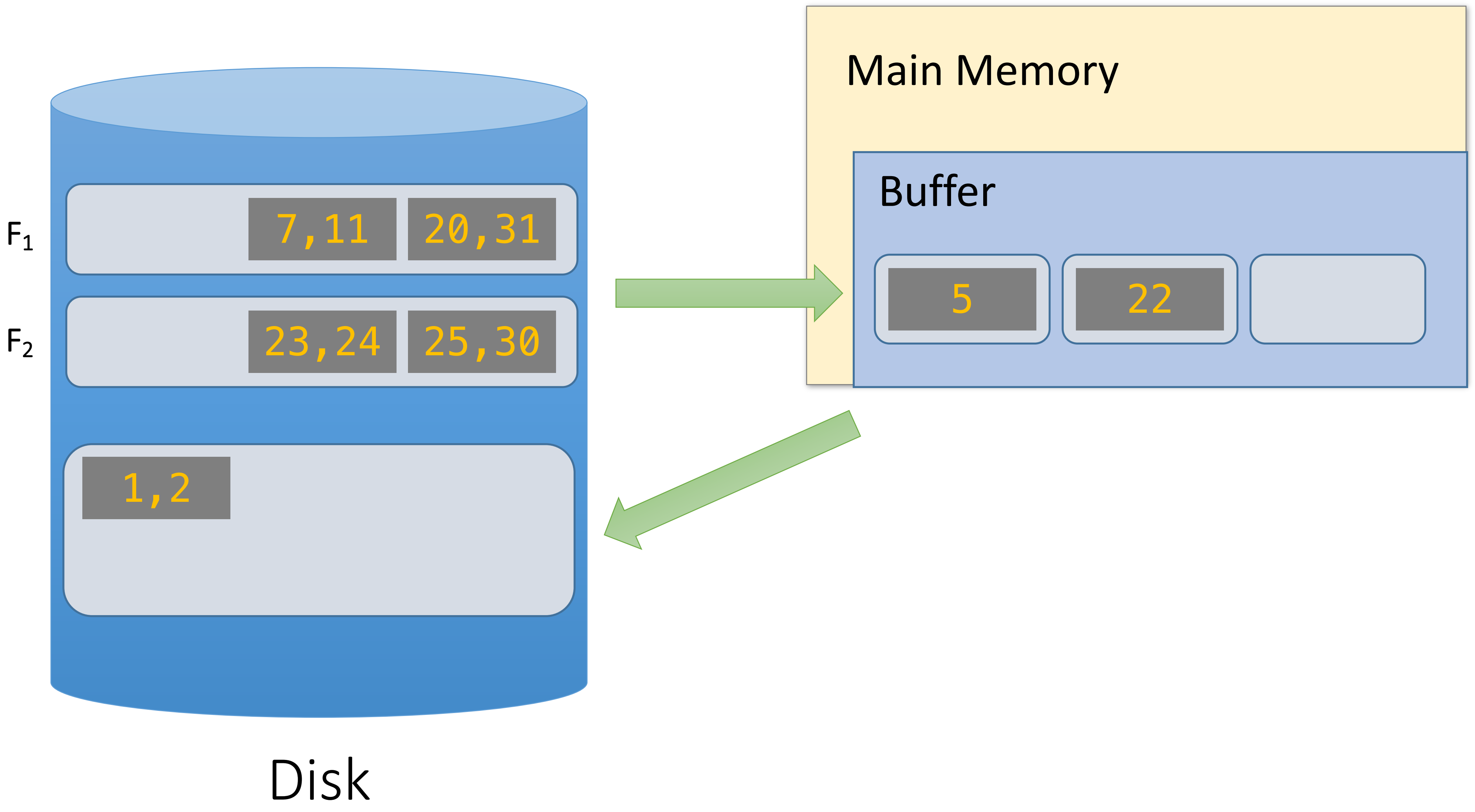
Output:
One *merged*
sorted file



External Merge Algorithm

Input:
Two sorted files

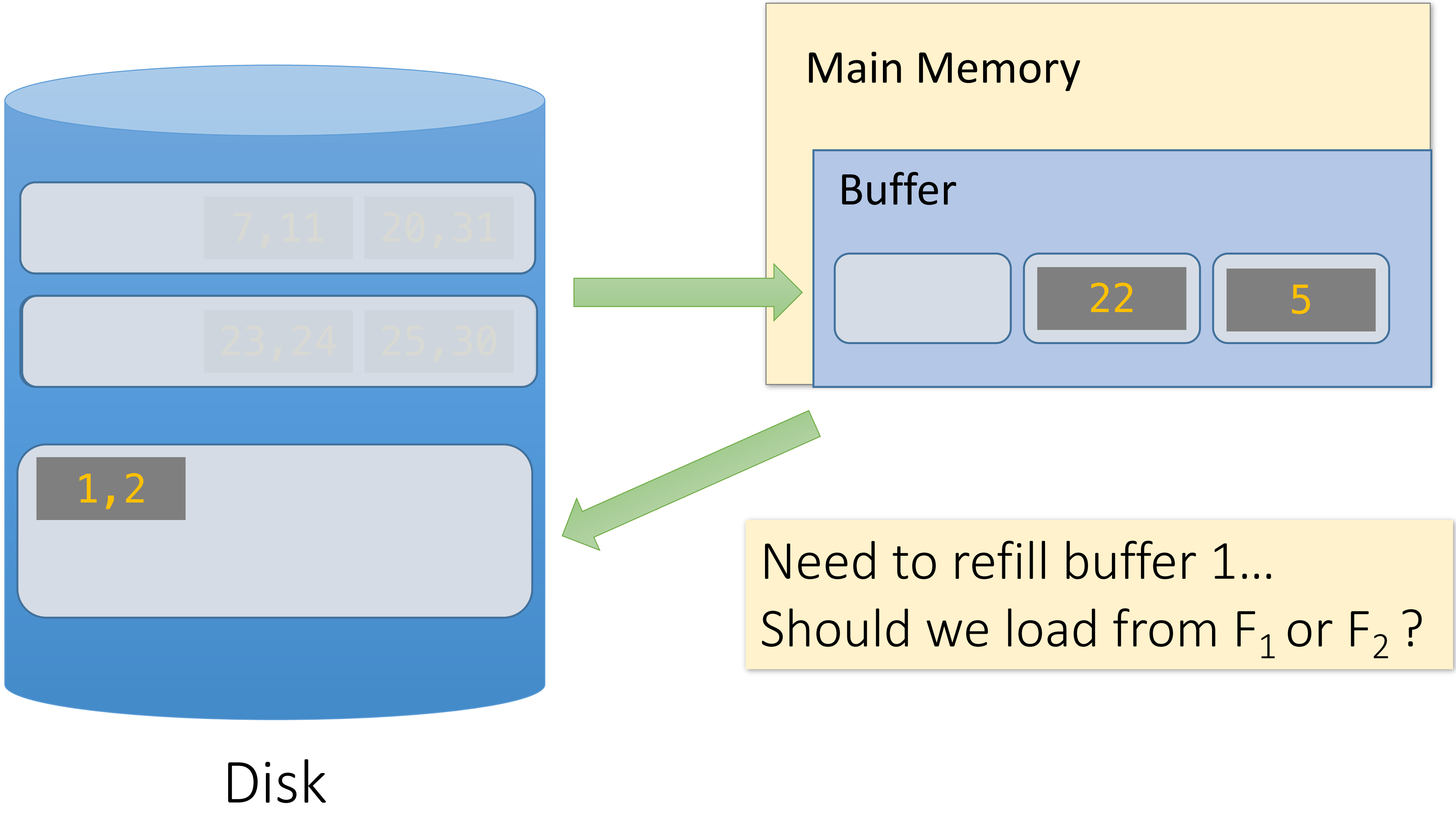
Output:
One *merged* sorted file



External Merge Algorithm

Input:
Two sorted
files

Output:
One *merged*
sorted file

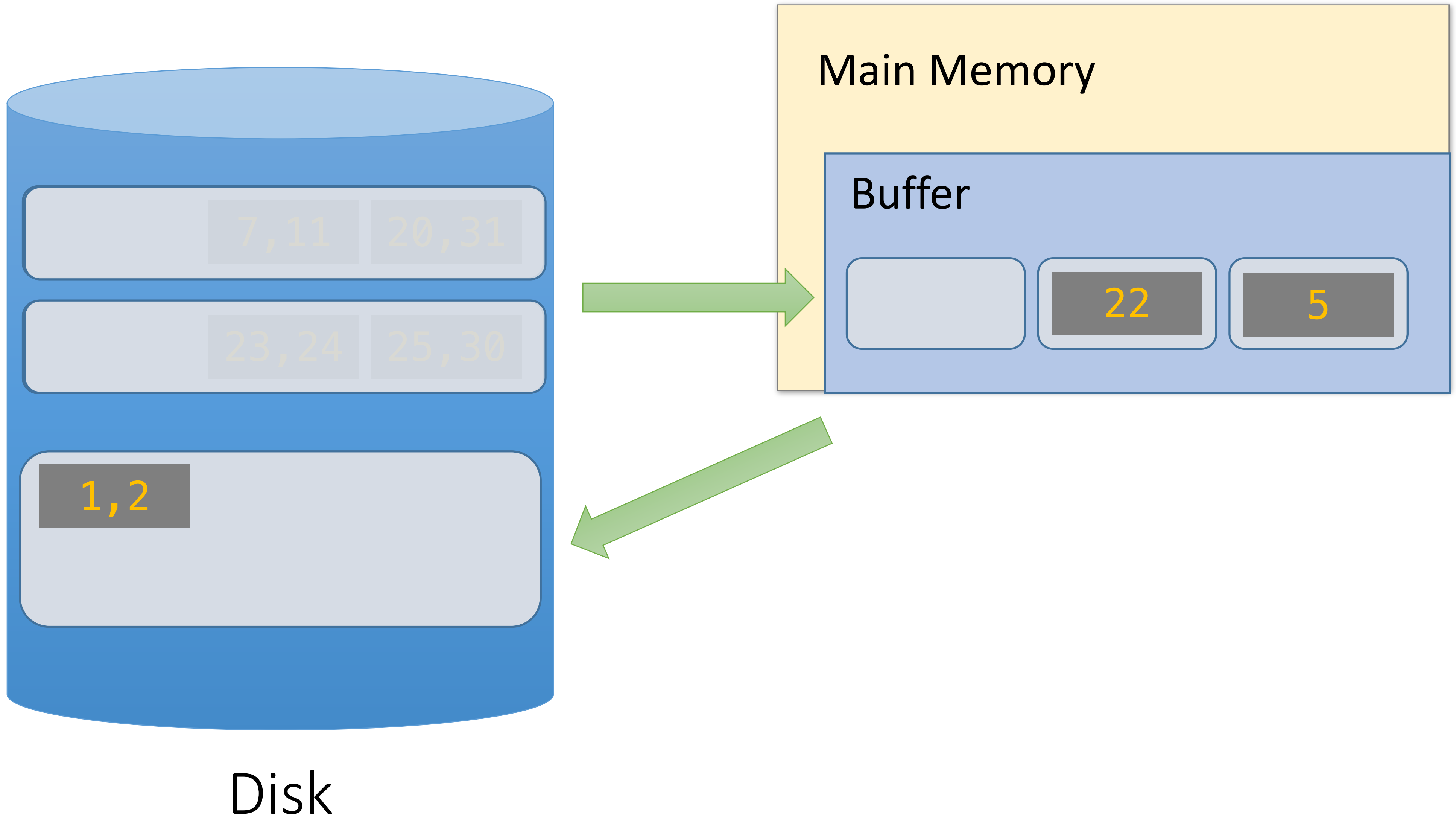


Need to refill buffer 1…
Should we load from F_1 or F_2 ?

External Merge Algorithm

Input:
Two sorted
files

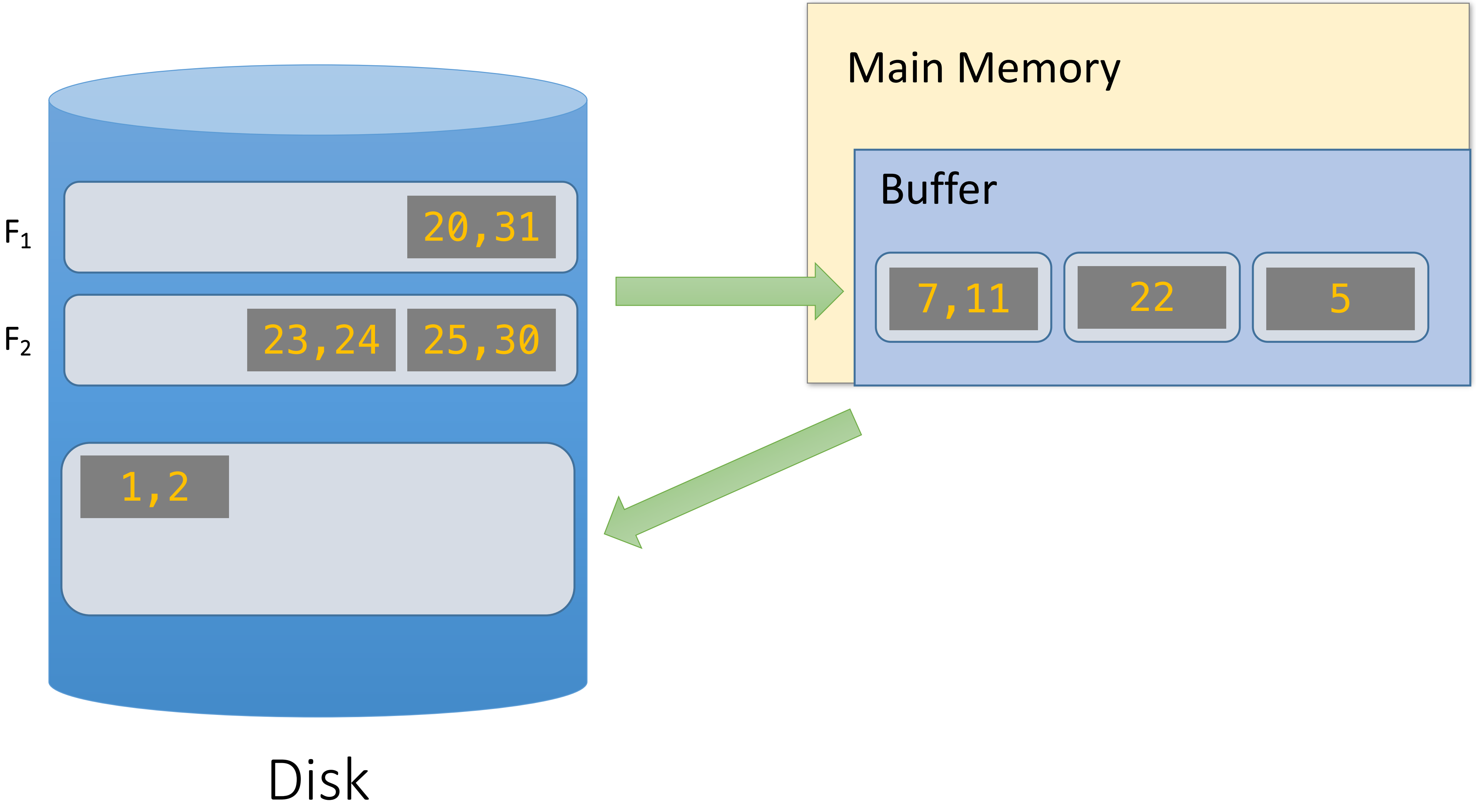
Output:
One *merged*
sorted file



External Merge Algorithm

Input:
Two sorted files

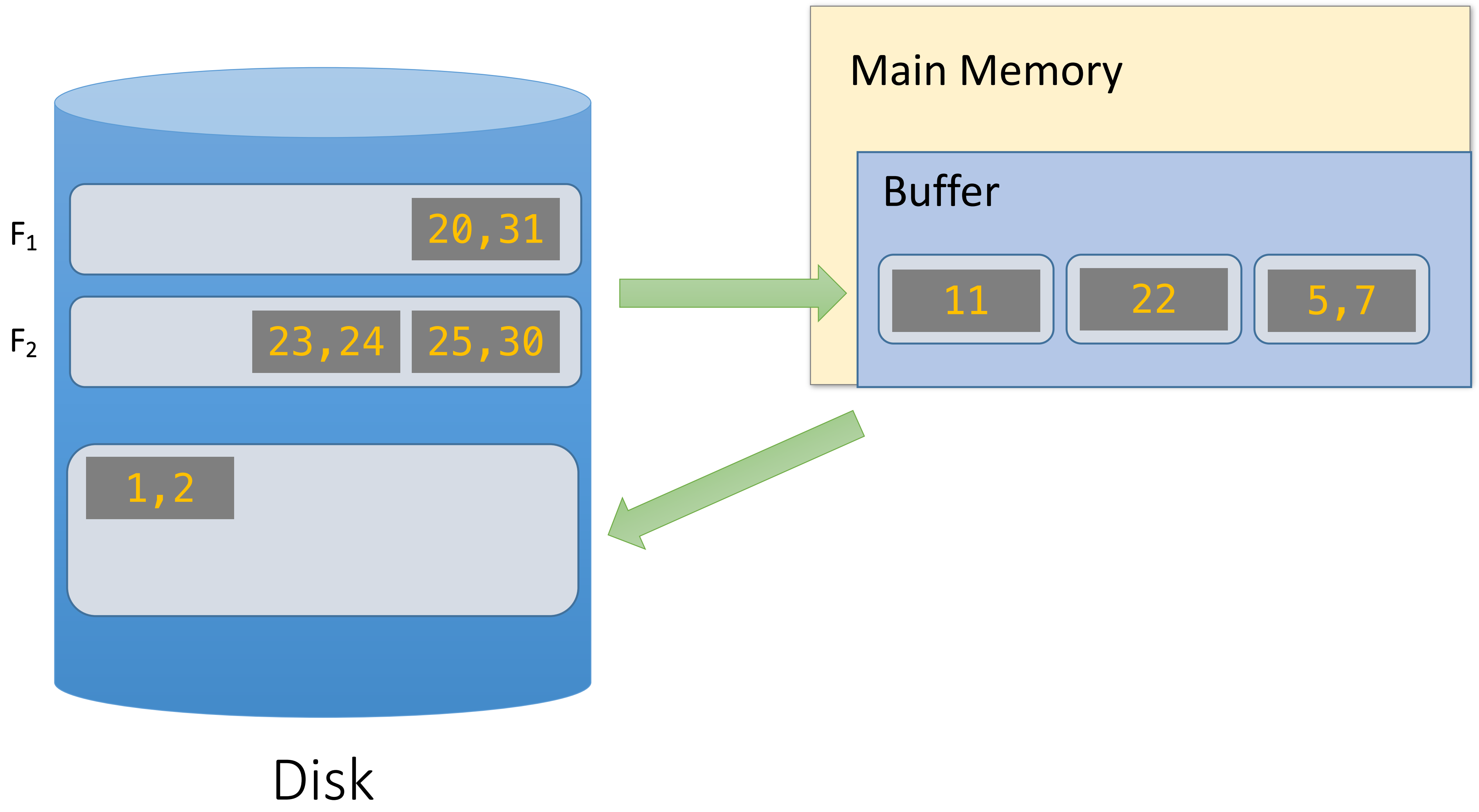
Output:
One *merged* sorted file



External Merge Algorithm

Input:
Two sorted
files

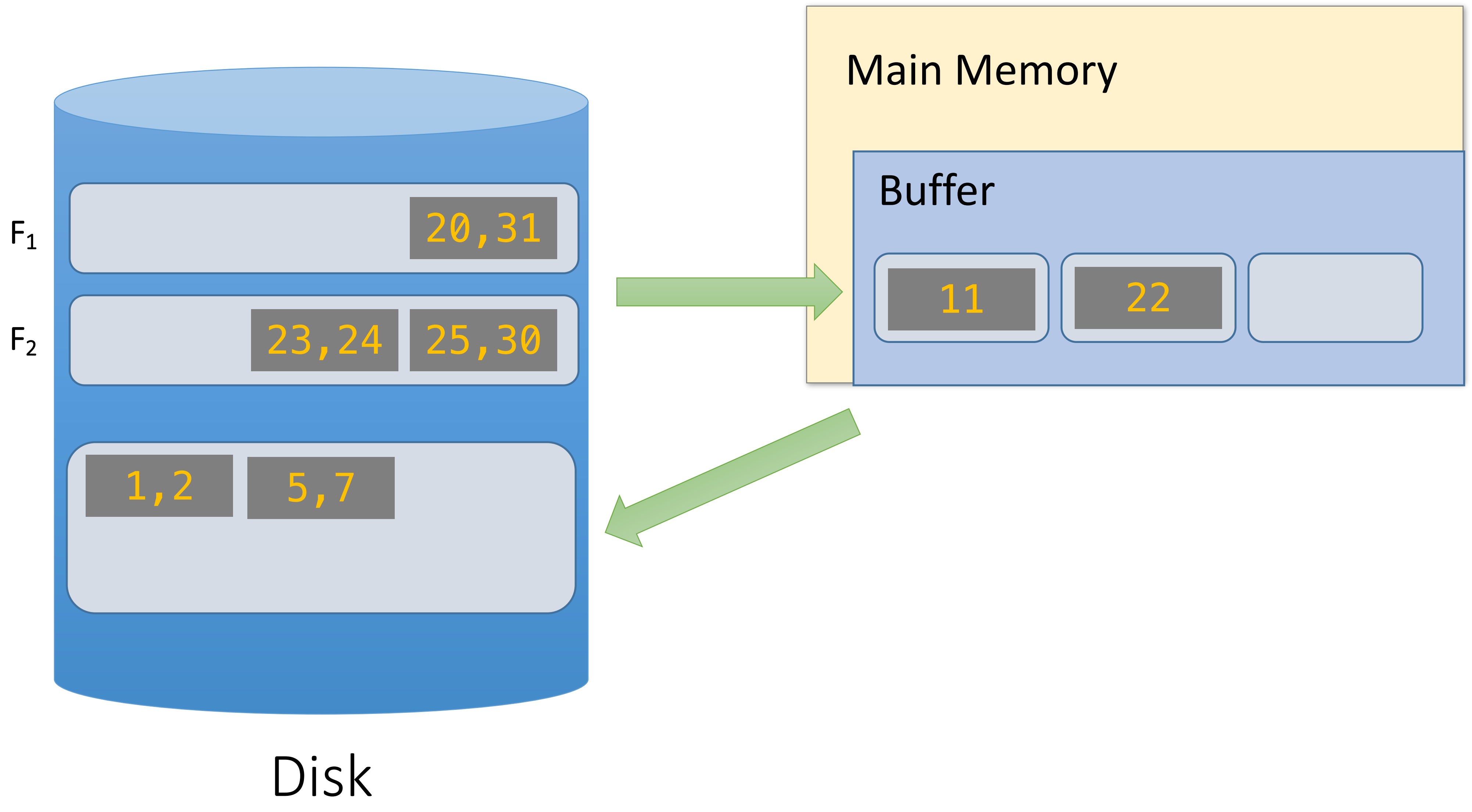
Output:
One *merged*
sorted file



External Merge Algorithm

Input:
Two sorted files

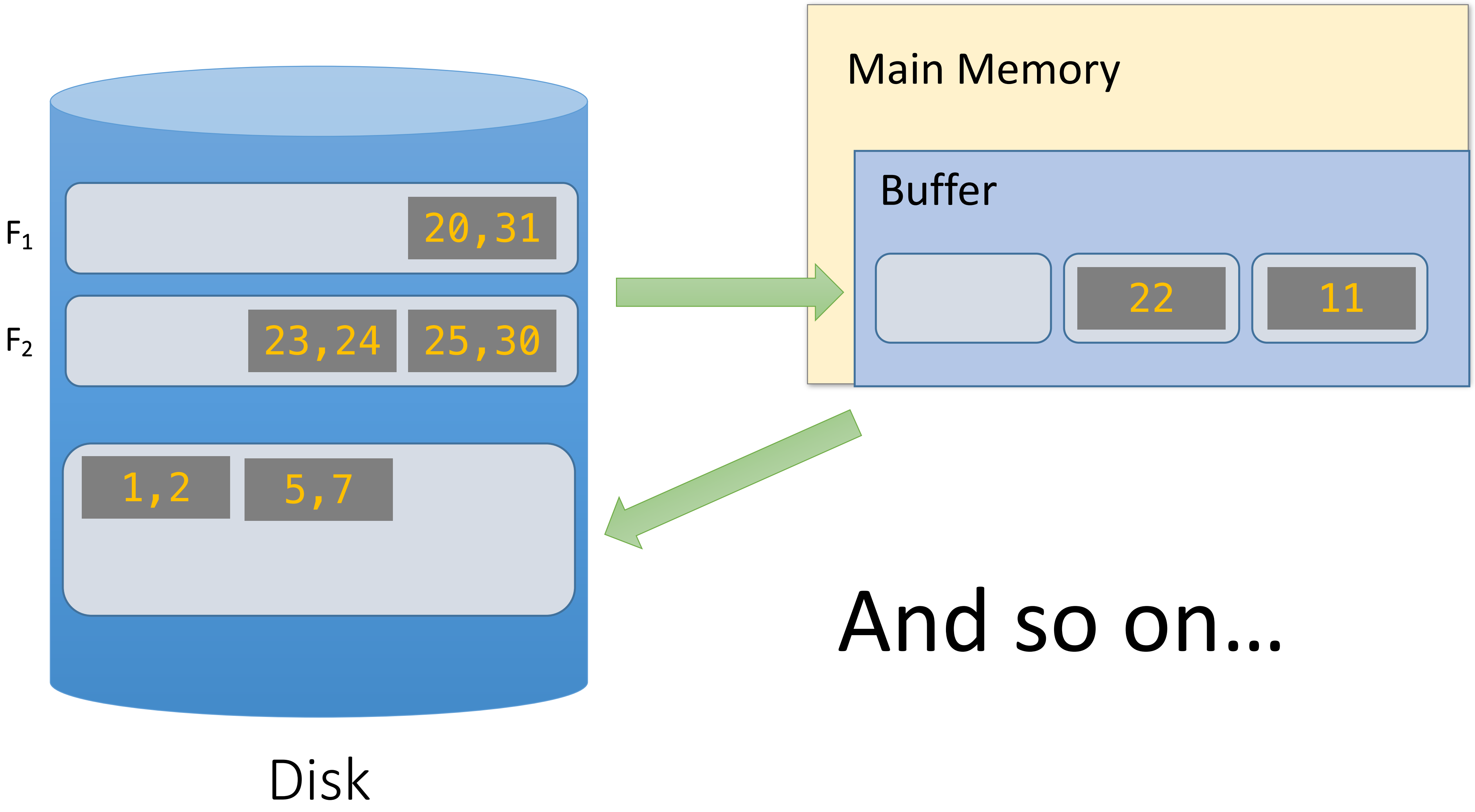
Output:
One *merged* sorted file



External Merge Algorithm

Input:
Two sorted
files

Output:
One *merged*
sorted file



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.

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

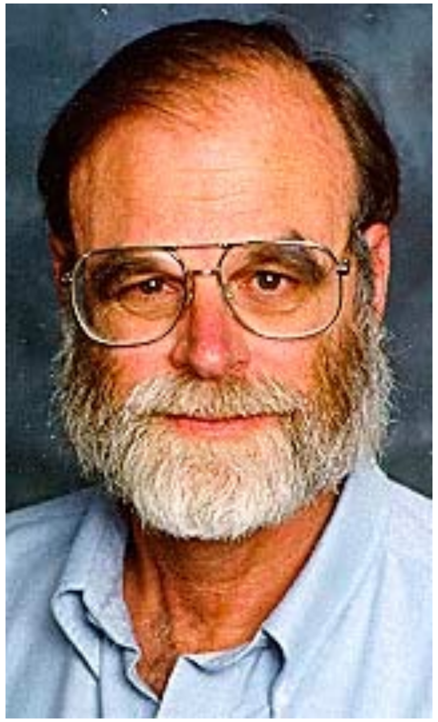
- Sorting is first step in *bulk loading* B+ tree index.

- *Sort-merge* join algorithm involves sorting

Next 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*

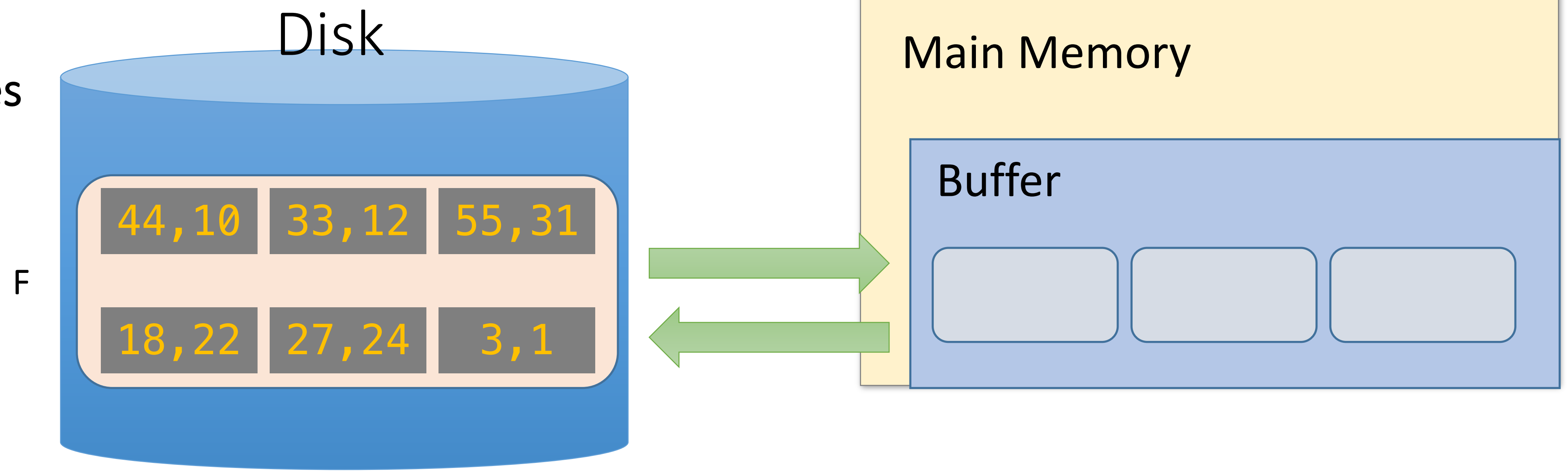
3. **Keep merging** the resulting runs *(each time = a "pass")* until left with one sorted file!

External Merge Sort Algorithm

Example:

- 3 Buffer pages
- 6-page file

Orange file = unsorted



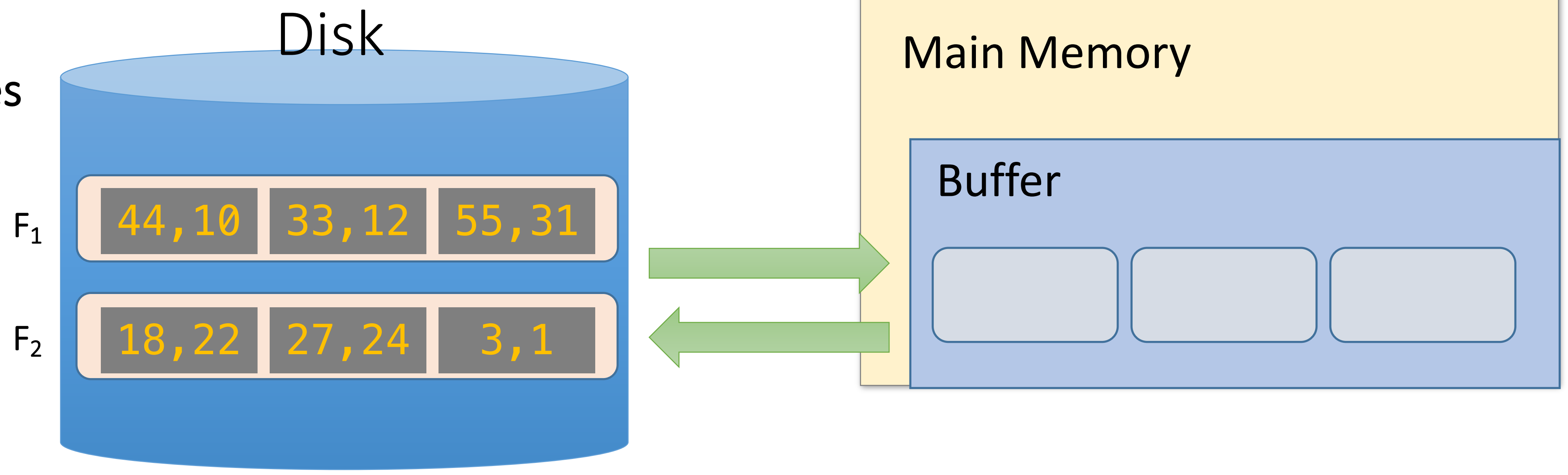
1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm

Example:

- 3 Buffer pages
- 6-page file

Orange file = unsorted



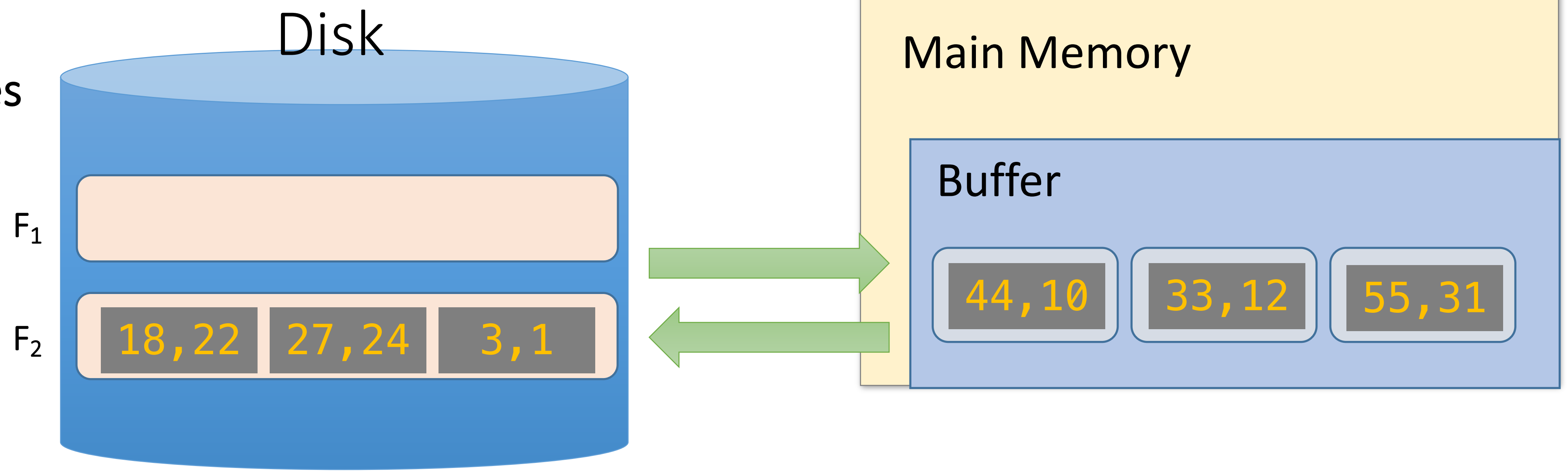
1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm

Example:

- 3 Buffer pages
- 6-page file

Orange file = unsorted



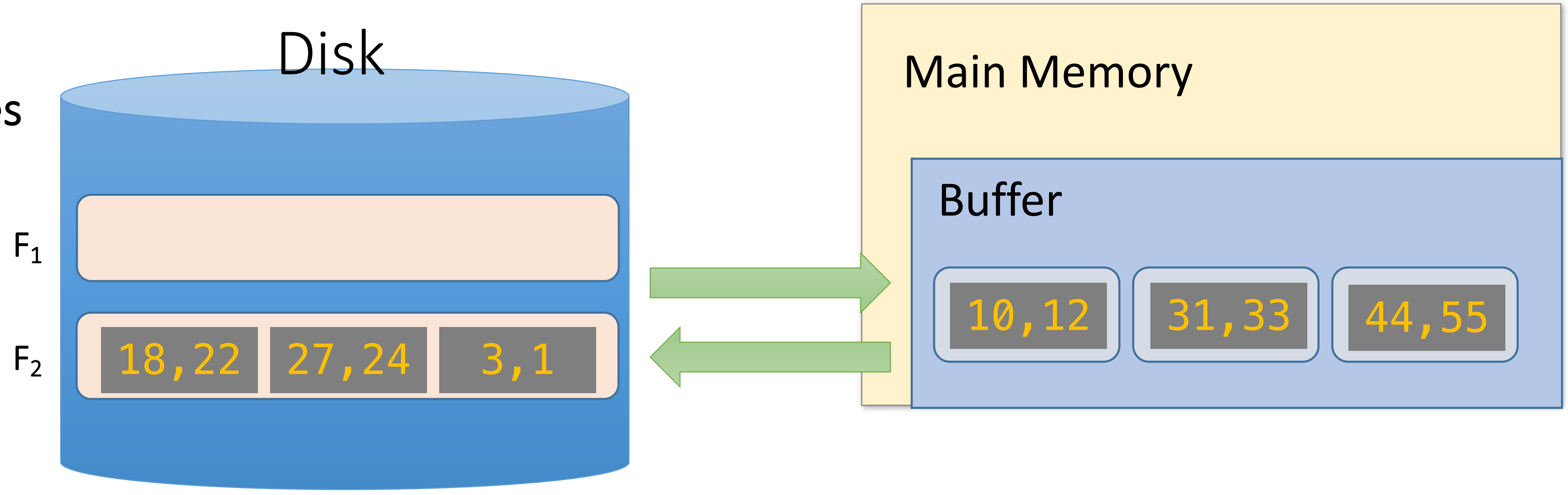
1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm

Example:

- 3 Buffer pages
- 6-page file

Orange file = unsorted

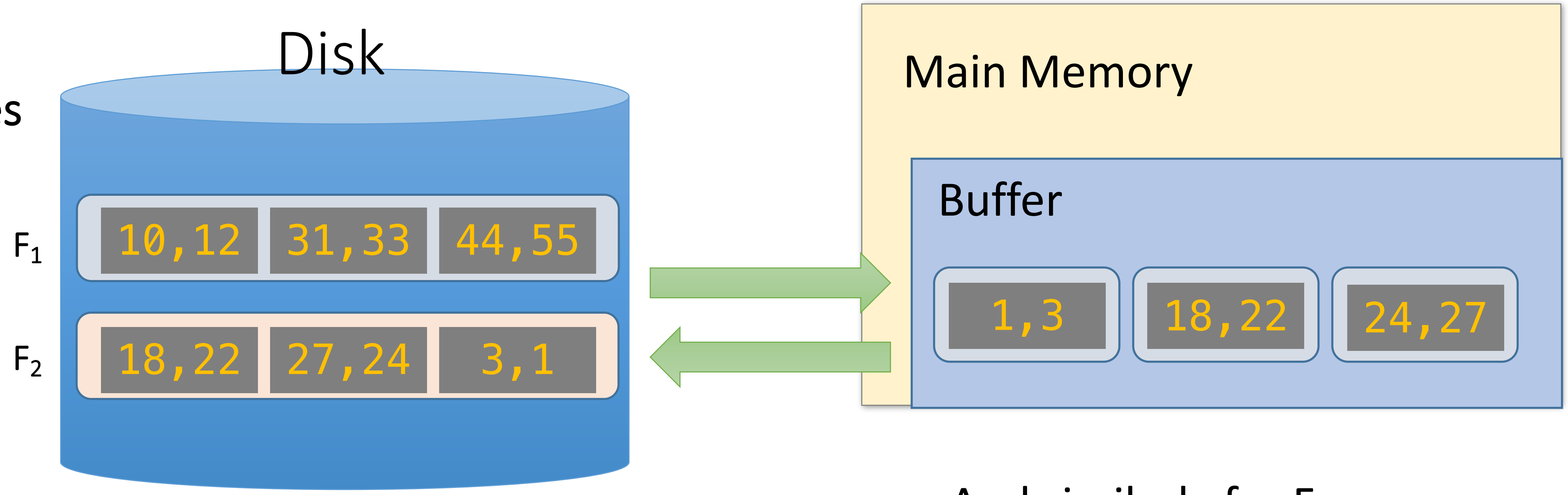


1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm

Example:
- 3 Buffer pages
- 6-page file

Each sorted file is a called a *run*



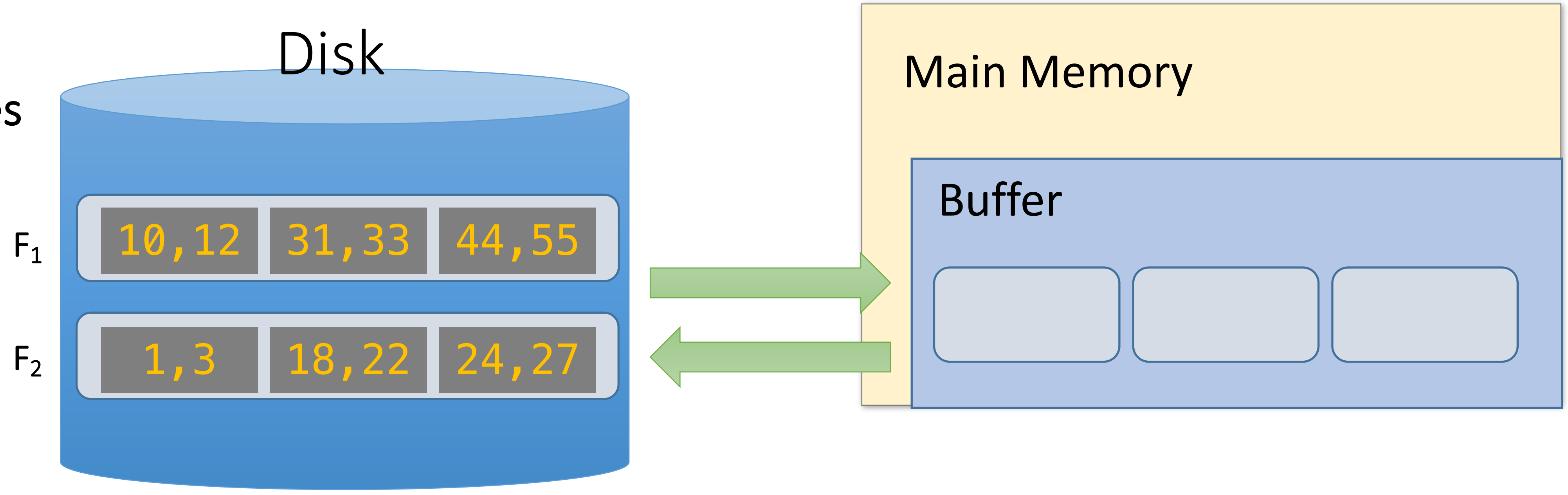
And similarly for F_2

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

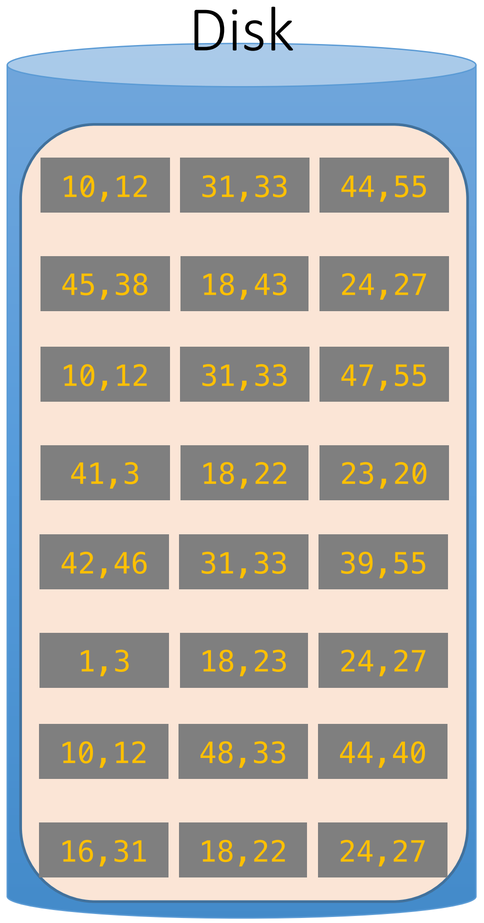
For 3 buffer pages, 6 page file:

1. Split into two 3-page files 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

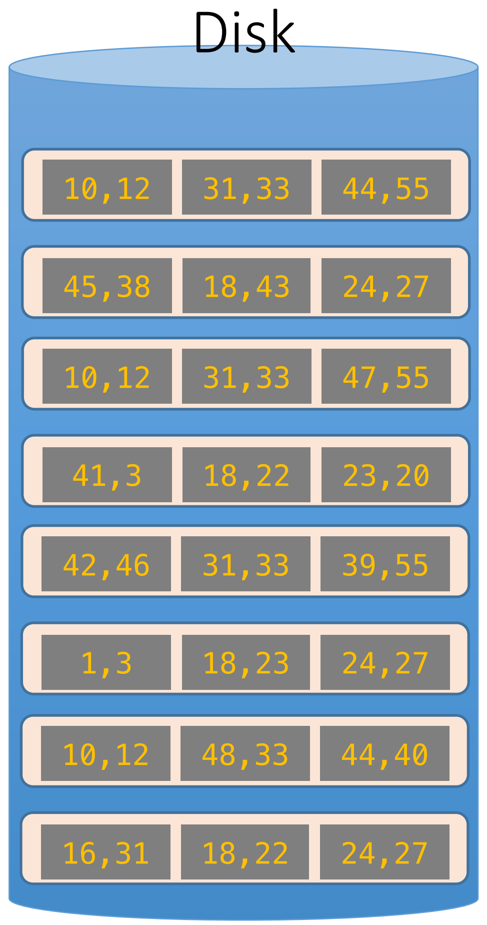
3. Total cost = 24 IO

Running External Merge Sort on Larger Files



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

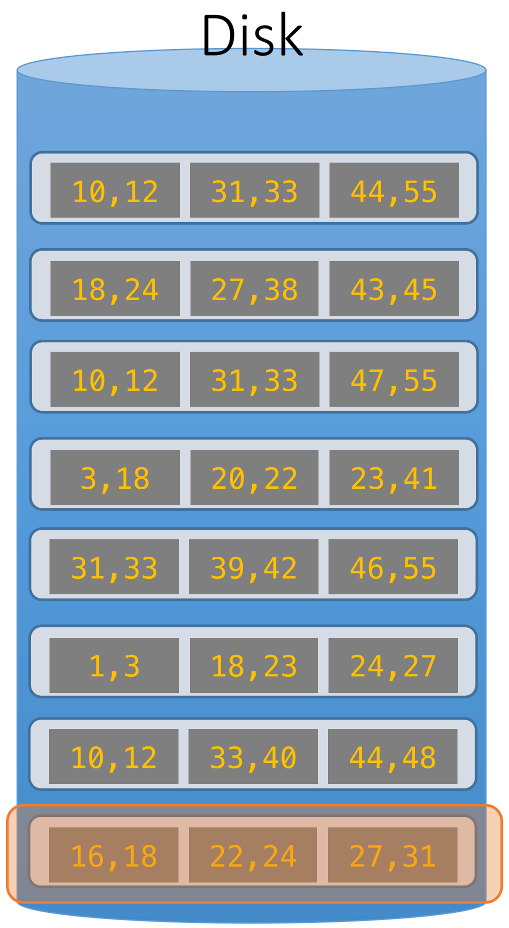
Running External Merge Sort on Larger Files



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

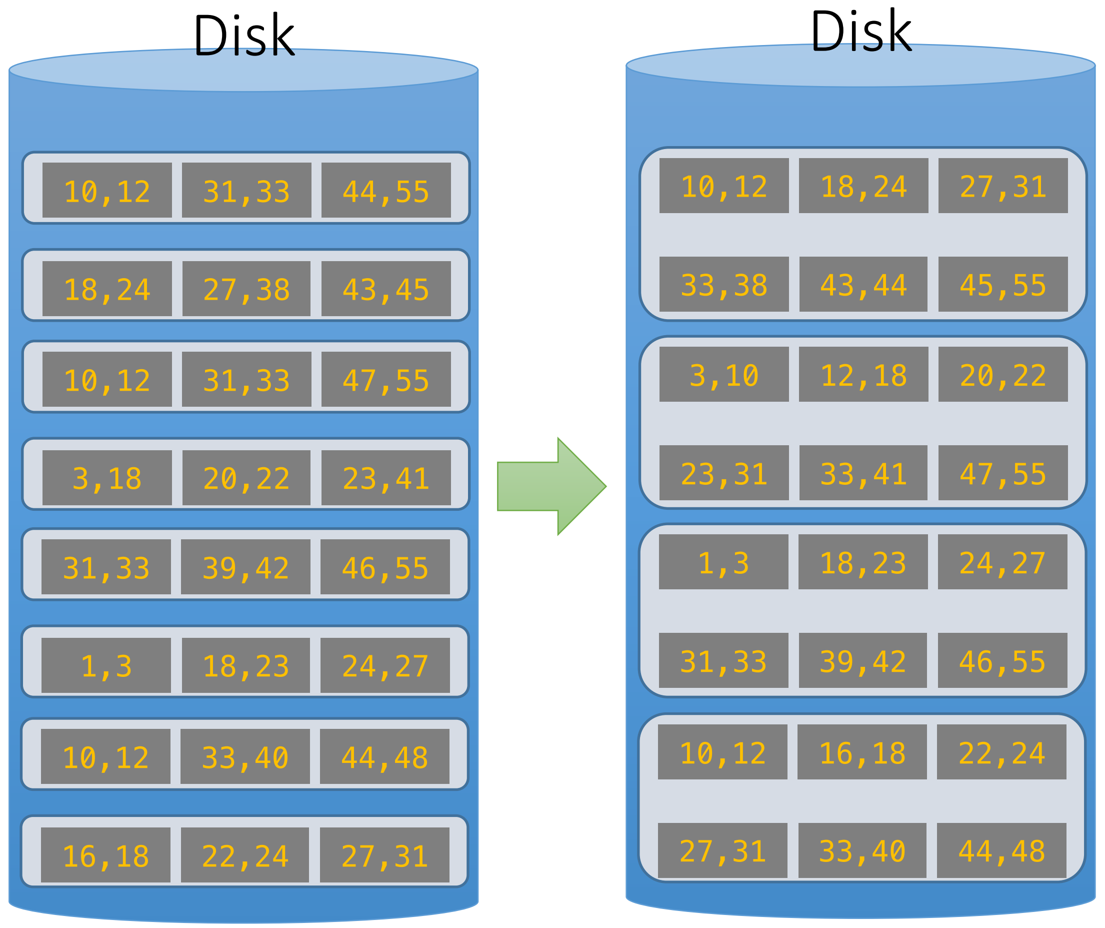


1. Split into files small enough to sort in buffer…

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

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

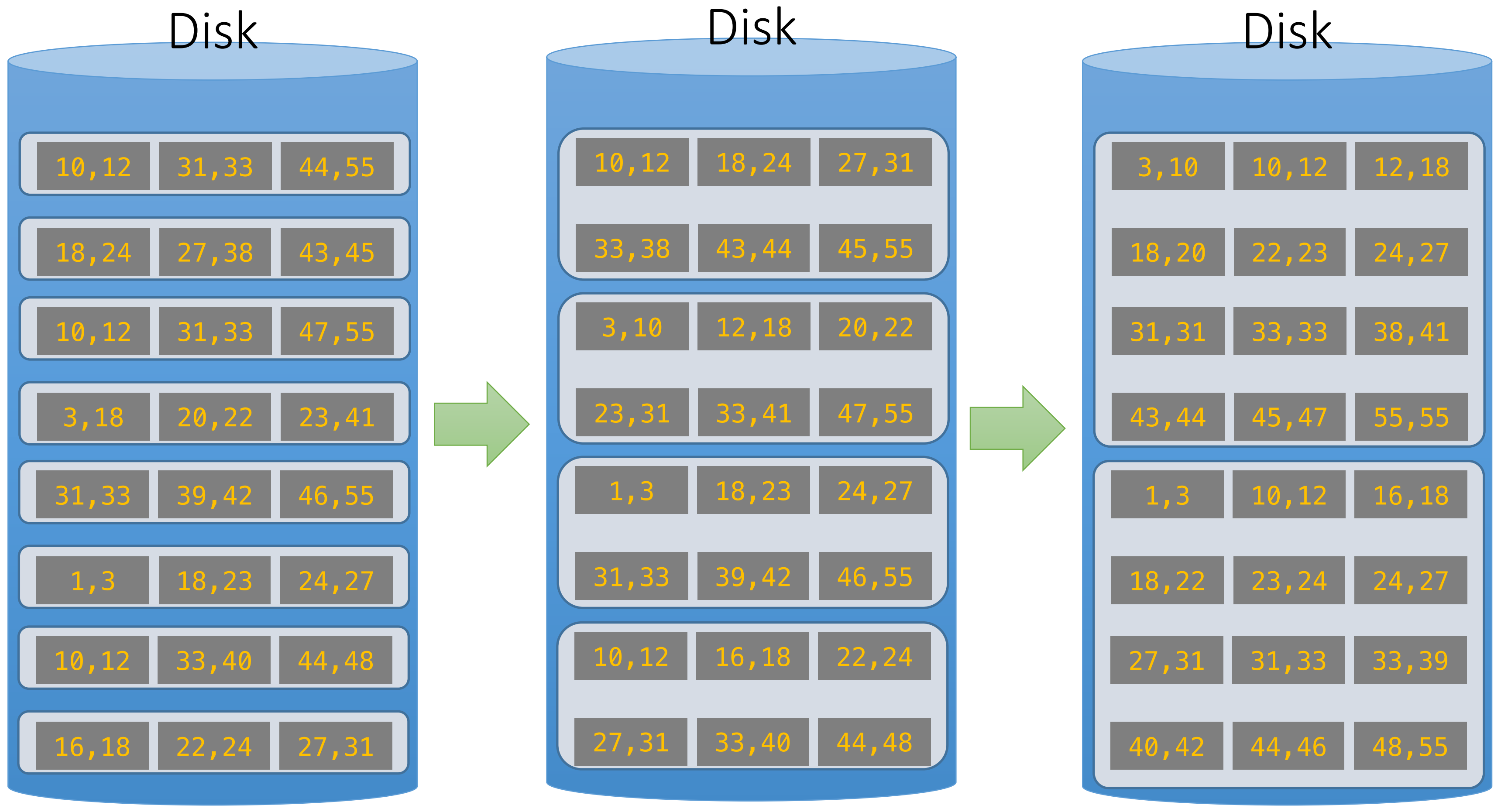
Running External Merge Sort on Larger Files



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

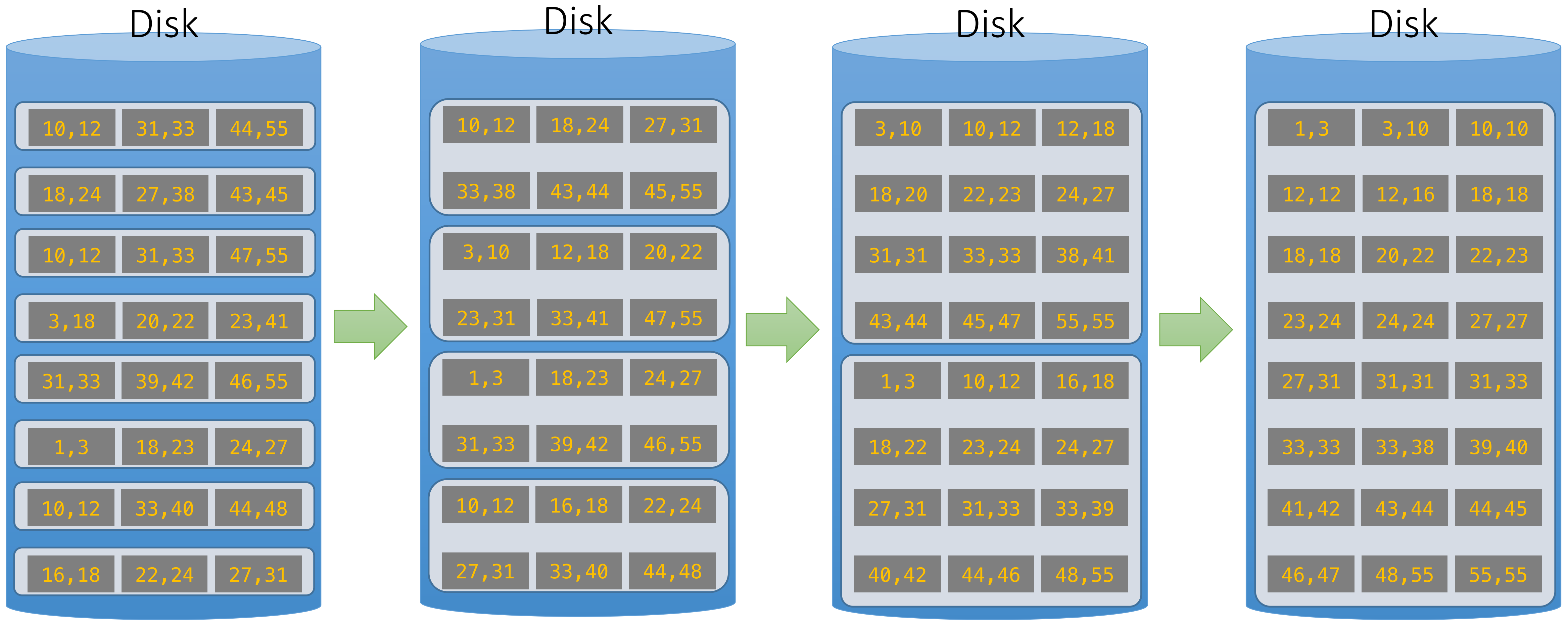


Assume we still only have *3* buffer pages *(Buffer not pictured)*

3. And repeat…

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

Running External Merge Sort on Larger Files

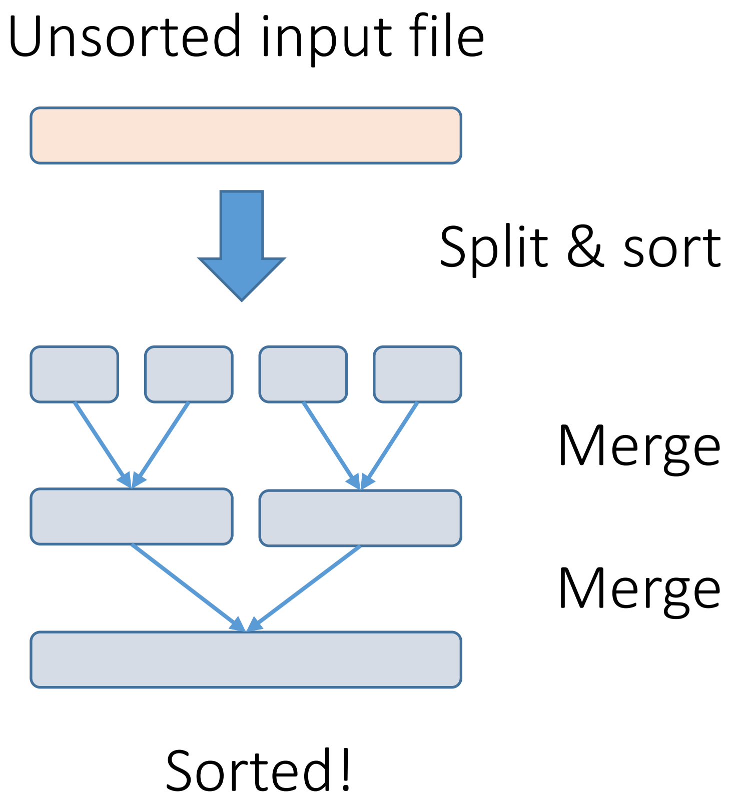


4. And repeat!

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 $\lceil \log_2 N \rceil$ 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!

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 (no covered)

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>

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

Starting with runs of length 1

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

Performing ***B***-way merges

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