L18: The I/O model and External Sort

CS3200 Database design (sp18 s2)

https://course.ccs.neu.edu/cs3200sp18s2/

3/22/2018

Announcements!

- If I did not know your name during exam2: come with your name plate, sit in the front, and make sure I notice you ☺
- Worried about your grade, come and contribute
- HW5: collaboration policy
- Office Hours (not tomorrow unless you raise your hand now)
- Interested in becoming a TA for cs3200 in a future semester
- Outline today
  - Sorting
  - Indexing

External Merge Algorithm

- Input: 2 sorted lists of length m and n
- Output: 1 sorted list of length m + n
- Required: At least … (?) Buffer Pages
- IOs: … (?)

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:

$$A_1 \le A_2 \le \cdots \le A_n$$
$$B_1 \le B_2 \le \cdots \le B_m$$

Then:

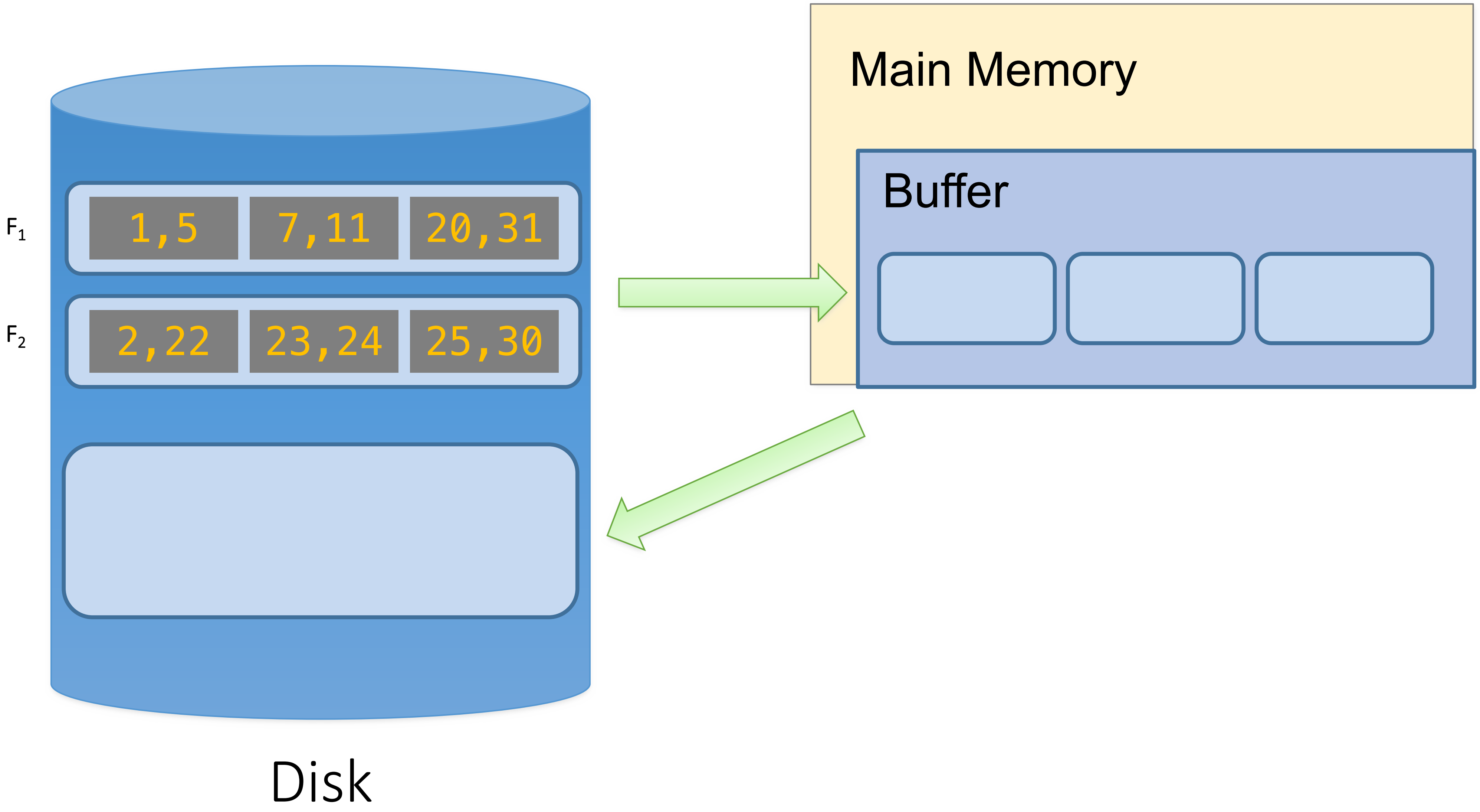
$$Min(A_1, B_1) \le A_i$$
$$Min(A_1, B_1) \le B_j$$

for i=1….n and j=1….m

External Merge Algorithm

Input:
Two sorted files

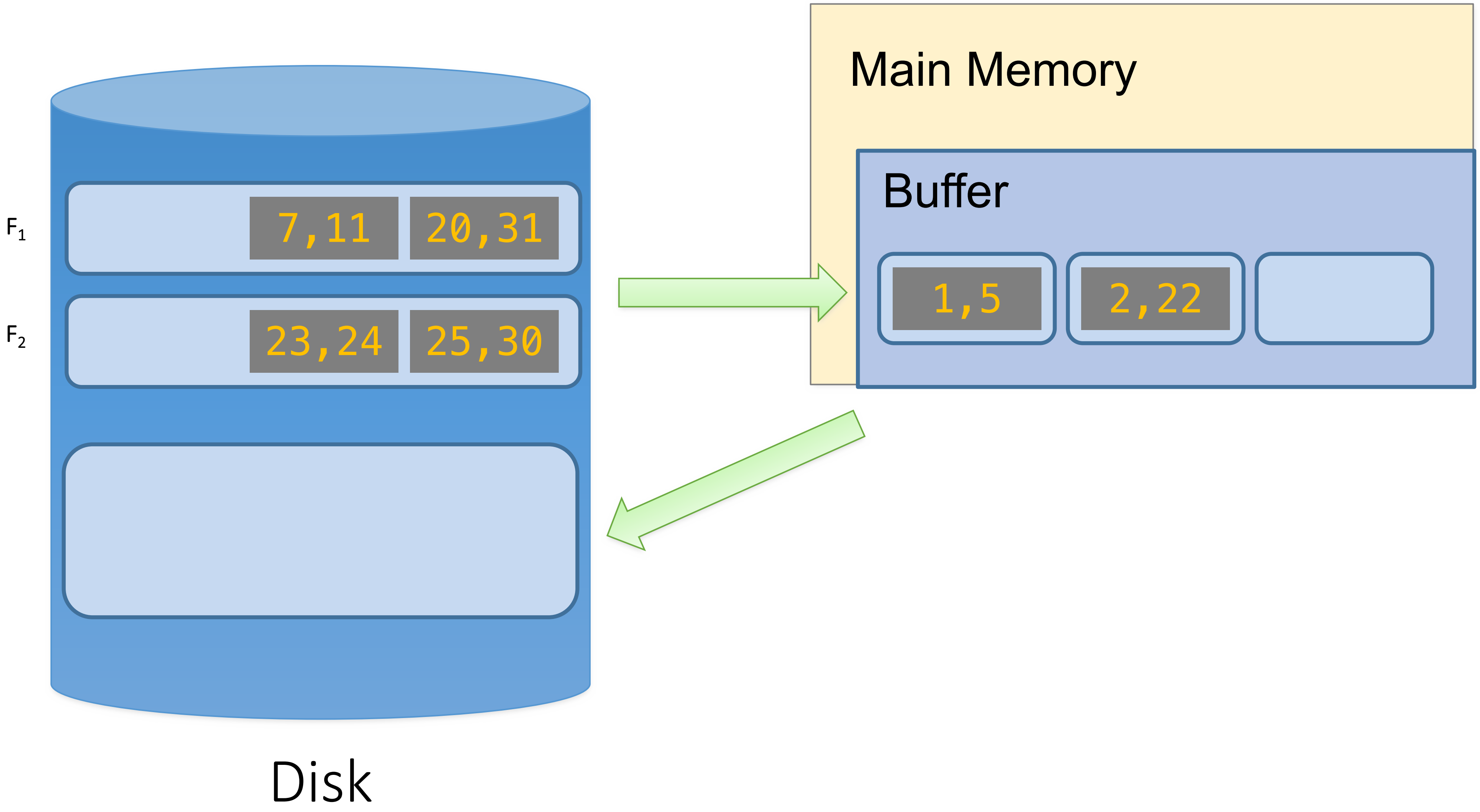
Output:
One *merged* sorted file



External Merge Algorithm

Input:
Two sorted
files

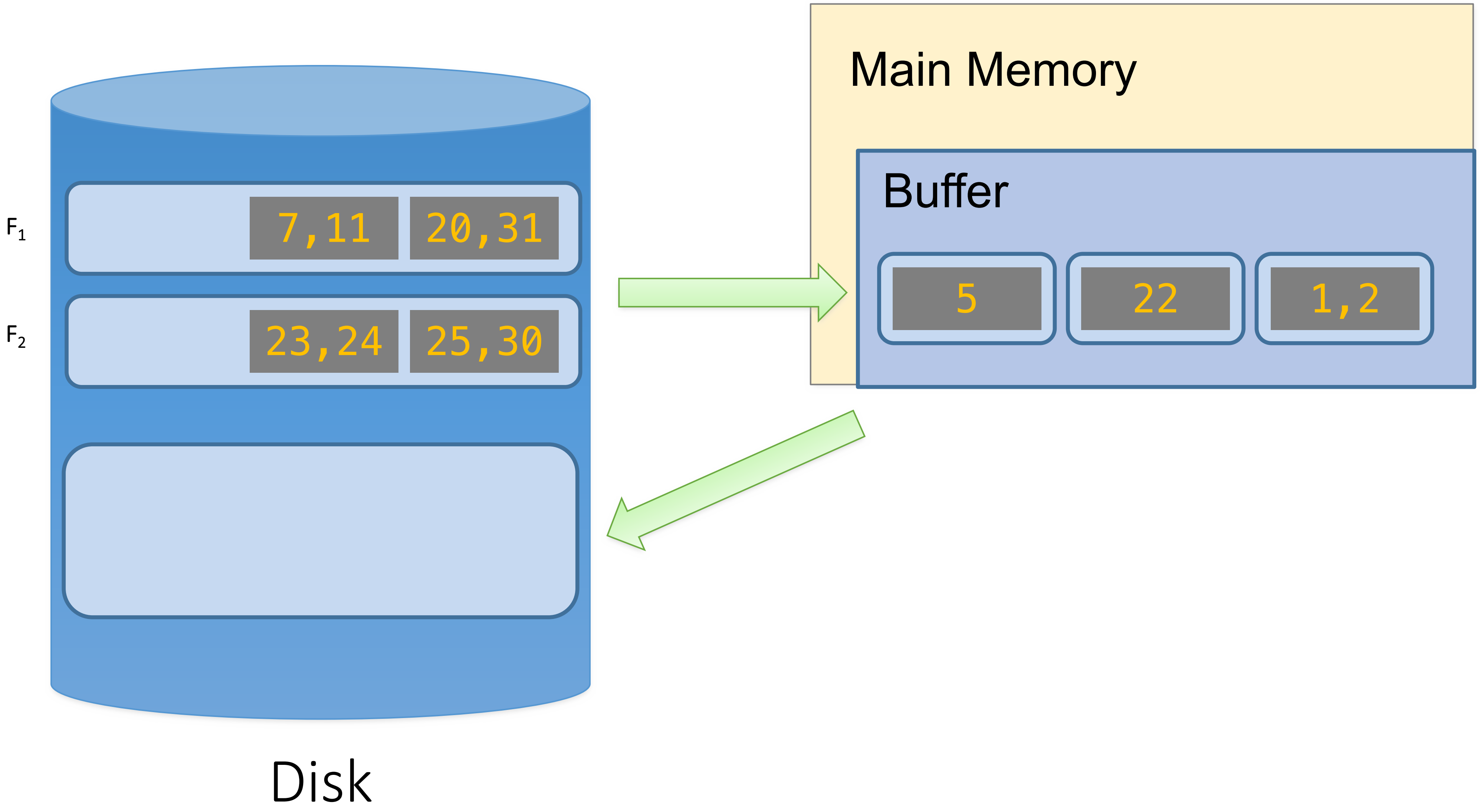
Output:
One *merged*
sorted file



External Merge Algorithm

Input:
Two sorted
files

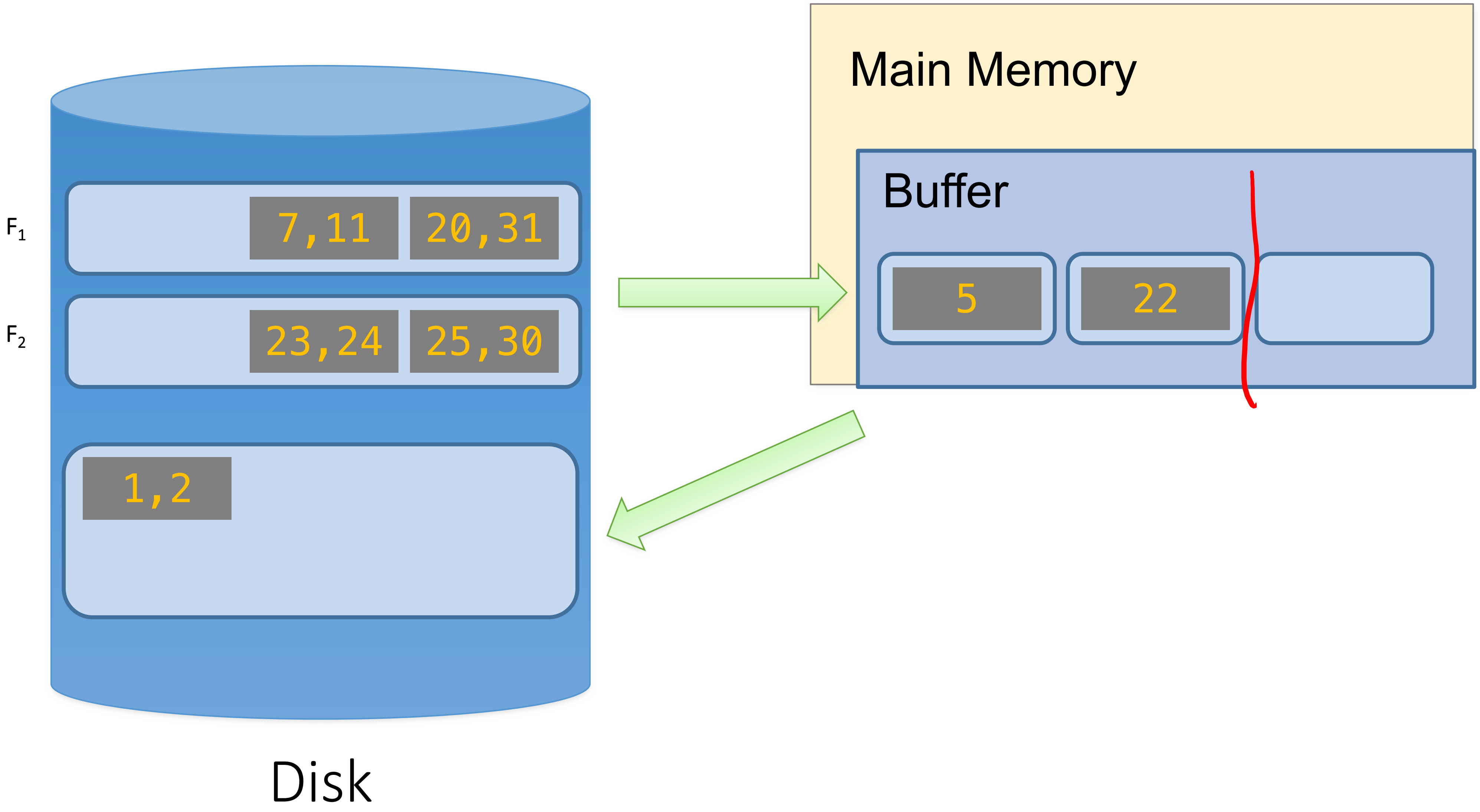
Output:
One *merged*
sorted file



External Merge Algorithm

Input:
Two sorted
files

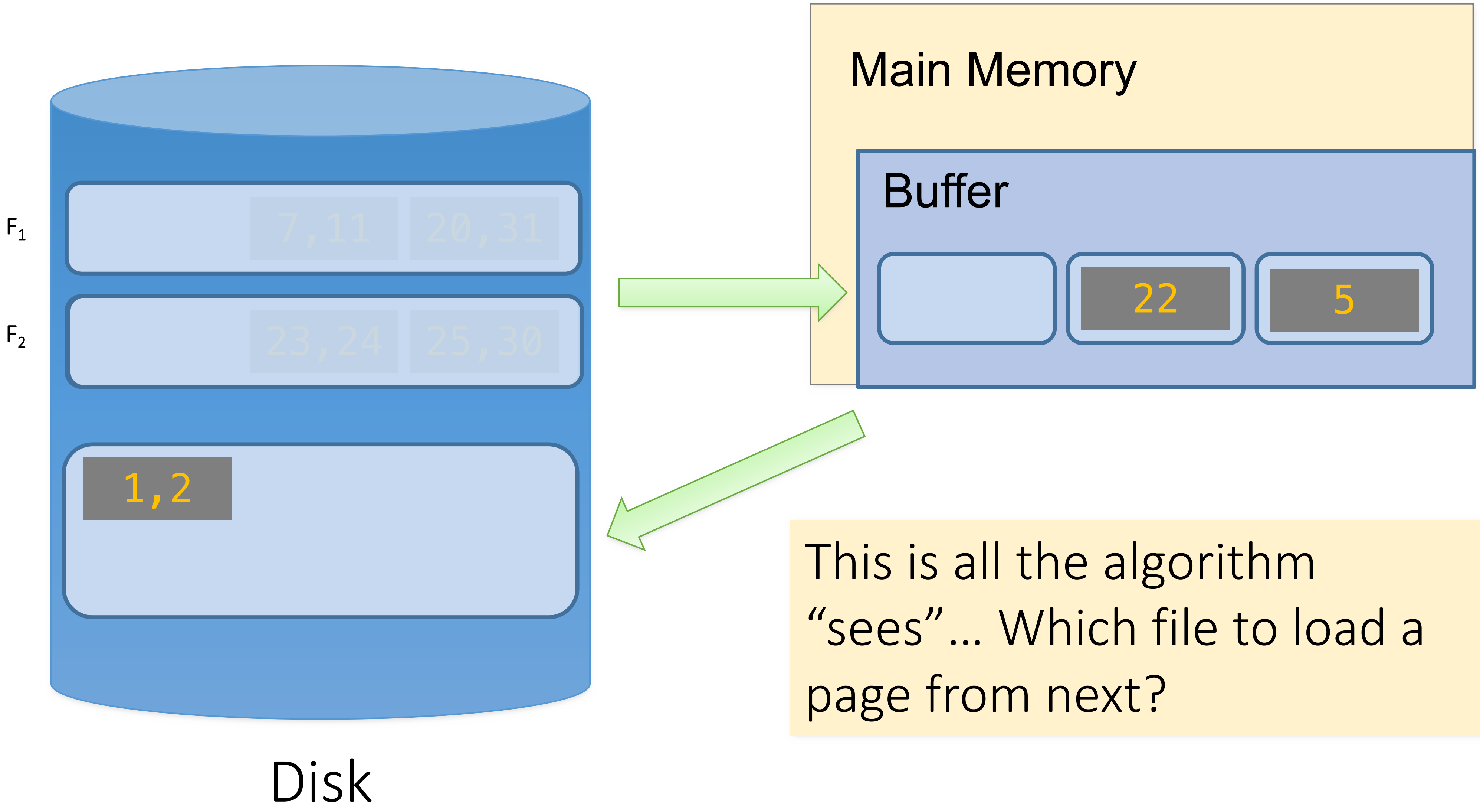
Output:
One *merged*
sorted file



External Merge Algorithm

Input:
Two sorted files

Output:
One *merged* sorted file

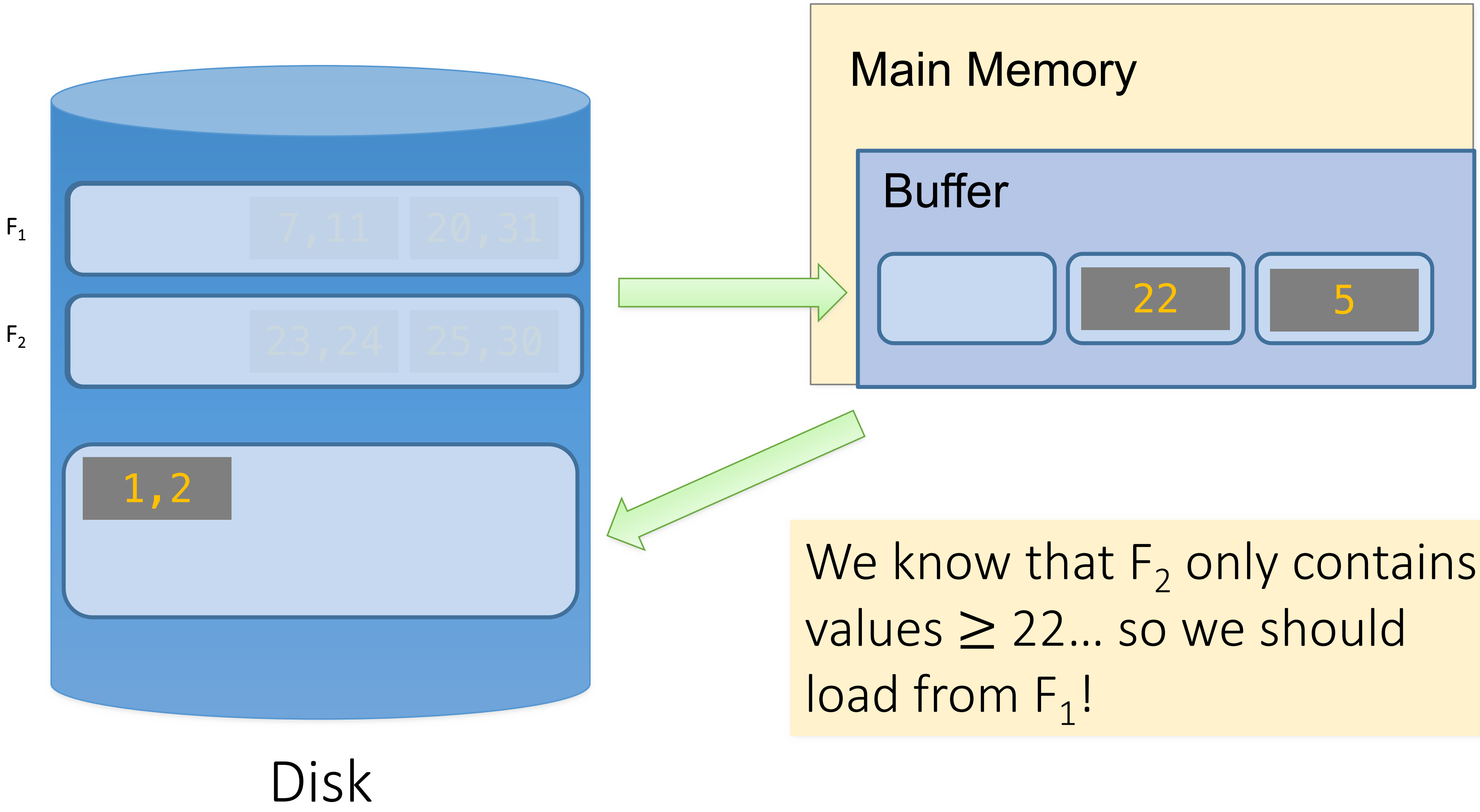


This is all the algorithm "sees"… Which file to load a page from next?

External Merge Algorithm

Input:
Two sorted files

Output:
One *merged* sorted file

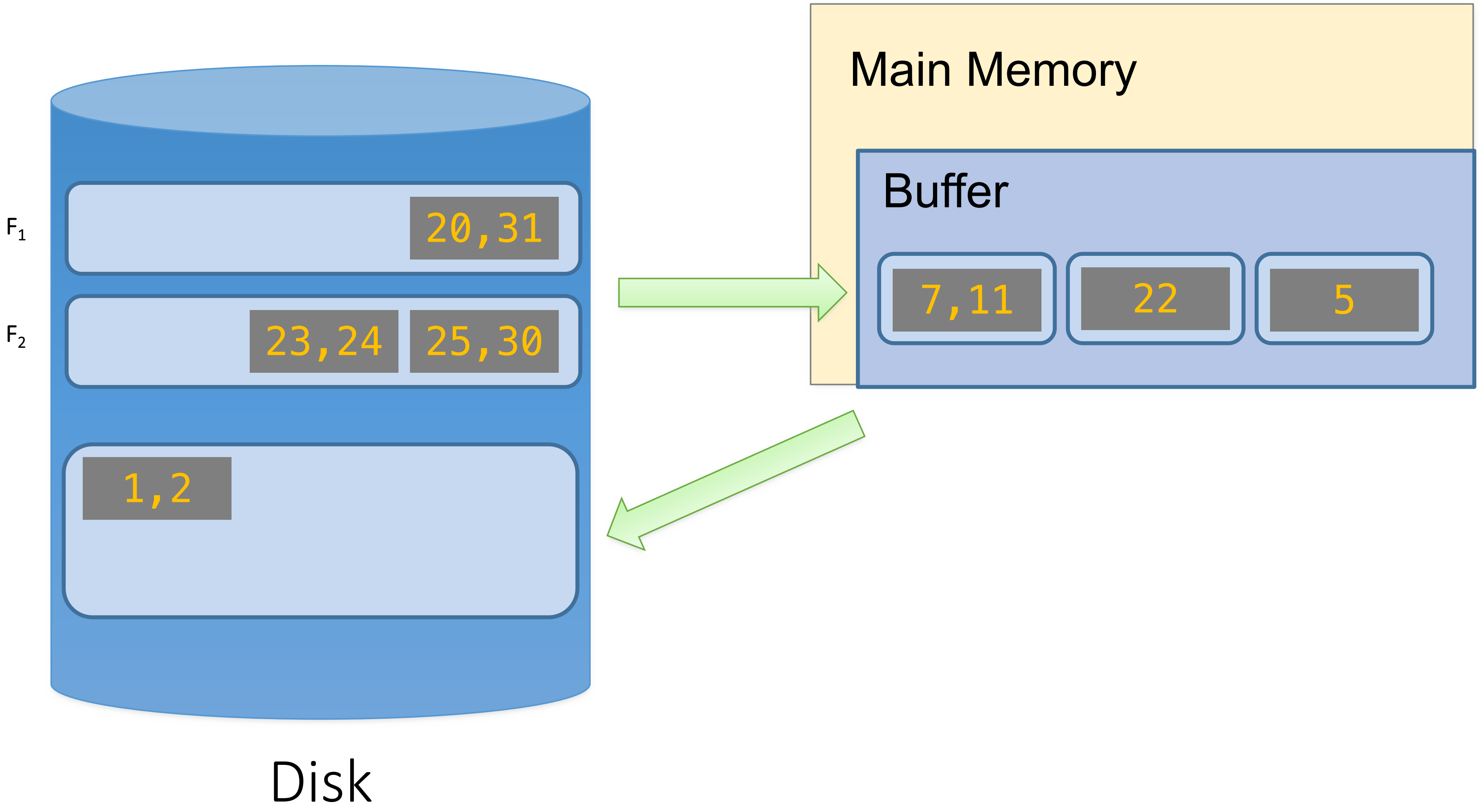


We know that F_2 only contains values ≥ 22… so we should load from F_1!

External Merge Algorithm

Input:
Two sorted files

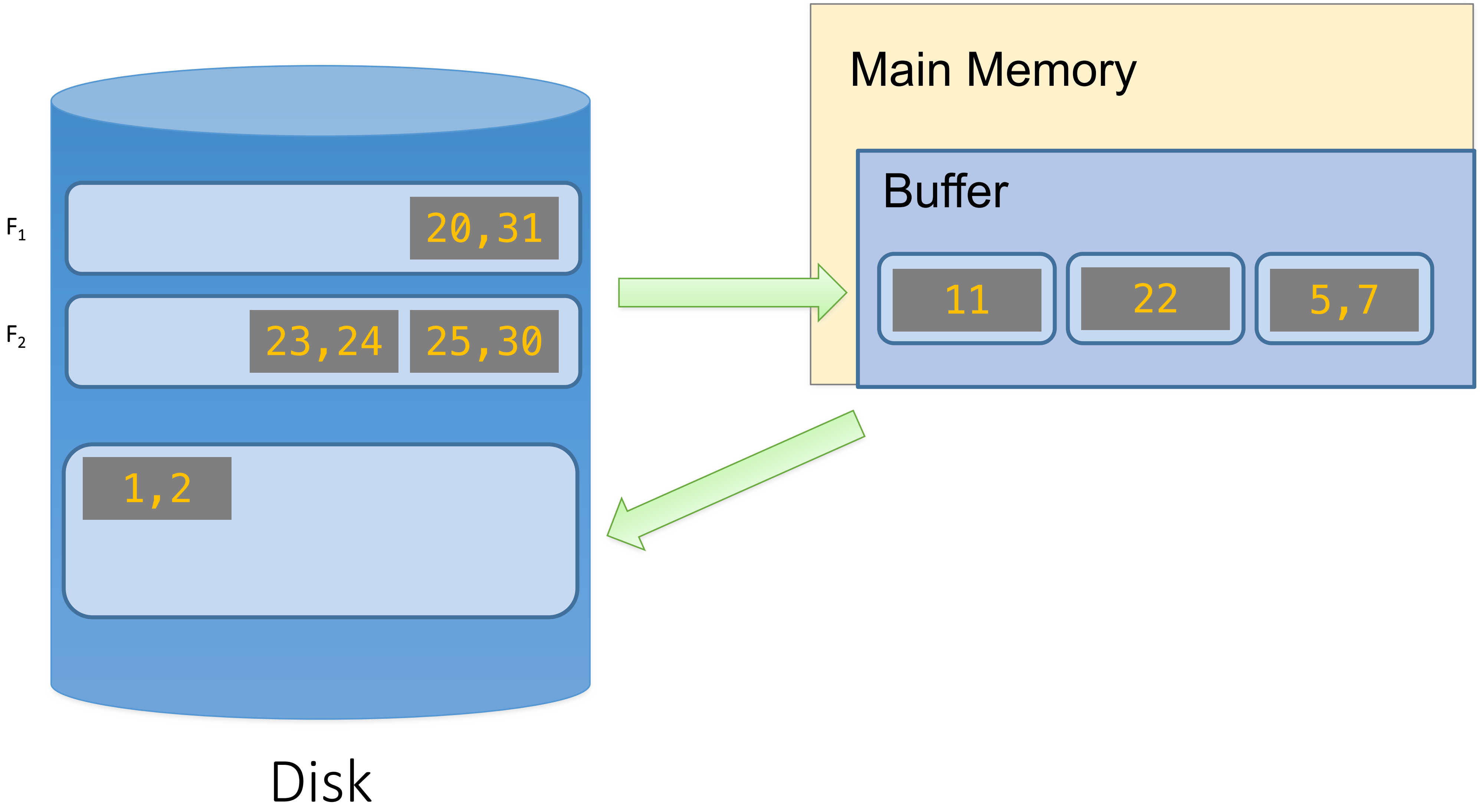
Output:
One *merged* sorted file



External Merge Algorithm

Input:
Two sorted
files

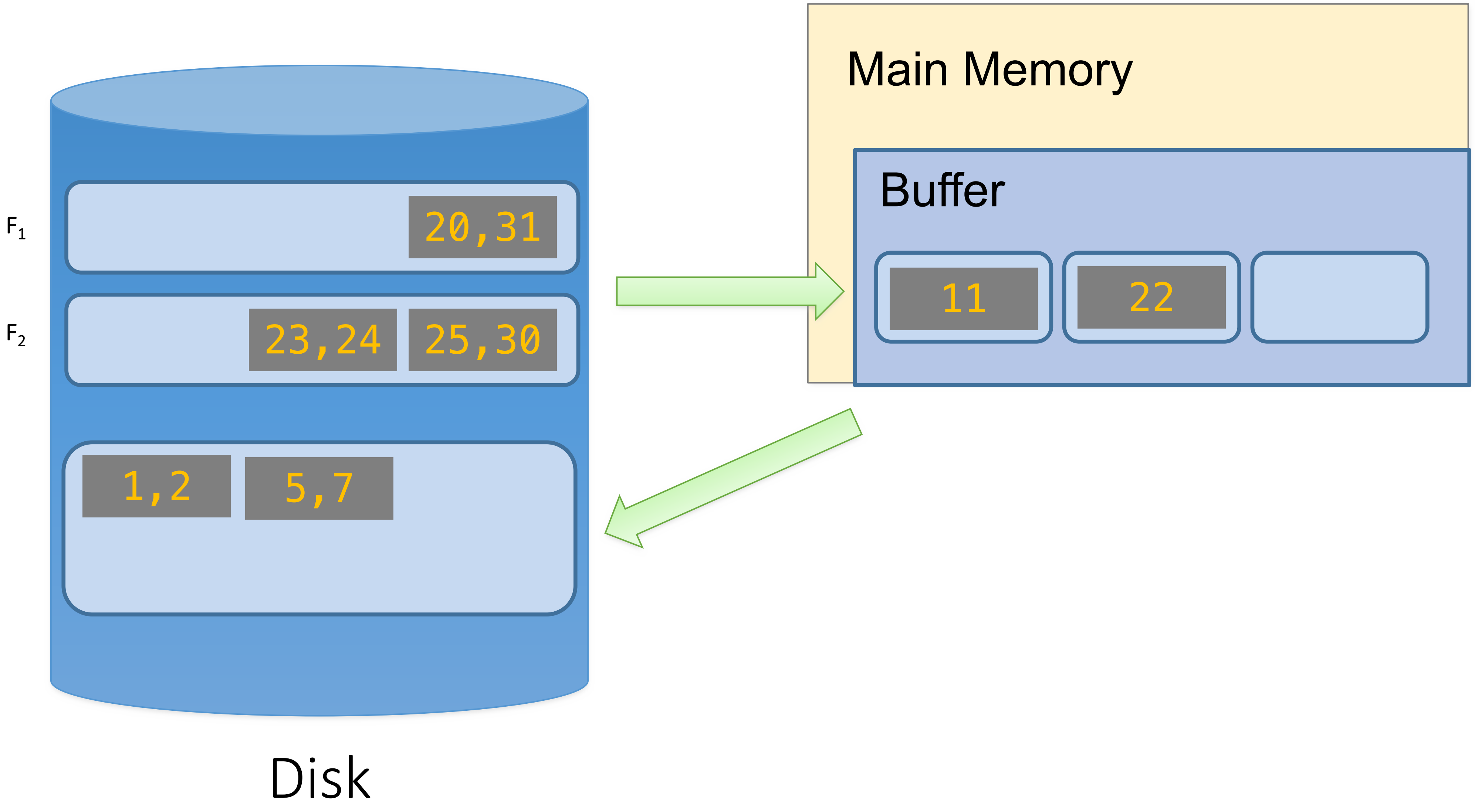
Output:
One *merged*
sorted file



External Merge Algorithm

Input:
Two sorted
files

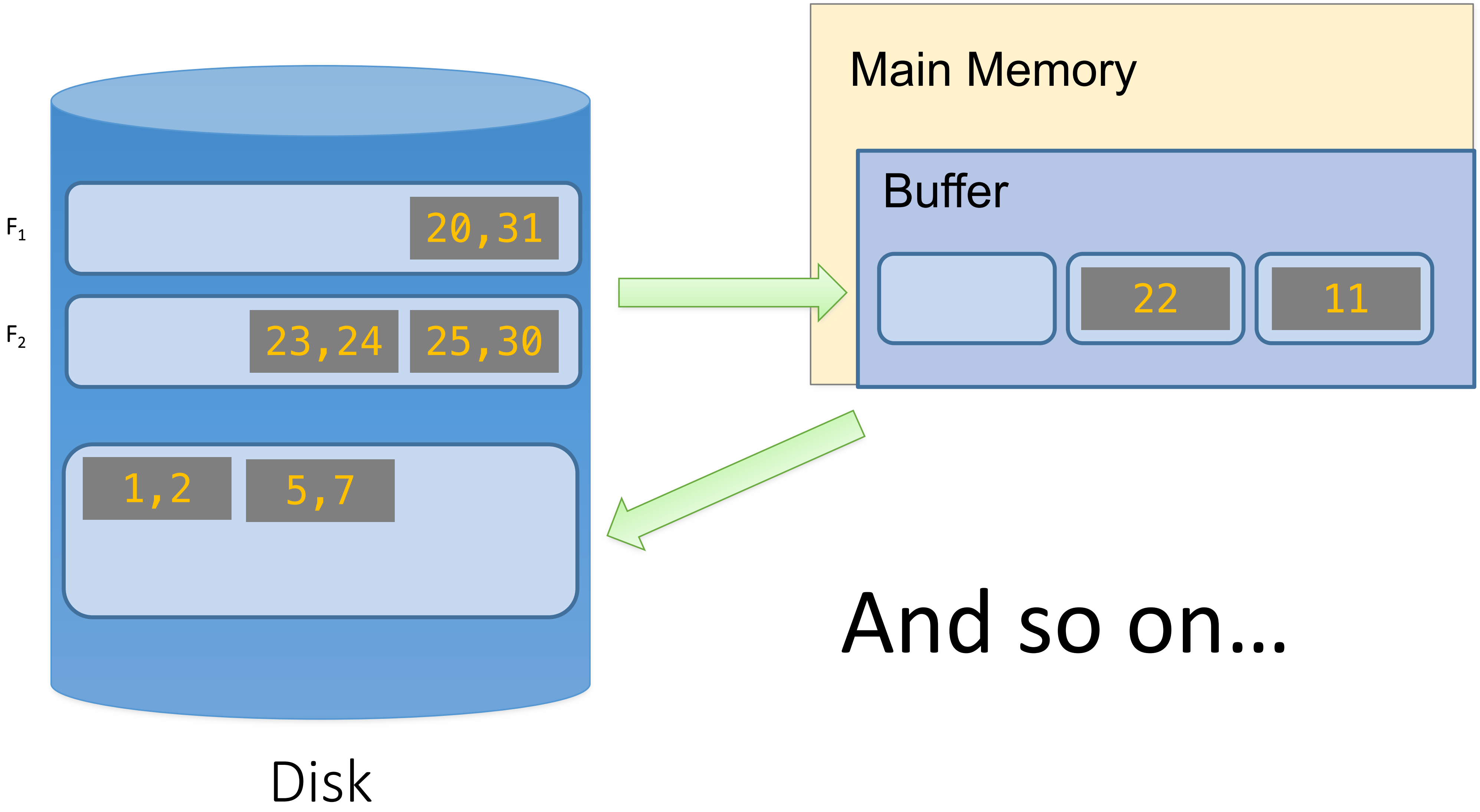
Output:
One *merged*
sorted file



External Merge Algorithm

Input:
Two sorted files

Output:
One *merged* sorted file



And so on…

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

How many buffer pages do we need to merge B lists?

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

3. 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?)

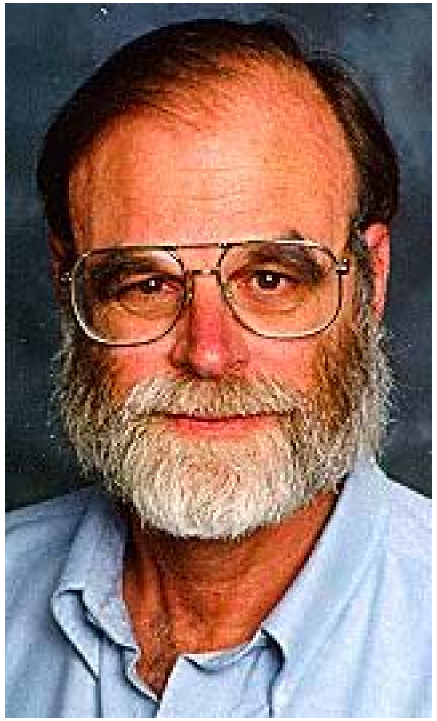
- Sort-merge join algorithm involves sorting

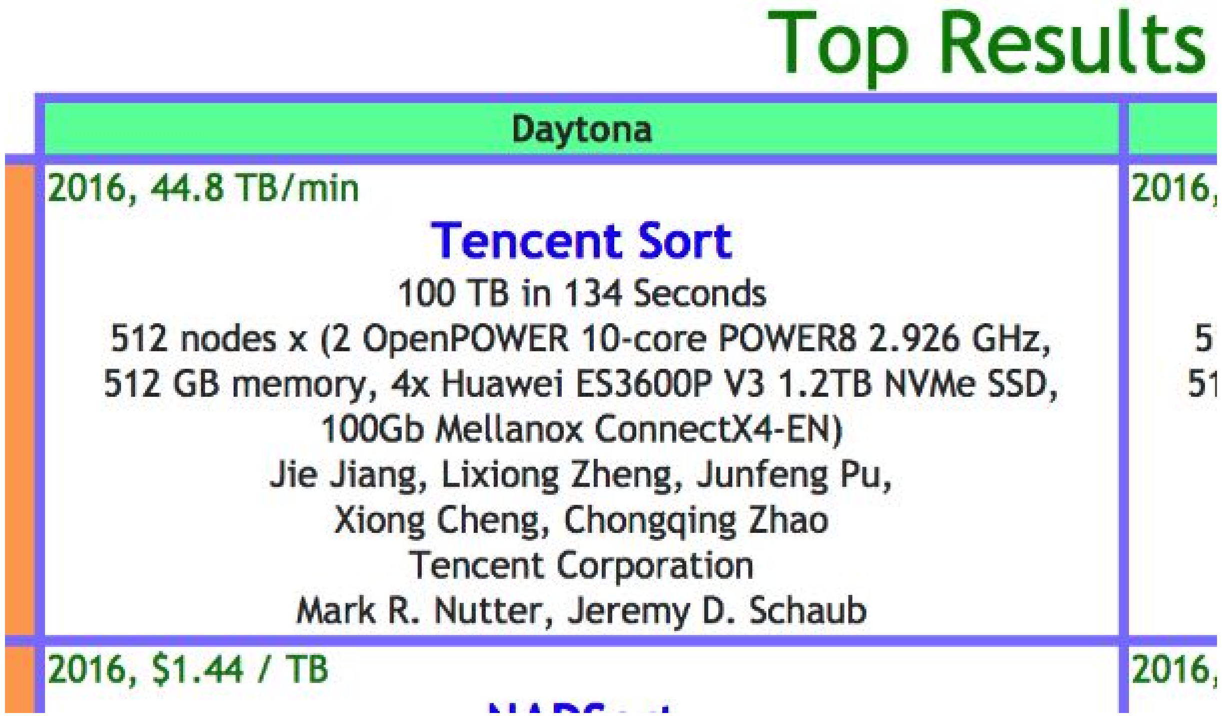
- Sorting is first step in bulk loading B+ tree index.

Next lectures

Do people care?

http://sortbenchmark.org





Sort benchmark bears his name

So how do we sort big files?

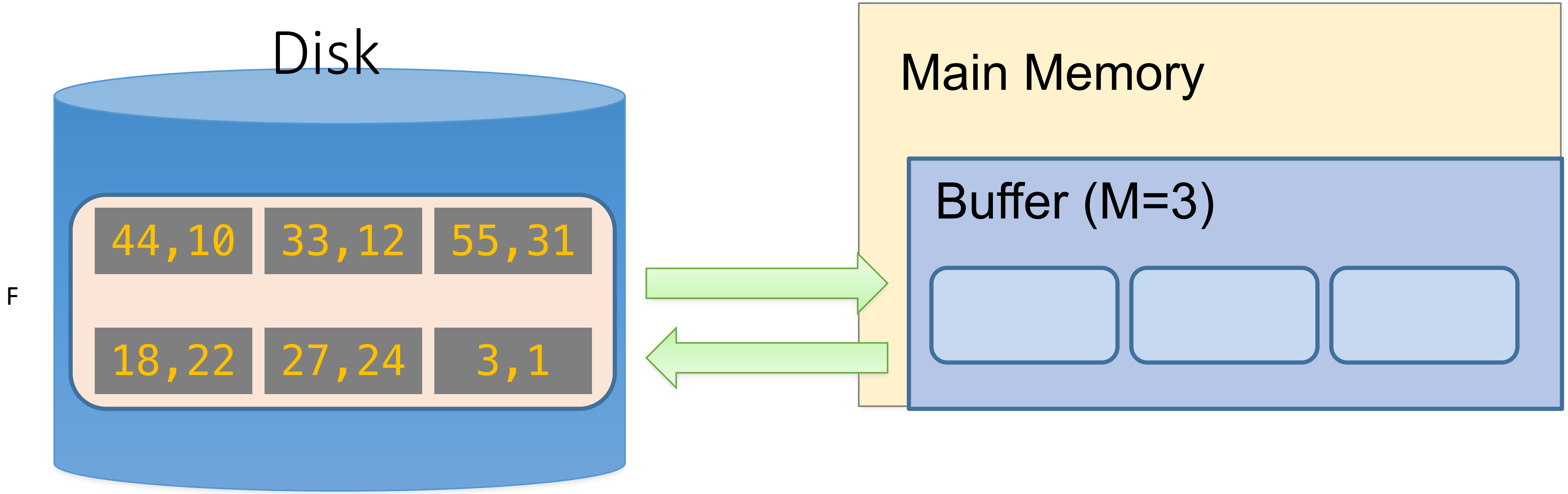
- Split into chunks small enough to sort in memory (“**runs**”)

- Merge pairs (or groups) of runs using the external merge algorithm

- Keep merging the resulting runs (each time = a “**pass**”) until left with one sorted file!

External Merge Sort Algorithm

Example:
- M = 3 Buffer pages
- 6-page file



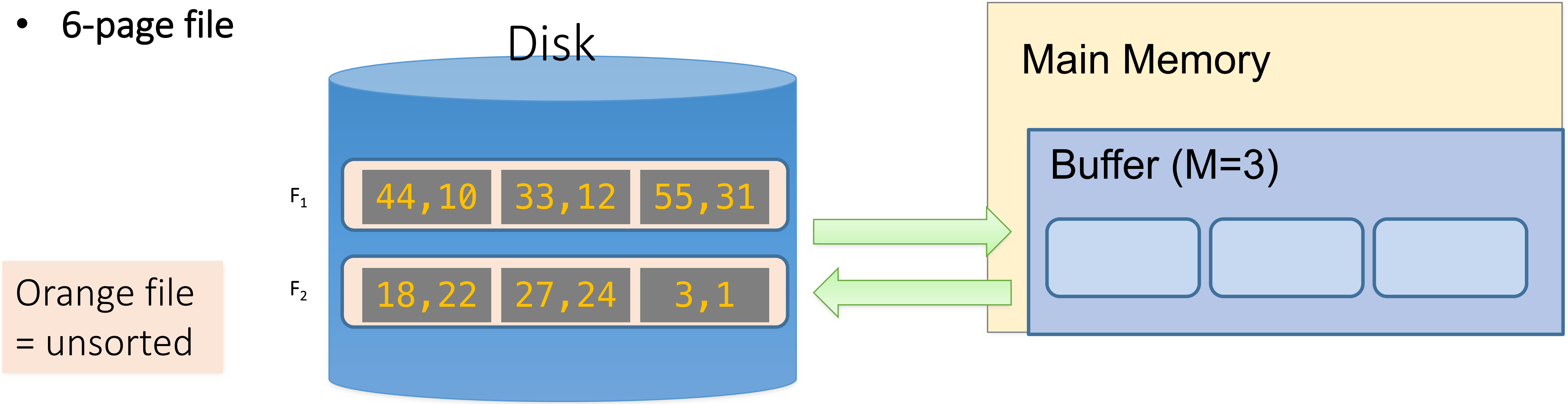
Orange file
= unsorted

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

External Merge Sort Algorithm

Example:

- M = 3 Buffer pages
- 6-page file

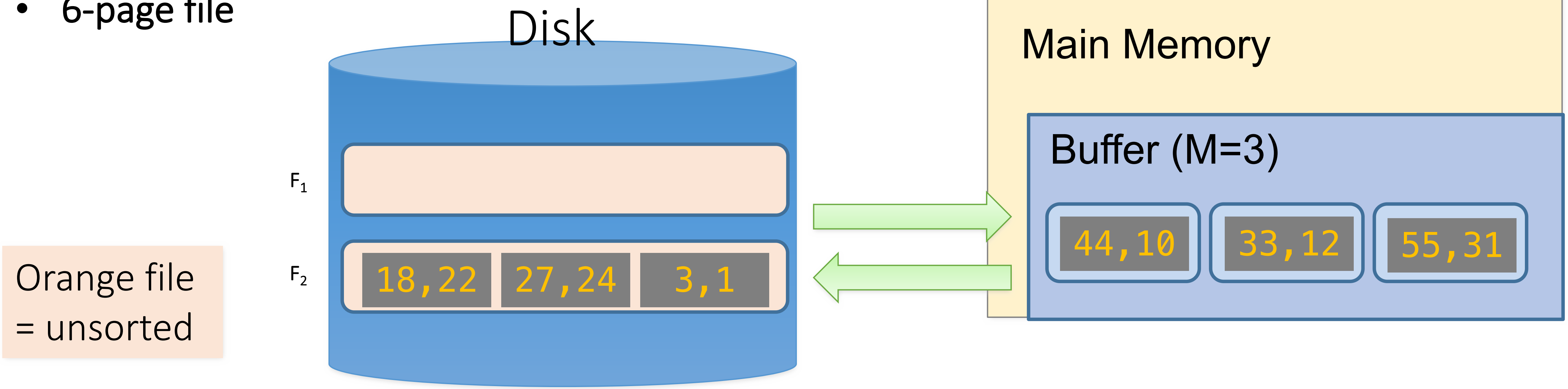


1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm

Example:

- M = 3 Buffer pages
- 6-page file

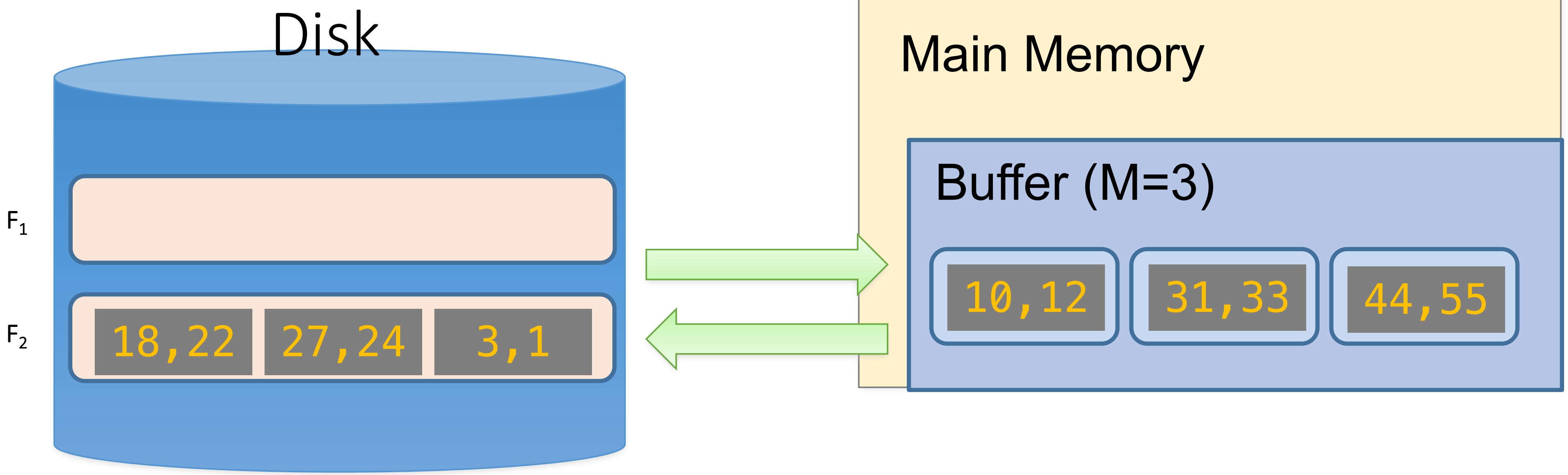


1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm

Example:

- M = 3 Buffer pages
- 6-page file



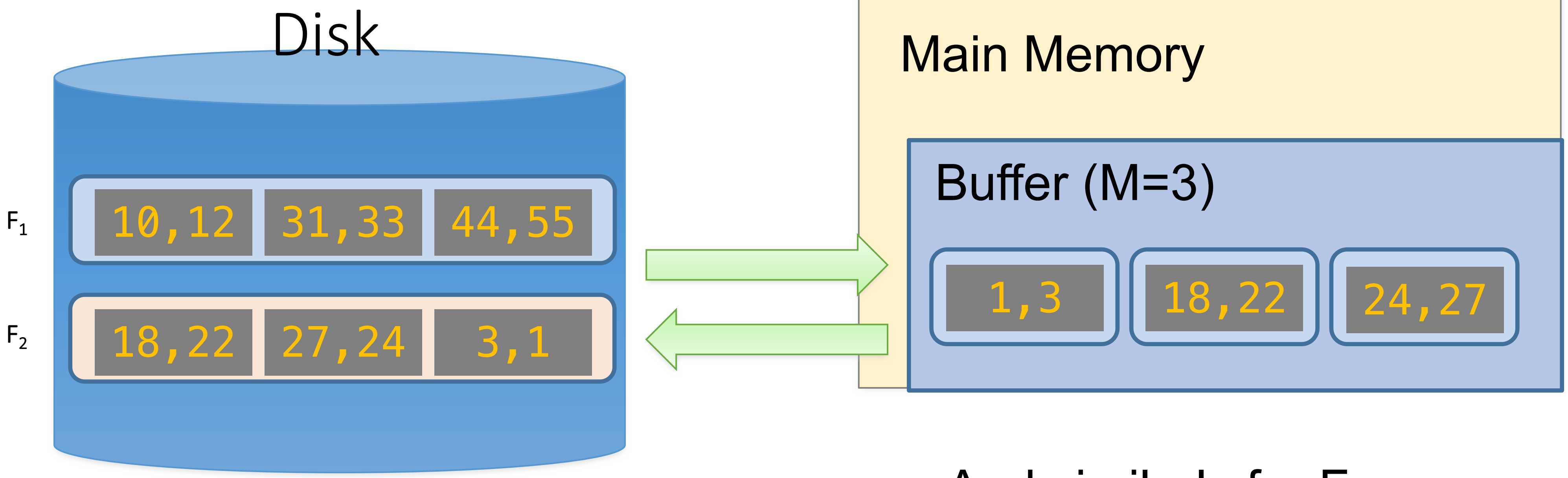
Orange file = unsorted

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

External Merge Sort Algorithm

Example:
- M = 3 Buffer pages
- 6-page file

Each sorted file is a called a *run*



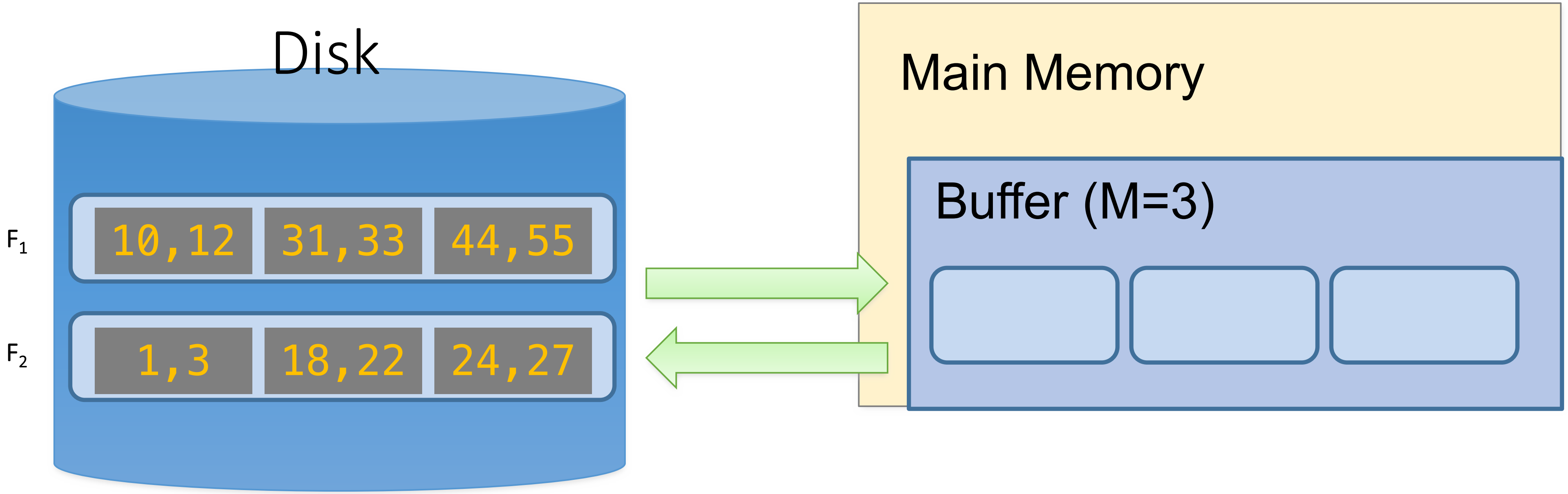
And similarly for F_2

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

External Merge Sort Algorithm

Example:

- M = 3 Buffer pages
- 6-page file



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

Calculating IO Cost

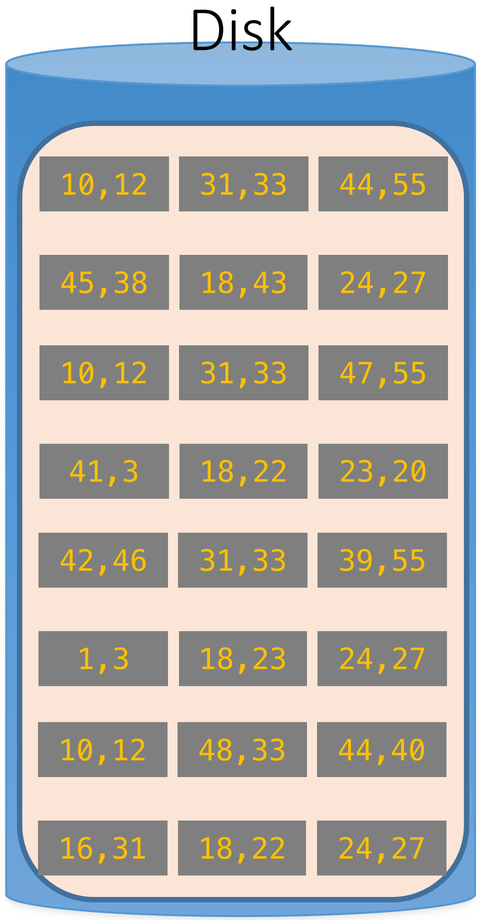
For 3 buffer pages, 6 page file:

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



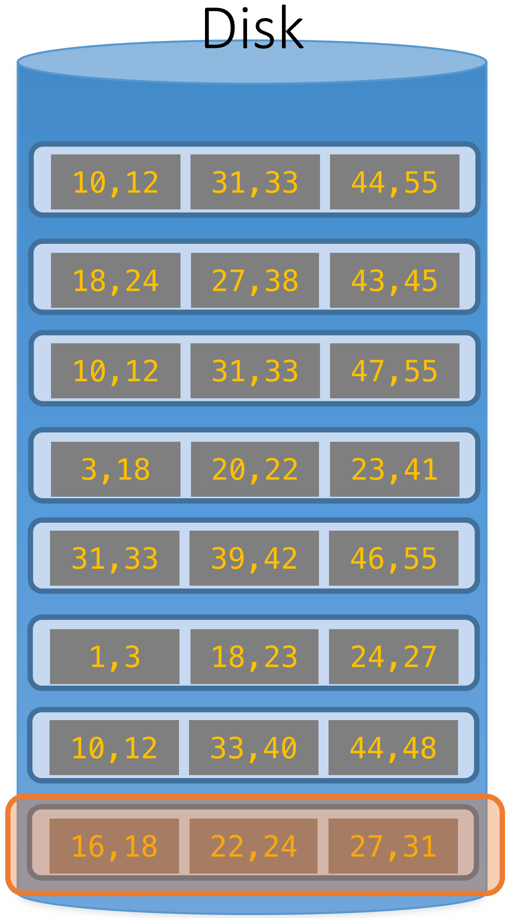
Assume we still only have *3* buffer pages *(Buffer not pictured)*; *M=3*

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); M=3*

Running External Merge Sort on Larger Files

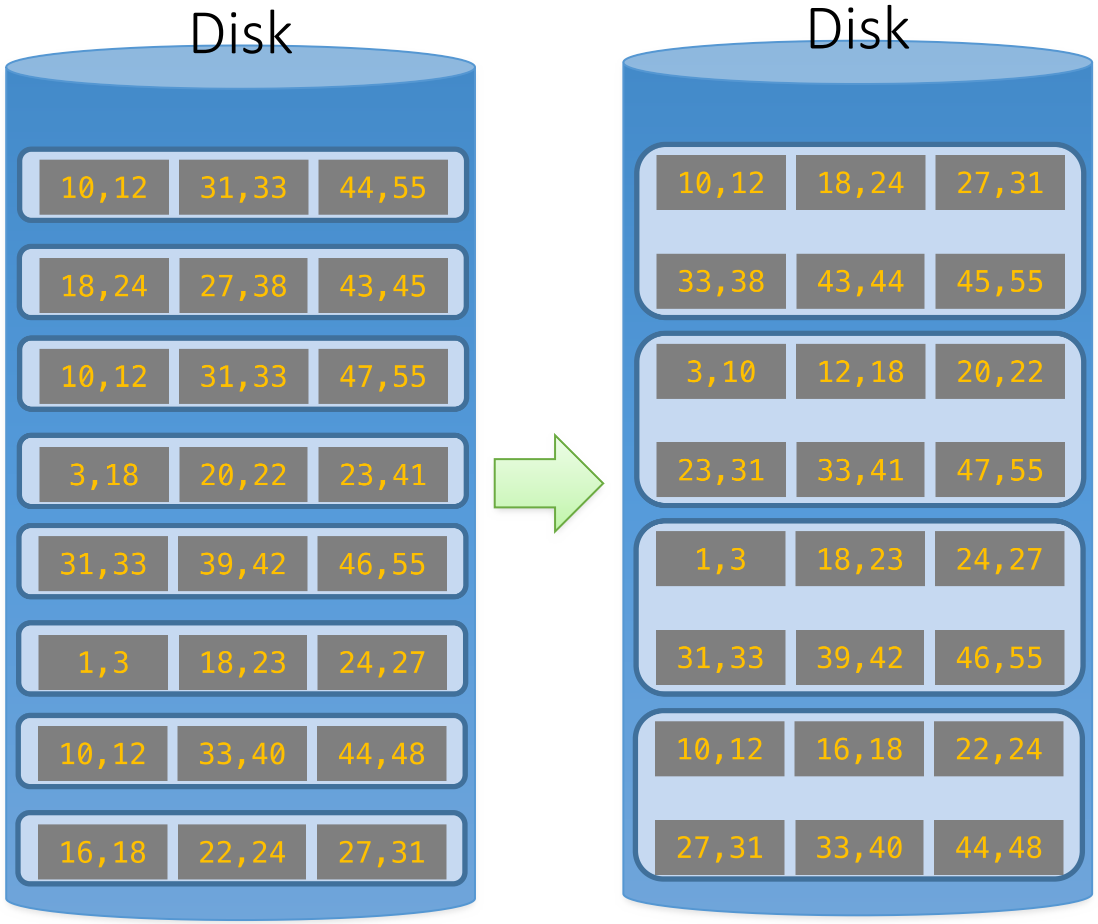


1. Split into files small enough to sort in buffer… and sort

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

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

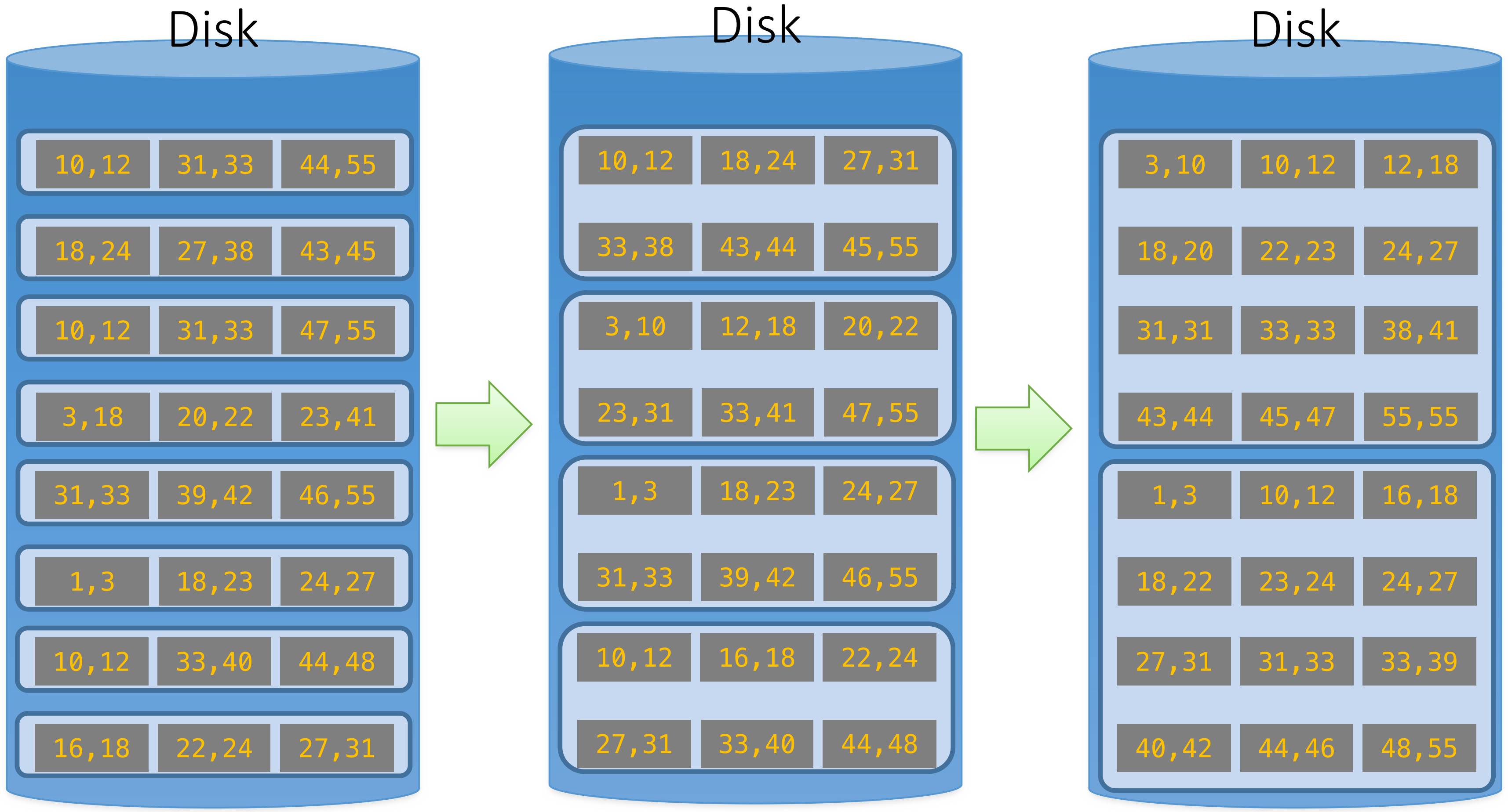
Running External Merge Sort on Larger Files



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

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

Running External Merge Sort on Larger Files

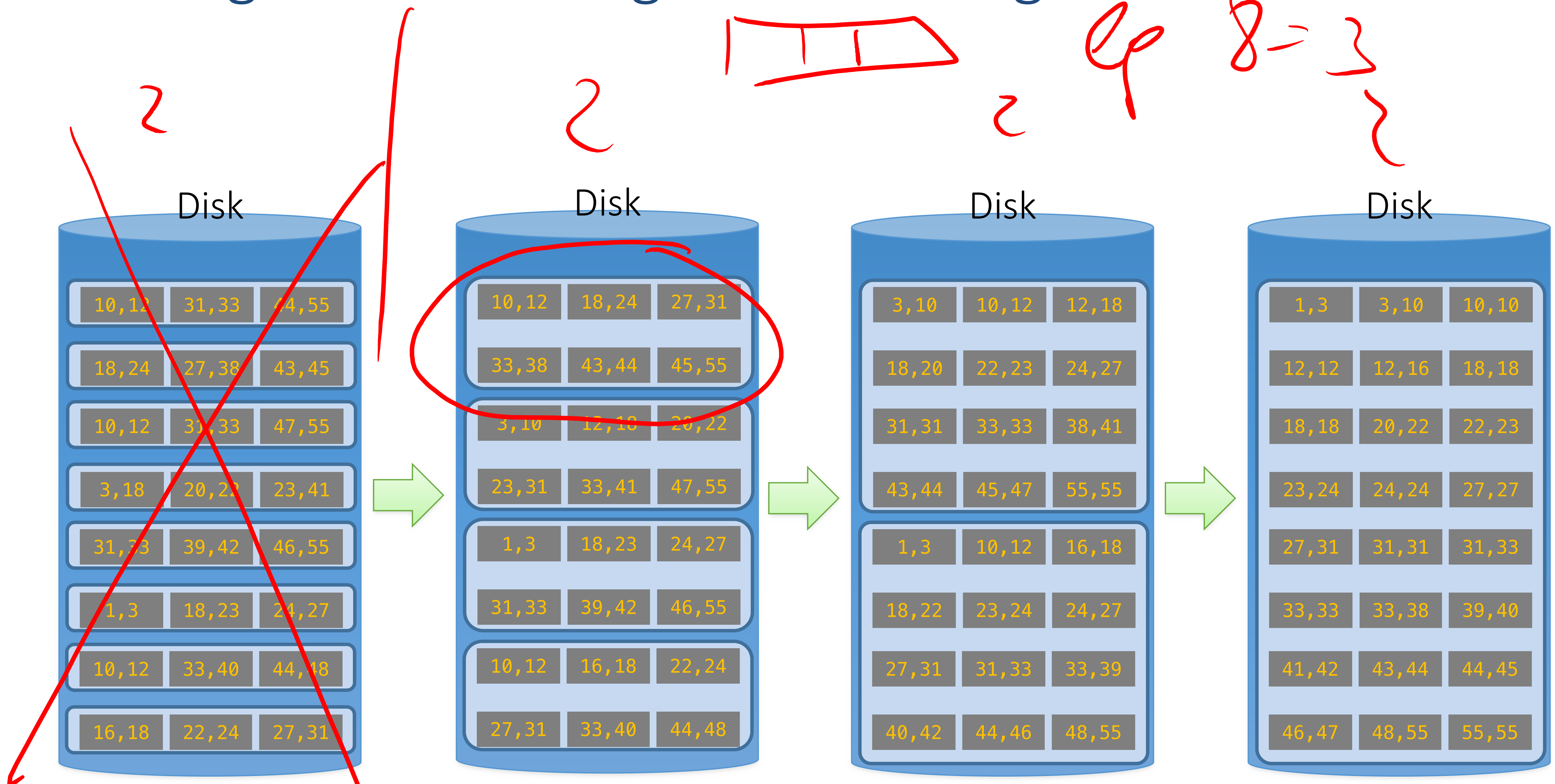


Assume we still only have *3* buffer pages *(Buffer not pictured); M=3*

3. And repeat…

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

Running External Merge Sort on Larger Files

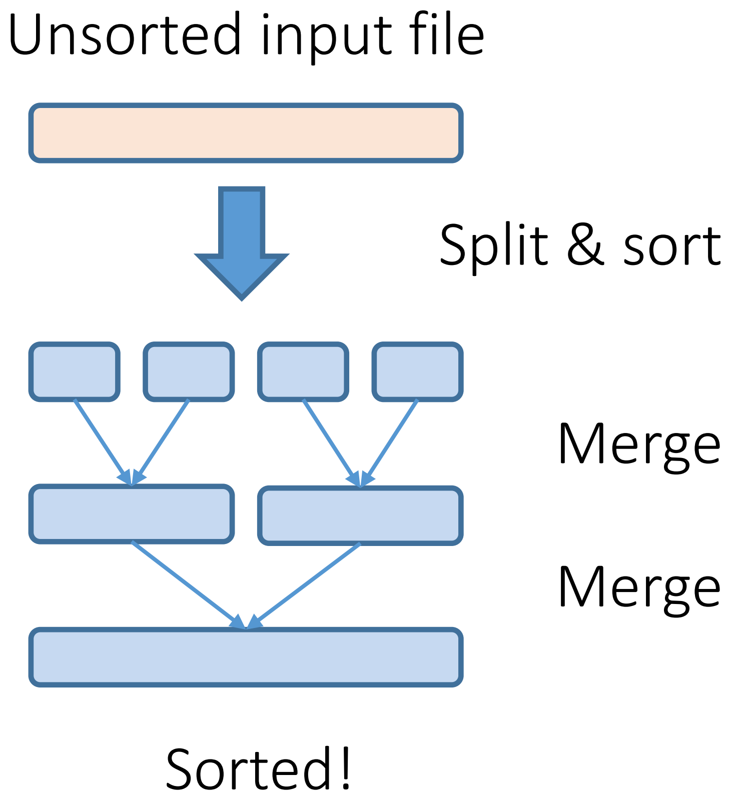


4. And repeat!

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!

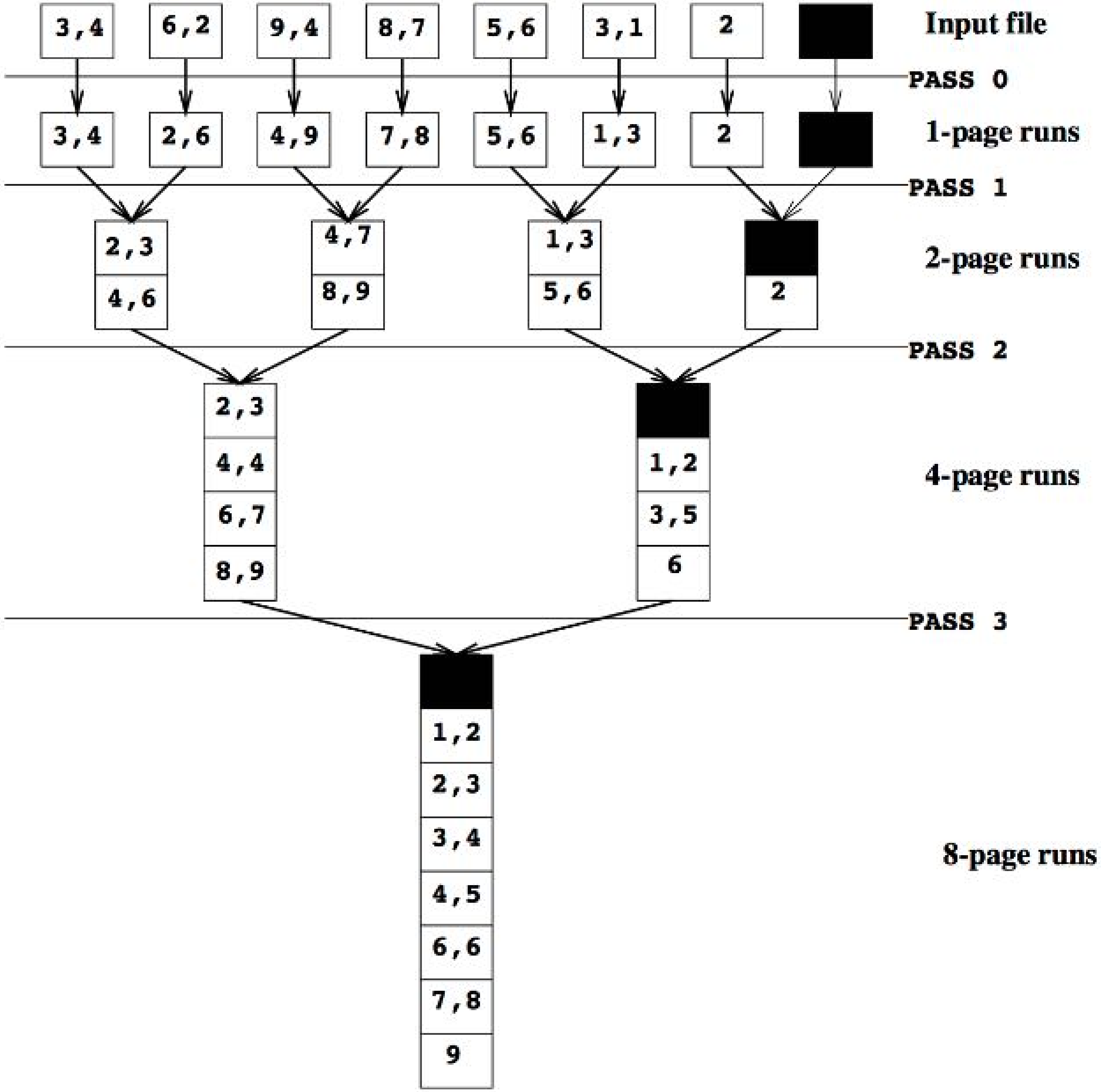


Figure 11.2 Two-Way Merge Sort of a Seven-Page File

Source: "Cow book" (Ramakrishan, Gehrke, Database management systems, 2003)

External Merge Sort: Optimizations

Now assume we have M buffer pages; three optimizations:

1. Increase the length of initial runs

2. (M-1)-way merges

3. Repacking

Using M buffer pages to reduce # of passes

Suppose we have M buffer pages now; we can:

1. Increase length of initial runs. Sort M at a time!

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

IO Cost:

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

Starting with runs of length 1

Starting with runs of length *M*

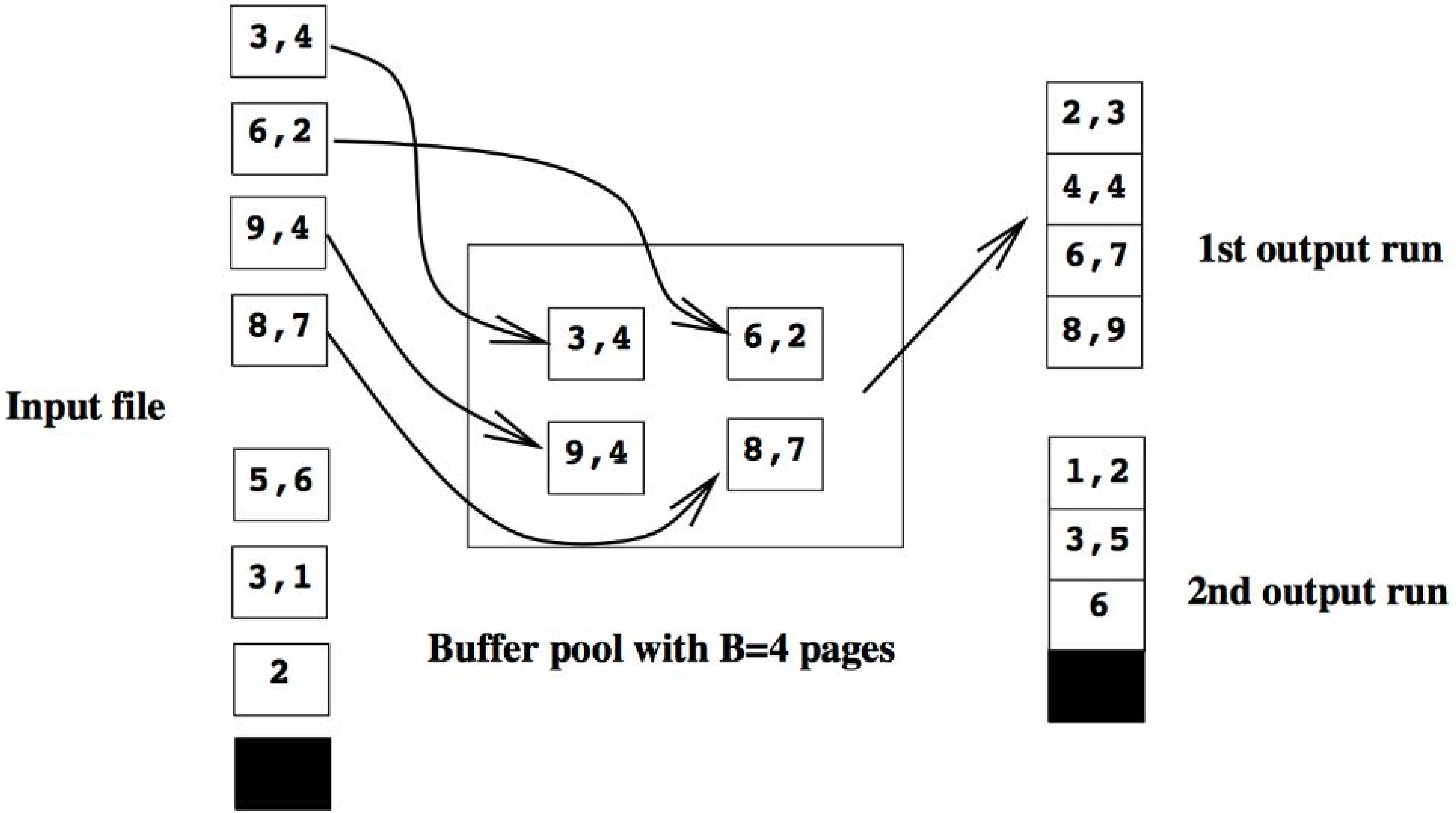


Figure 11.4 External Merge Sort with B Buffer Pages: Pass 0

Source: "Cow book" (Ramakrishan, Gehrke, Database management systems, 2003)

Using M buffer pages to reduce # of passes

Suppose we have M buffer pages now; we can:

2. Perform a (M-1)-way merge.

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

IO Cost:

$2N(\lceil \log_2 N \rceil + 1)$ → $2N\left(\left\lceil \log_2 \frac{\boldsymbol{N}}{\boldsymbol{M}} \right\rceil + 1\right)$ → $2N\left(\left\lceil \log_{M-1} \frac{\boldsymbol{N}}{\boldsymbol{M}} \right\rceil + 1\right)$

Starting with runs of length 1

Starting with runs of length ***M***

Performing ***(M-1)***-way merges

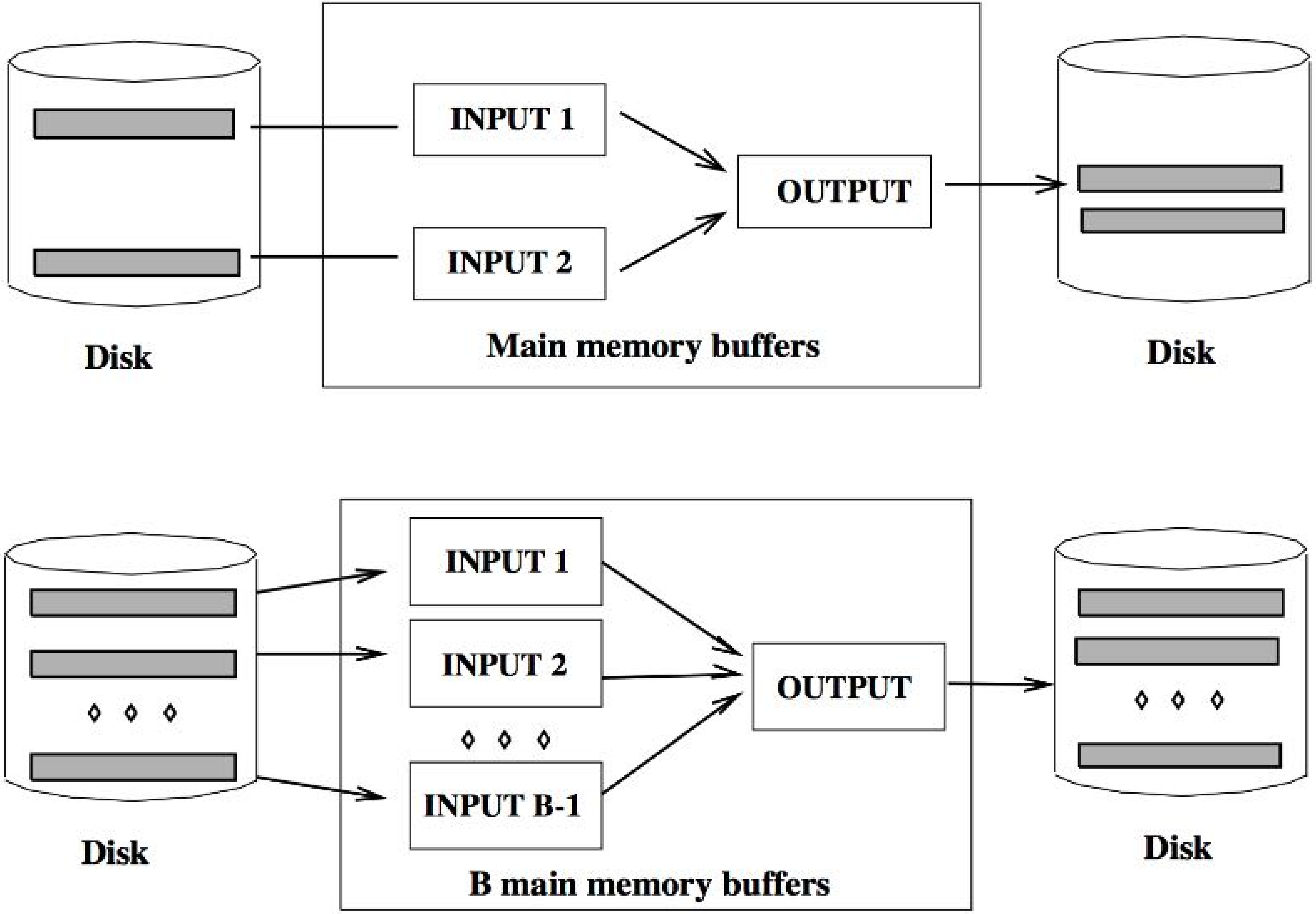


Figure 11.5 External Merge Sort with B Buffer Pages: Pass $i > 0$

Source: "Cow book" (Ramakrishan, Gehrke, Database management systems, 2003)

N	B=3	B=5	B=9	B=17	B=129	B=257
100	7	4	3	2	1	1
1,000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

Figure 11.7 Number of Passes of External Merge Sort

Source: "Cow book" (Ramakrishan, Gehrke, Database management systems, 2003)

Repacking for even longer initial runs

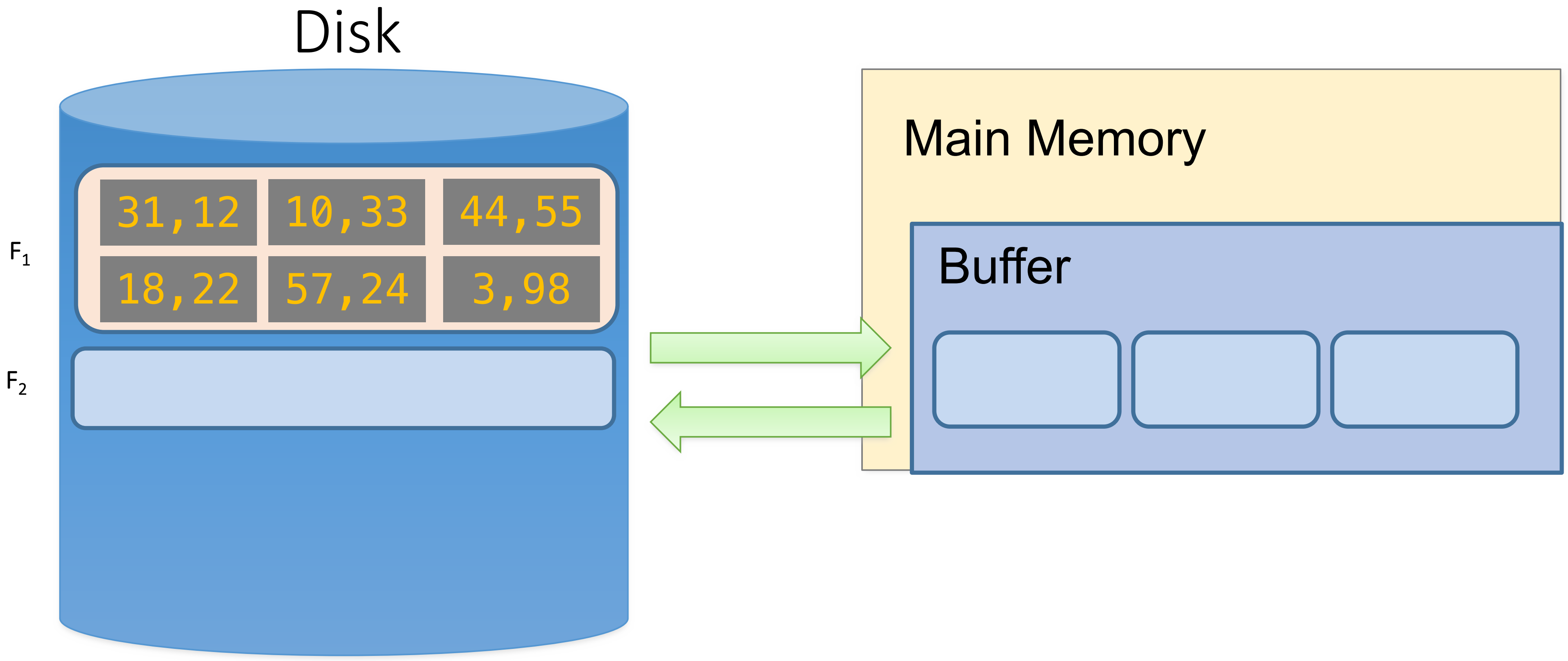
- With B buffer pages, we can now start with M-length initial runs and use (M-1) way merges) to get $2N(\left\lceil \log_{\mathrm{M}-1} \frac{\boldsymbol{N}}{\mathrm{M}} \right\rceil + 1)$ IO cost…

- 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 (replacement sort)

Repacking Example: 3 page buffer

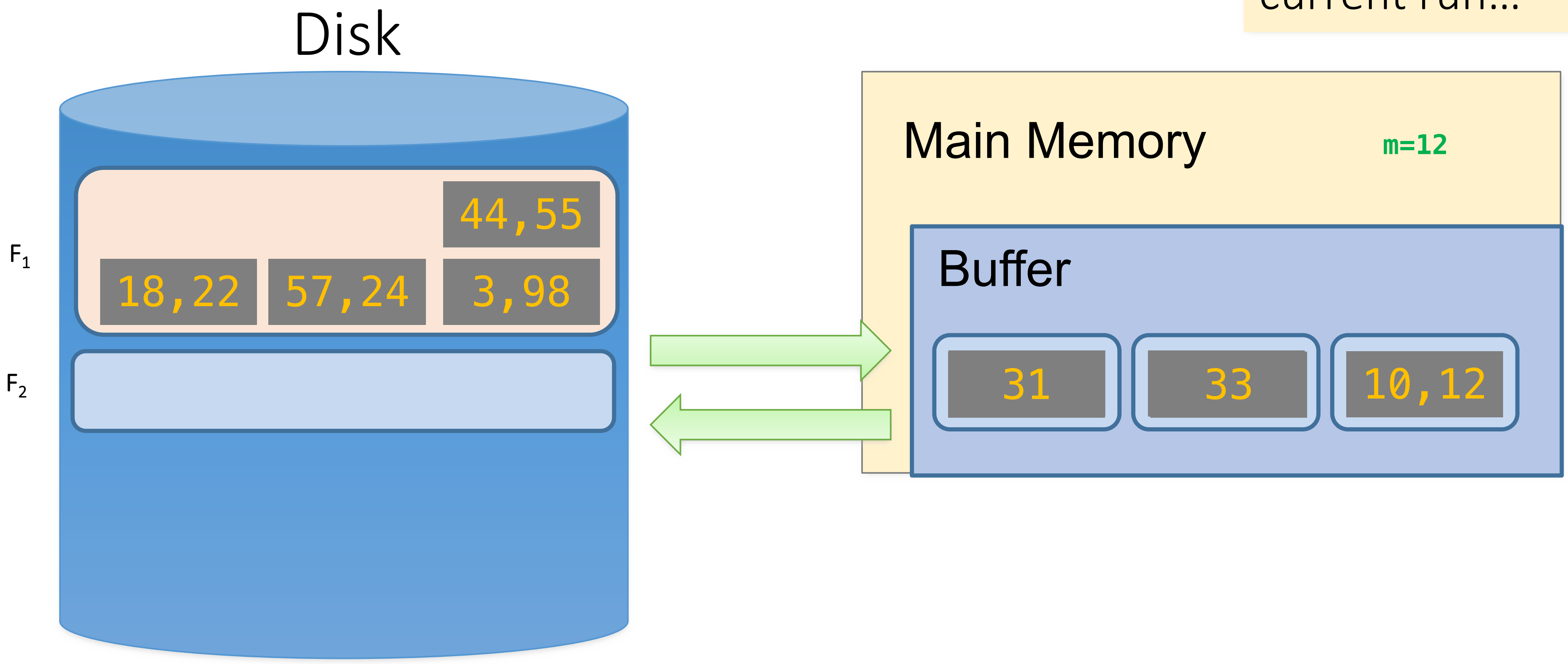
- Start with unsorted single input file, and load 2 pages



Repacking Example: 3 page buffer

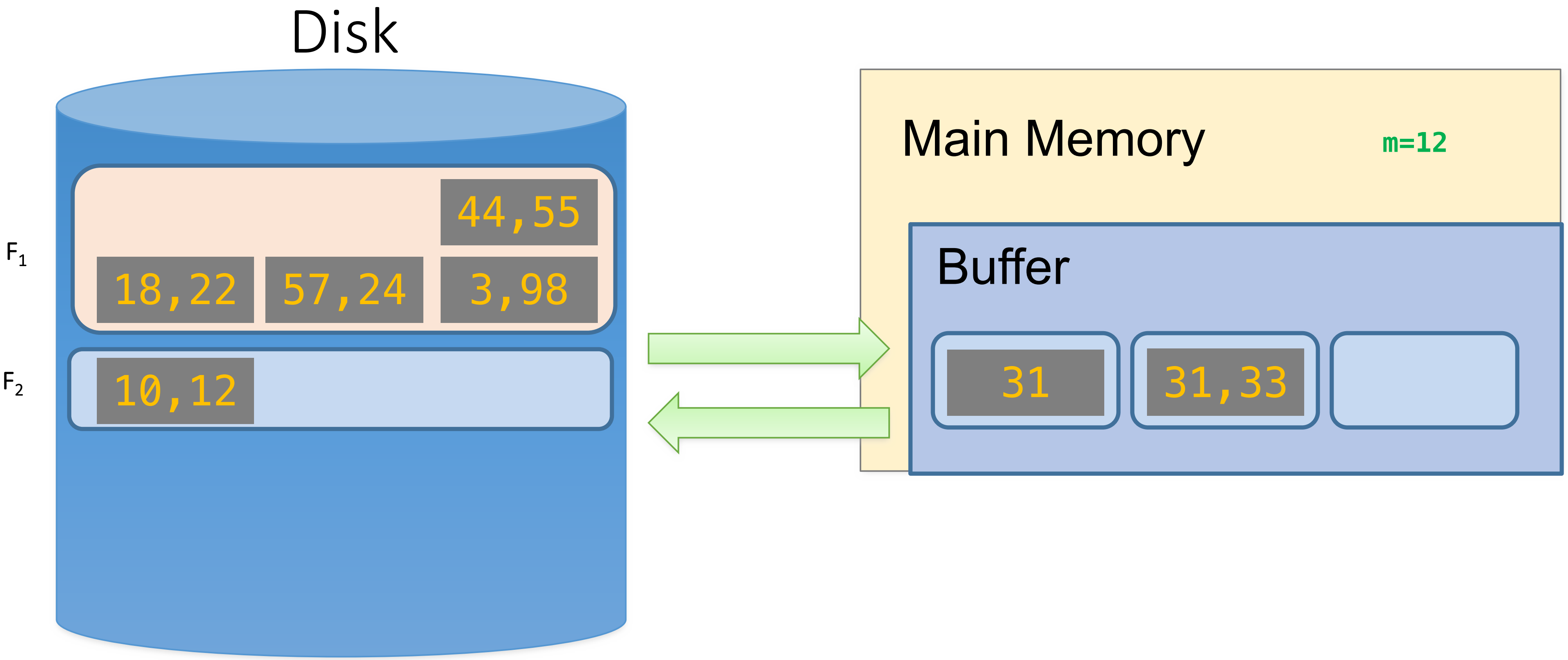
- Take the minimum two values, and put in output page

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



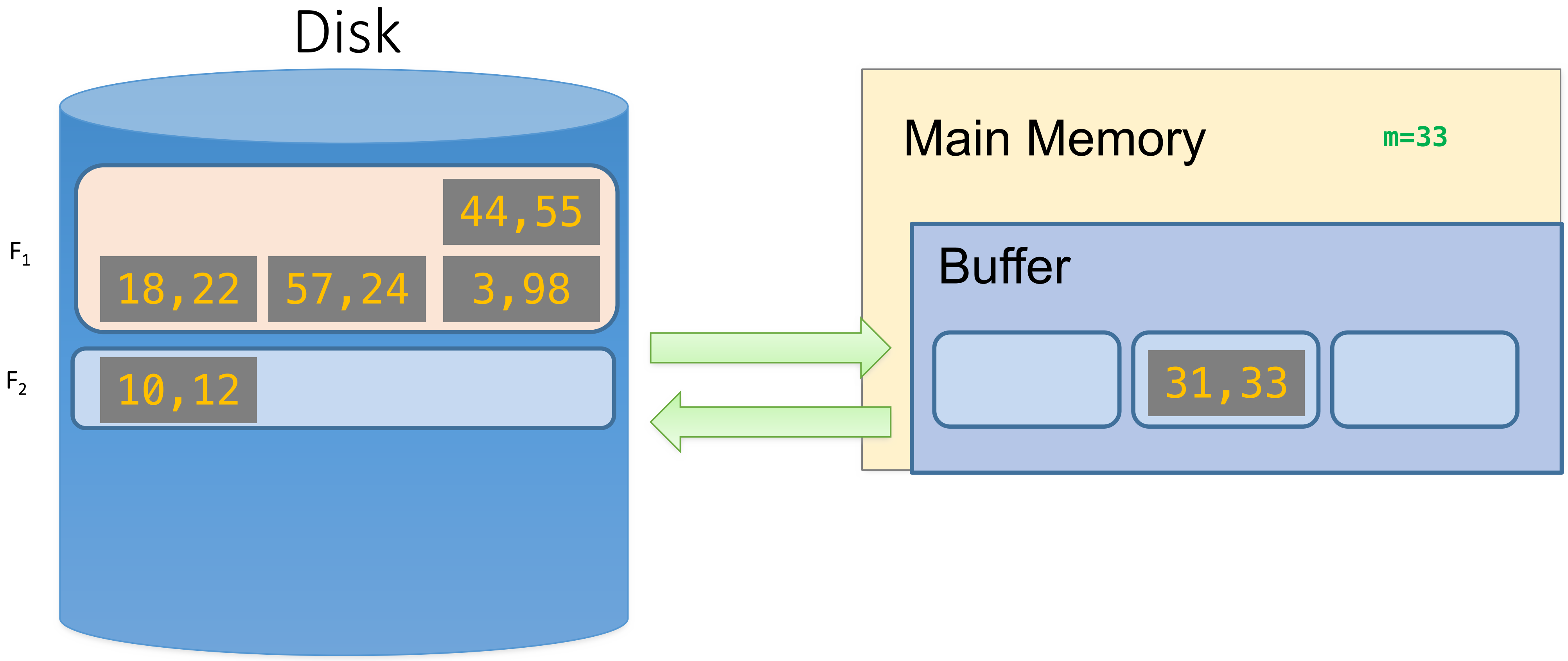
Repacking Example: 3 page buffer

- Next, repack



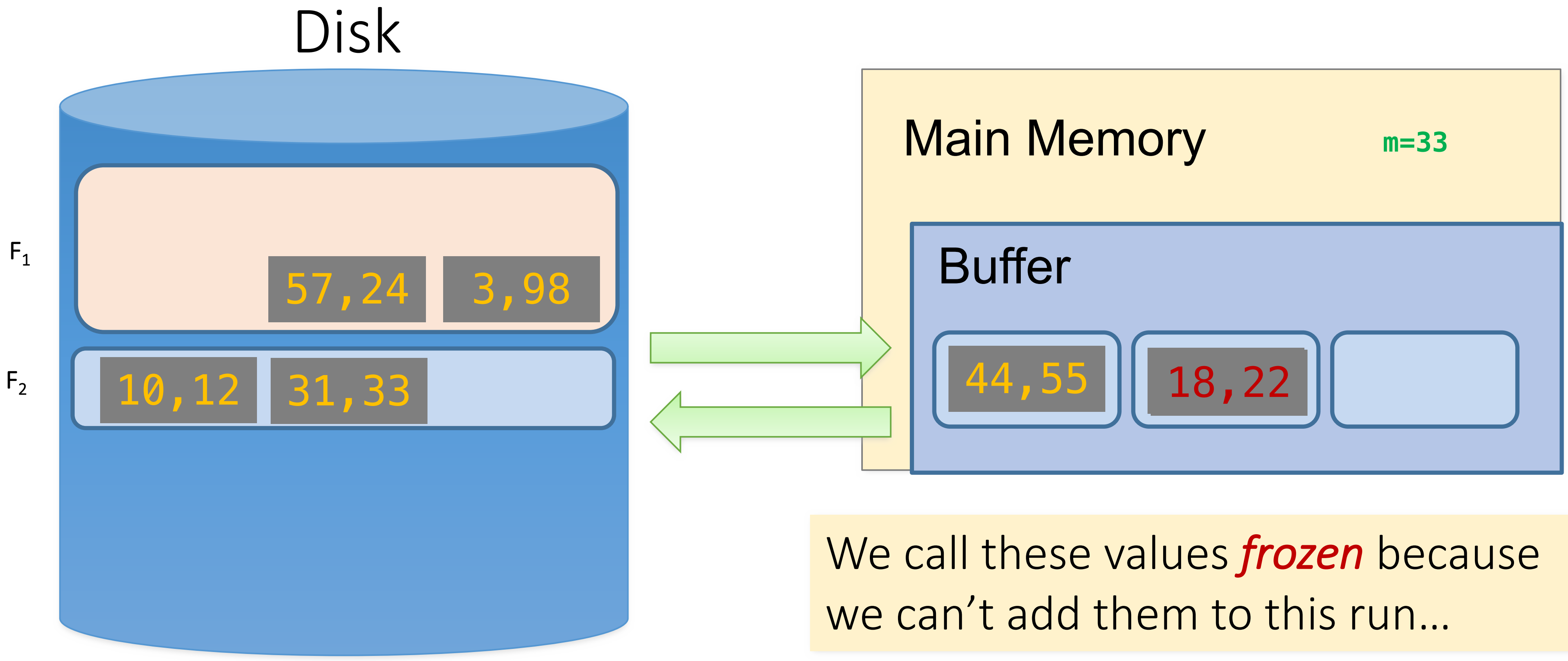
Repacking Example: 3 page buffer

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



Repacking Example: 3 page buffer

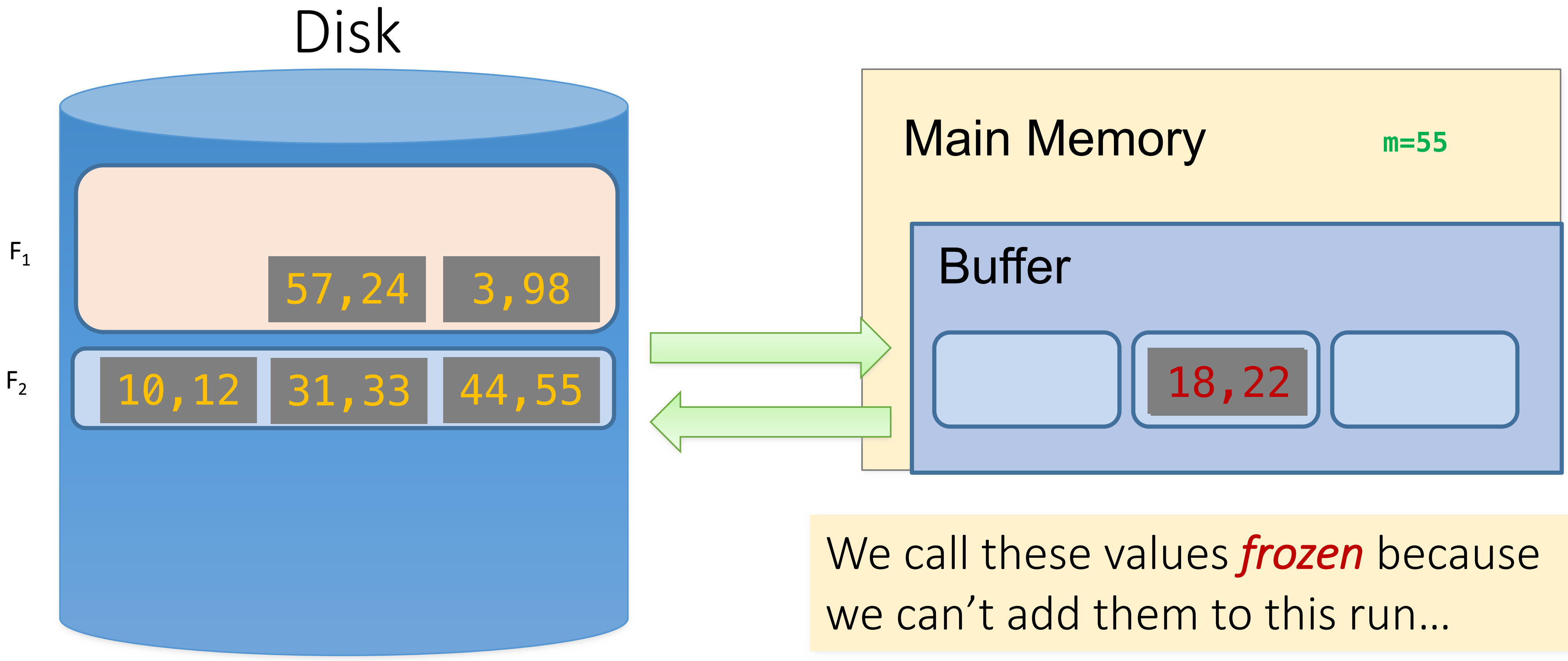
- Now, however, the smallest values are less than the largest (last) in the sorted run…



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

Repacking Example: 3 page buffer

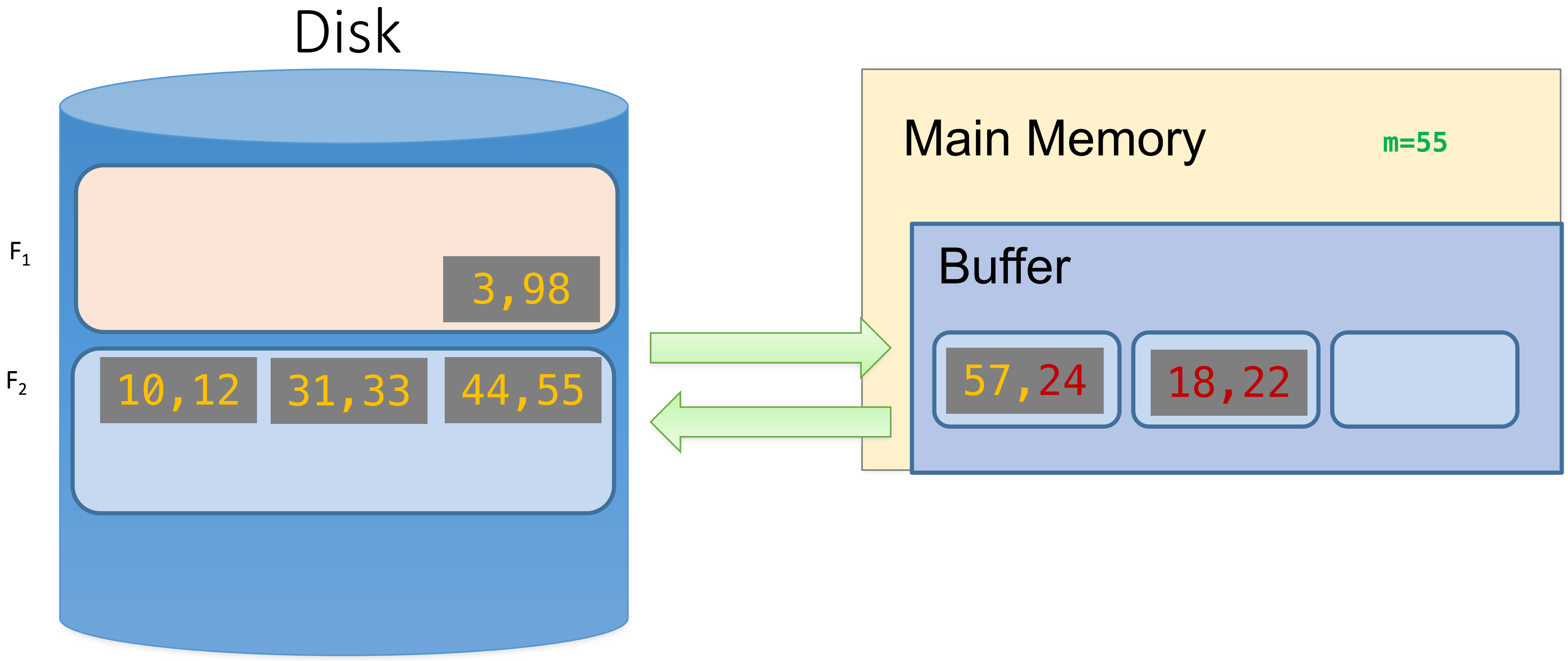
- Now, however, the smallest values are less than the largest (last) in the sorted run…



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

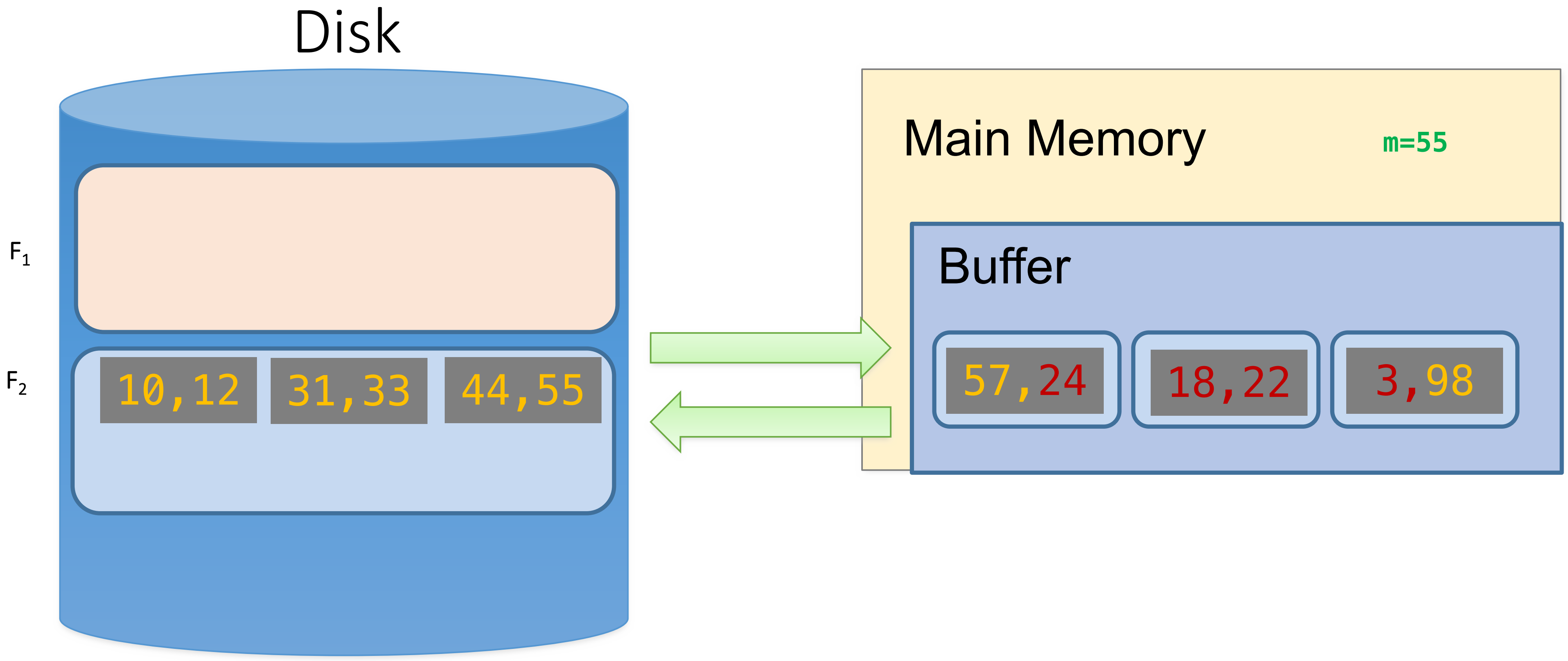
Repacking Example: 3 page buffer

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



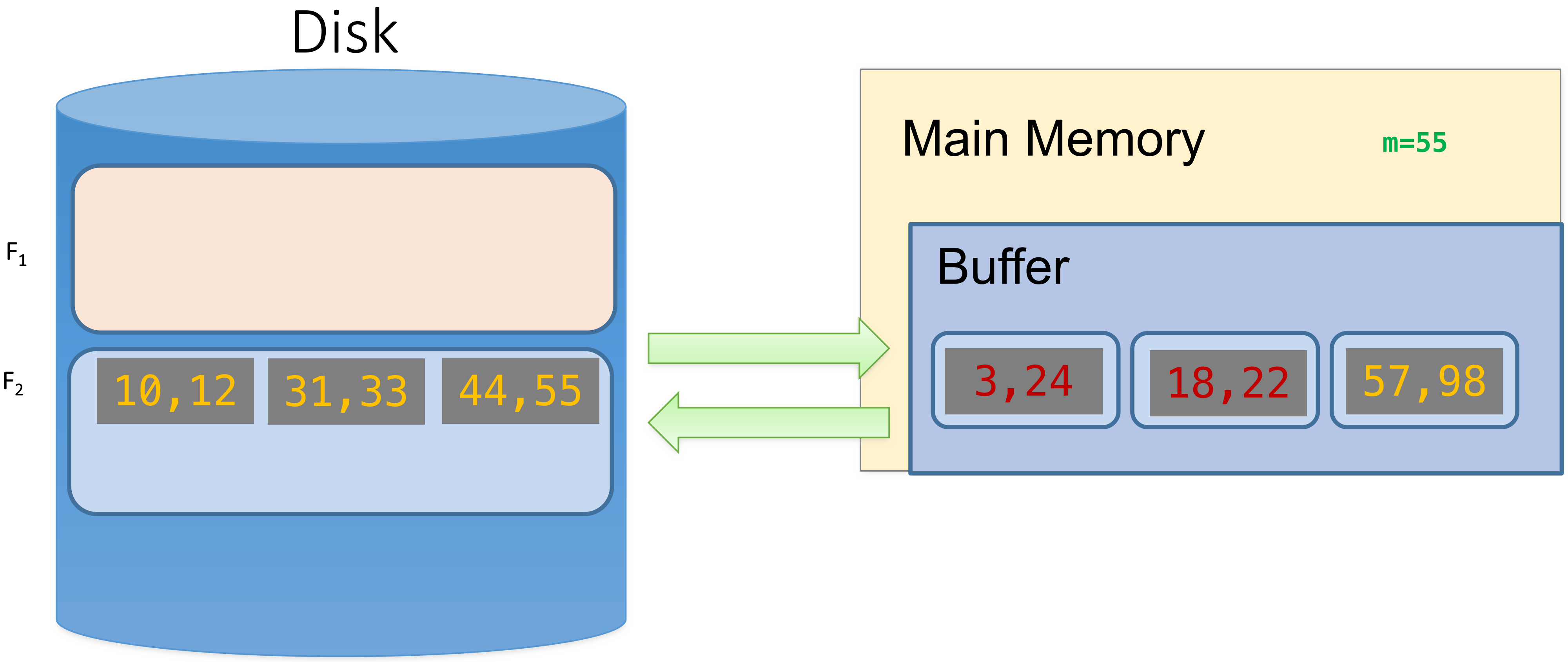
Repacking Example: 3 page buffer

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



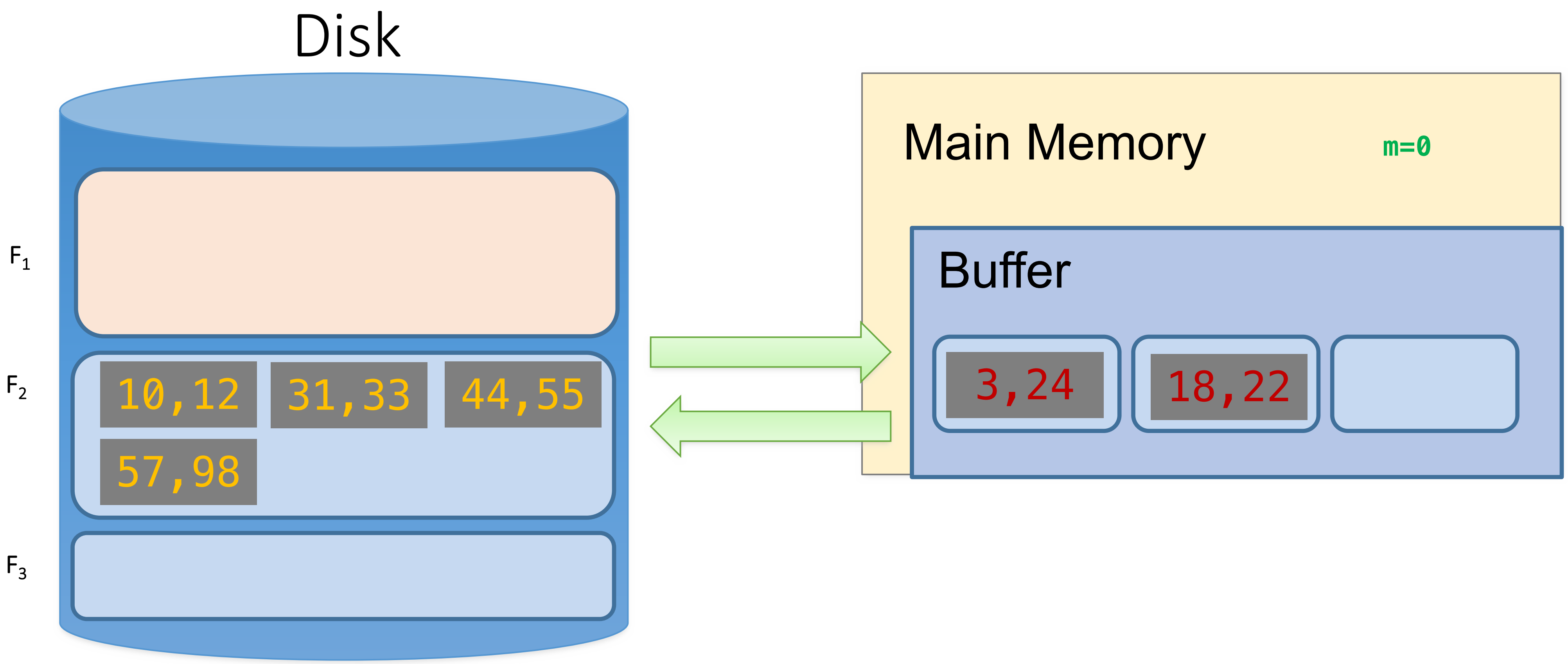
Repacking Example: 3 page buffer

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



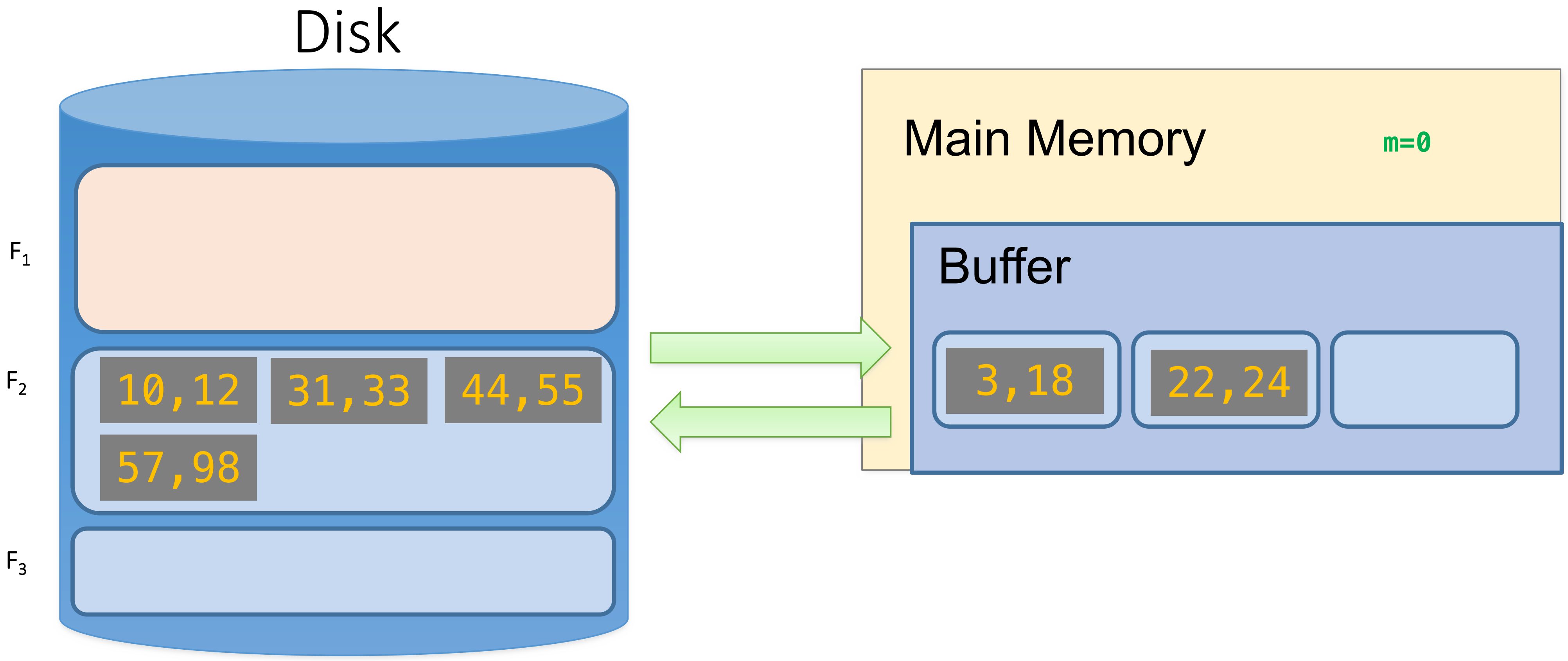
Repacking Example: 3 page buffer

- 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



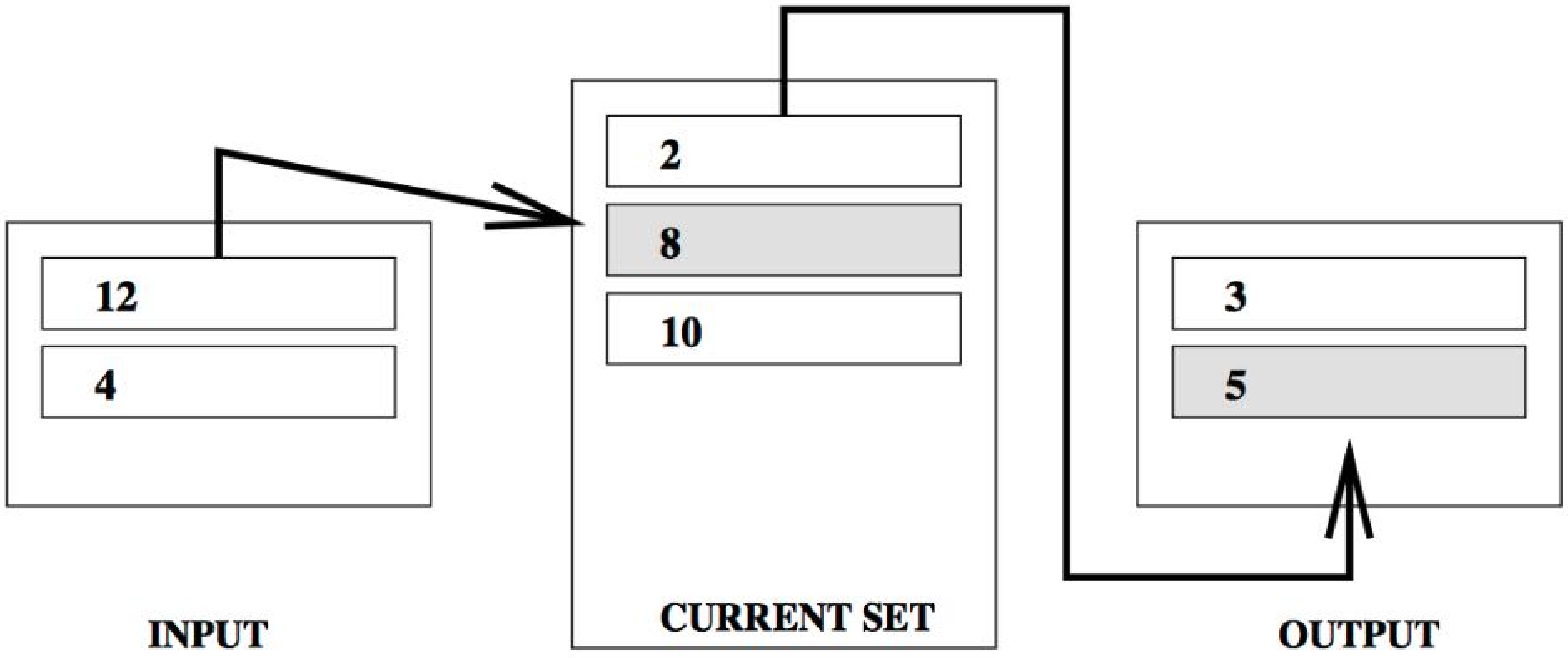
Repacking

- Note that, for buffer with M 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 M

- In general, with repacking we do no worse than without it!

- Engineer's approximation: runs will have ~2M length

$$\sim 2N\left(\left\lceil \log_{M-1} \frac{\boldsymbol{N}}{\boldsymbol{2M}} \right\rceil + 1\right)$$



Source: "Cow book" (Ramakrishan, Gehrke, Database management systems, 2003)

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

B+ trees: and IO-aware index structure

1) Indexes: Motivations & Basics
2) B+ Trees

“If you don’t find it in the index,
look very carefully through the entire catalog”

- Sears, Roebuck and Co., Consumers Guide, 1897

What we will learn next

- Indexes: Motivation

- Indexes: Basics

- ACTIVITY 41: Creating indexes

Index Motivation

- Suppose we want to search for people of a specific age

`Person(name, age)`

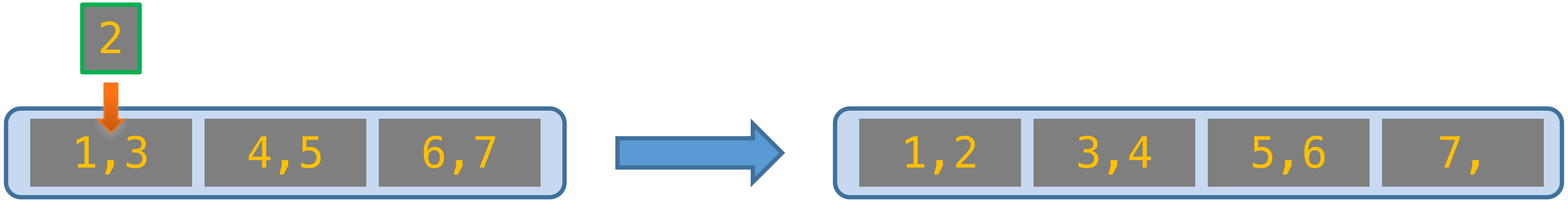
- First idea: Sort the records by age… we know how to do this fast!

- How many IO operations to search over N sorted records?
  - Simple scan: O(N)
  - Binary search: O($\mathbf{log_2}\ \boldsymbol{N}$)

Could we get even cheaper search? E.g. go from $\mathbf{log_2}\ \boldsymbol{N}$ → $\mathbf{log_{200}}\ \boldsymbol{N}$?

Index Motivation

- What about if we want to insert a new person, but keep the list sorted?



- We would have to potentially shift N records, requiring up to ~ 2*N/P IO operations (where P = # of records per page)!
  - We could leave some "slack" in the pages…

Could we get faster insertions?

Index Motivation

- What about if we want to be able to search quickly along multiple attributes (e.g. not just age)?
  - We could keep multiple copies of the records, each sorted by one attribute set… this would take a lot of space

Can we get fast search over multiple attribute (sets) without taking too much space?

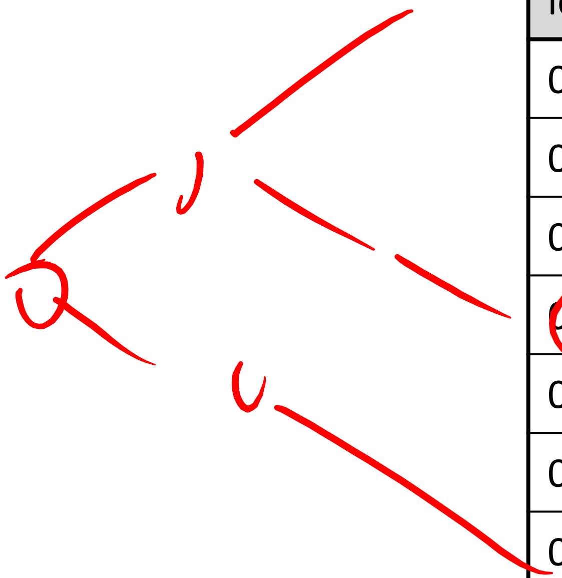
We'll create separate data structures called *indexes* to address all these points

Further Motivation for Indexes: NoSQL!

- NoSQL engines are (basically) just indexes!
  - A lot more is left to the user in NoSQL… one of the primary remaining functions of the DBMS is still to provide index over the data records, for the reasons we just saw!
  - Sometimes use B+ Trees (covered next), sometimes hash indexes (discussed later)

Indexes are critical across all DBMS types

High-level overview: indexes



id	age	salary	other
006	19	50k	...
005	20	55k	...
004	25	50k	...
007	30	80k	...
002	35	75k	...
003	35	70k	...
001	40	65k	...

id	age	salary	other
006	19	50k	...
004	25	50k	...
005	20	55k	...
001	40	65k	...
003	35	70k	...
002	35	75k	...
007	30	80k	...

data file = index file
clustered (primary) index

index file
unclustered (secondary) index

Indexes: High-level

- An <u>index</u> on a file speeds up selections on the <u>search key</u> fields for the index.
  - Search key properties
    - Any subset of fields
    - is not the same as key of a relation

- Example:

```
Product(name, maker, price)
```

On which attributes would you build indexes?

More precisely

- An **index** is a data structure mapping search keys to sets of rows in a database table
  - Provides efficient lookup & retrieval by search key value- usually much faster than searching through all the rows of the database table
- An index can store the full rows it points to (primary index) or pointers to those rows (secondary index)
  - We'll mainly consider secondary indexes

Operations on an Index

- Search: Quickly find all records which meet some condition on the search key attributes
  - More sophisticated variants as well. Why?

- Insert / Remove entries
  - Bulk Load / Delete. Why?

Indexing is one the most important features provided by a database for performance