

Lecture 12: The IO Model & External Sorting

Announcements

1. *Thank you for the great feedback (post coming soon)!*
2. Educational goals:
 1. **Tech changes, principles change more slowly** We teach principles and formal abstraction so you can adapt to a changing world and technology..
 2. **Ability to learn after you leave.** Why we give you new concepts in homeworks & projects. *We want you to be able to pick up those changing concepts. But we test you fairly.*
 3. **We select the essentials for you.** We've thought about the material quite a bit. Feedback helpful, but we'd hope to get the benefit of the doubt. ☺
3. Thank you for being awesome wrt the midterm.
 1. ... some of you started early... Not cool.
 2. SCPD people, a lot of you were *great!*

Today's Lecture

1. *[From 9-2]: Conflict Serializability & Deadlock*
2. The Buffer
3. External Merge Sort

1. Conflict Serializability & Deadlock

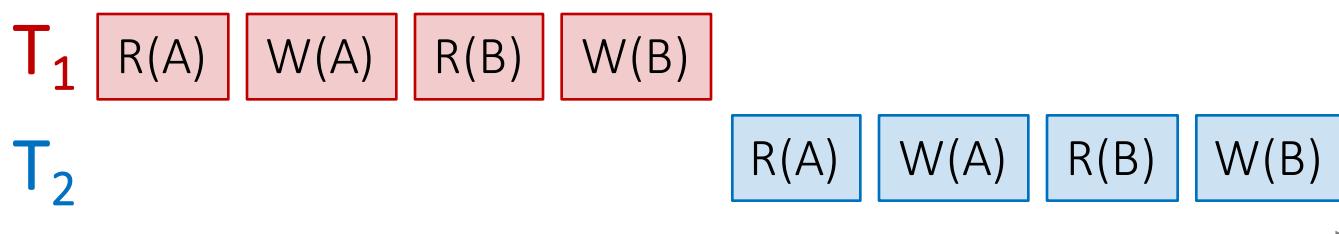
Recap from Lecture 9-2

What you will learn about in this section

1. RECAP: Concurrency
2. Conflict Serializability
3. DAGs & Topological Orderings
4. Strict 2PL
5. Deadlocks

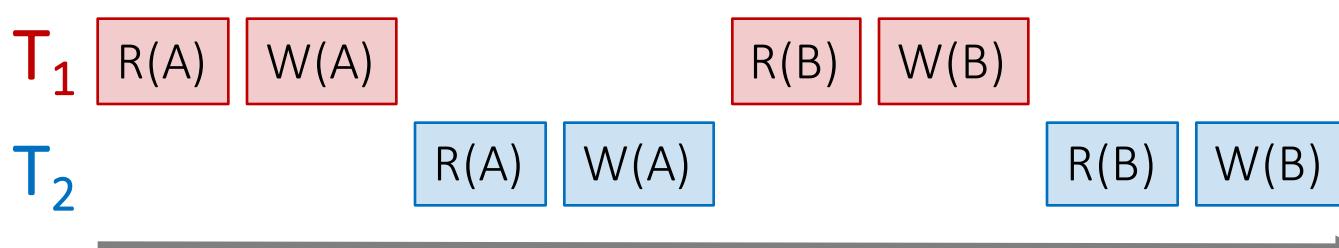
Recall: Concurrency as Interleaving TXNs

Serial Schedule:



- For our purposes, having TXNs occur concurrently means **interleaving their component actions (R/W)**

Interleaved Schedule:



We call the particular order of interleaving a **schedule**

Recall: Why Interleave TXNs?

- Interleaving TXNs might lead to anomalous outcomes... why do it?
- Several important reasons:
 - Individual TXNs might be *slow*- don't want to block other users during!
 - Disk access may be *slow*- let some TXNs use CPUs while others accessing disk!

All concern large differences in *performance*

Recall: Must Preserve Consistency & Isolation

- The DBMS has freedom to interleave TXNs
- However, it must pick an interleaving or **schedule** such that isolation and consistency are maintained
 - Must be *as if* the TXNs had executed serially!

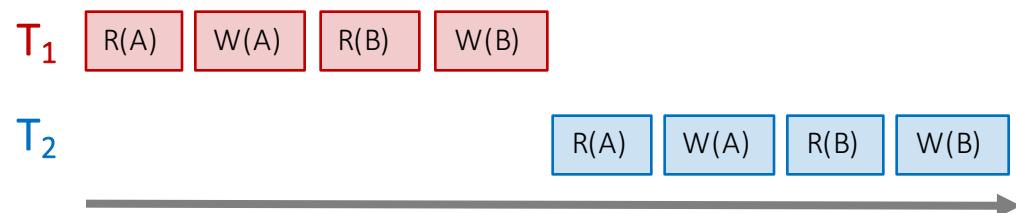
“With great power comes great responsibility”

ACID

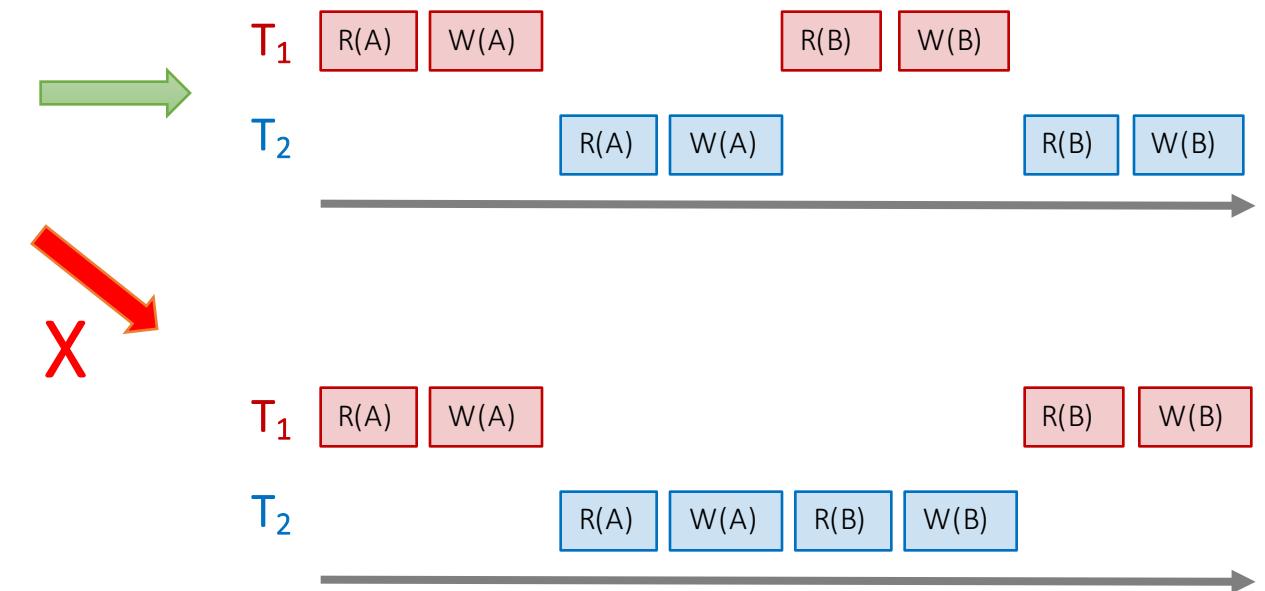
DBMS must pick a schedule which maintains isolation & consistency

Recall: “Good” vs. “bad” schedules

Serial Schedule:



Interleaved Schedules:



Why?

We want to develop ways of discerning “good” vs. “bad” schedules

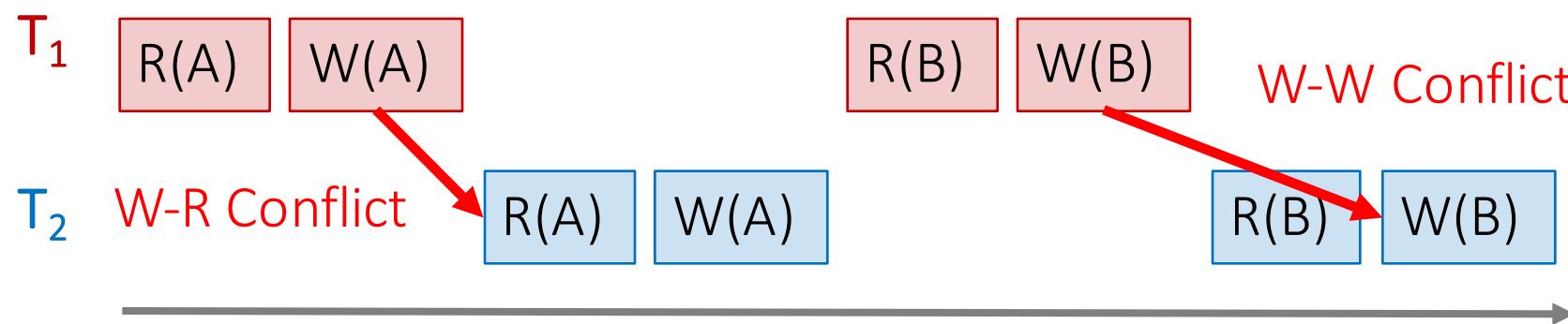
Ways of Defining “Good” vs. “Bad” Schedules

- Recall from last time: we call a schedule ***serializable*** if it is equivalent to *some* serial schedule
 - We used this as a notion of a “good” interleaved schedule, since a **serializable schedule will maintain isolation & consistency**
- Now, we’ll define a stricter, but very useful variant:
 - **Conflict serializability**

We'll need to define
conflicts first..

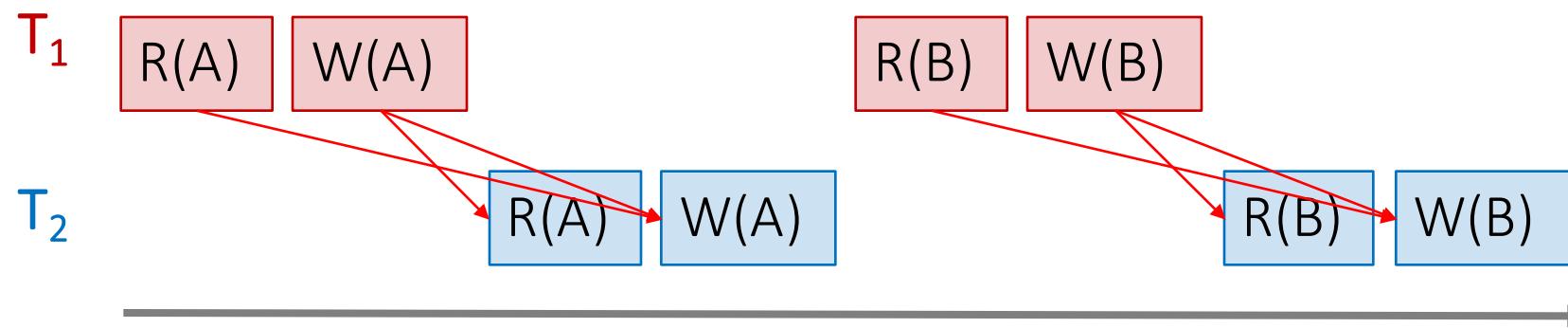
Conflicts

Two actions **conflict** if they are part of different TXNs, involve the same variable, and at least one of them is a write



Conflicts

Two actions **conflict** if they are part of different TXNs, involve the same variable, and at least one of them is a write



All “conflicts”!

Conflict Serializability

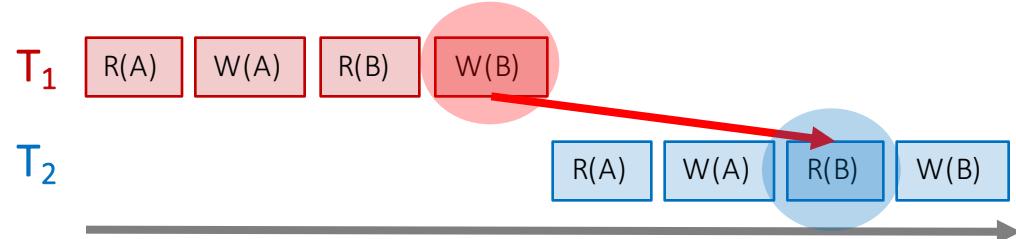
- Two schedules are **conflict equivalent** if:
 - They involve *the same actions of the same TXNs*
 - Every *pair of conflicting actions* of two TXNs are *ordered in the same way*
- Schedule S is **conflict serializable** if S is *conflict equivalent* to some serial schedule

Conflict serializable \Rightarrow serializable

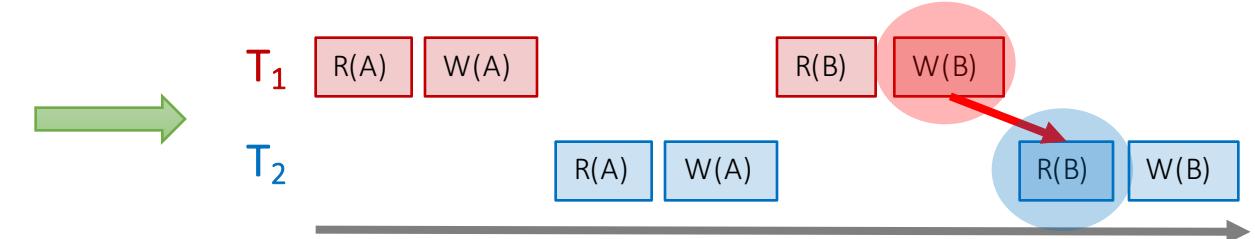
So if we have conflict serializable, we have consistency & isolation!

Recall: “Good” vs. “bad” schedules

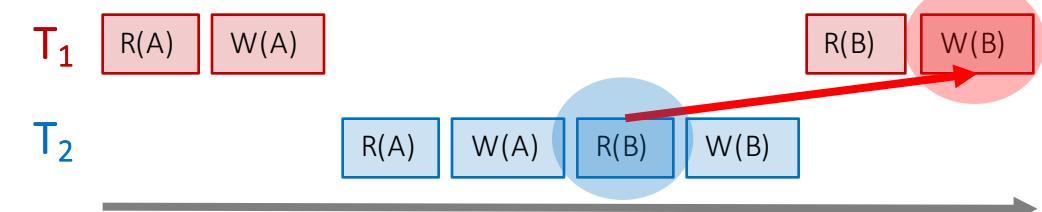
Serial Schedule:



Interleaved Schedules:



Note that in the “bad” schedule, the *order of conflicting actions is different than the above (or any) serial schedule!*



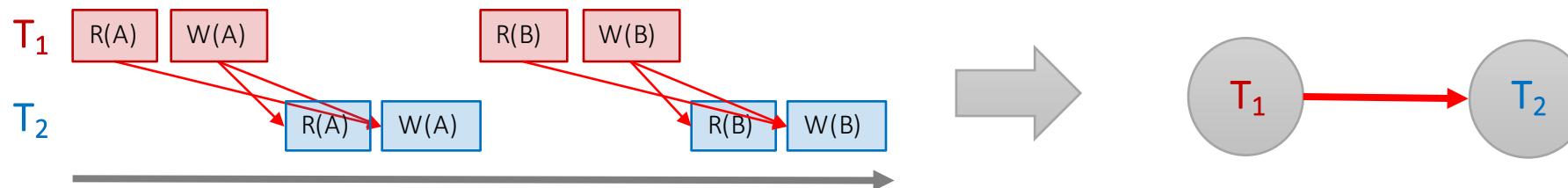
Conflict serializability also provides us with an operative notion of “good” vs. “bad” schedules!

Note: Conflicts vs. Anomalies

- **Conflicts** are things we talk about to help us characterize different schedules
 - Present in both “good” and “bad” schedules
- **Anomalies** are instances where isolation and/or consistency is broken because of a “bad” schedule
 - We often characterize different anomaly types by what types of conflicts predicated them

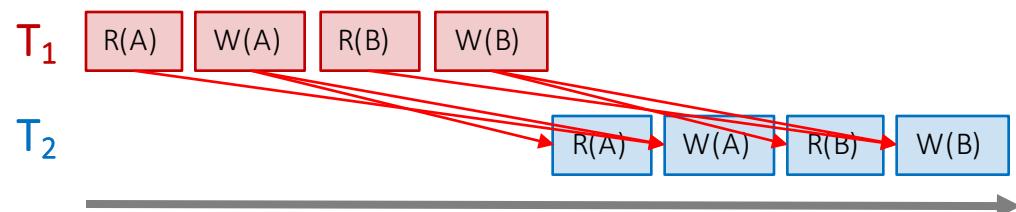
The Conflict Graph

- Let's now consider looking at conflicts **at the TXN level**
- Consider a graph where the **nodes are TXNs**, and there is an edge from $T_i \rightarrow T_j$ if any actions in T_i precede and conflict with any actions in T_j



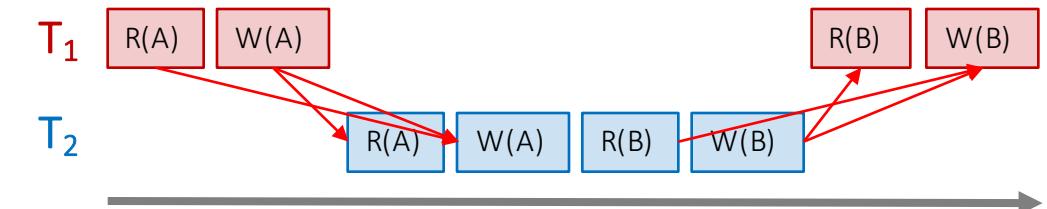
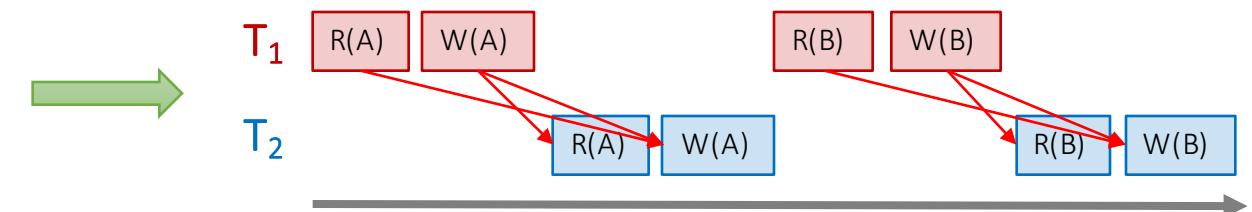
What can we say about “good” vs. “bad” conflict graphs?

Serial Schedule:



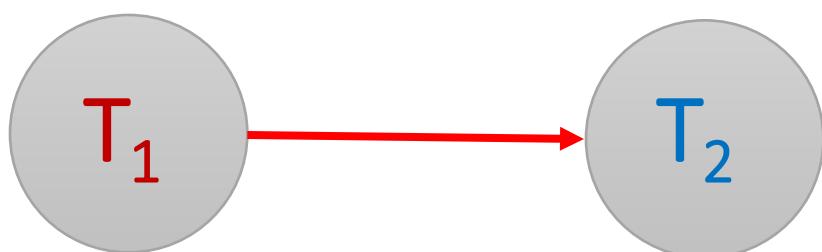
A bit complicated...

Interleaved Schedules:



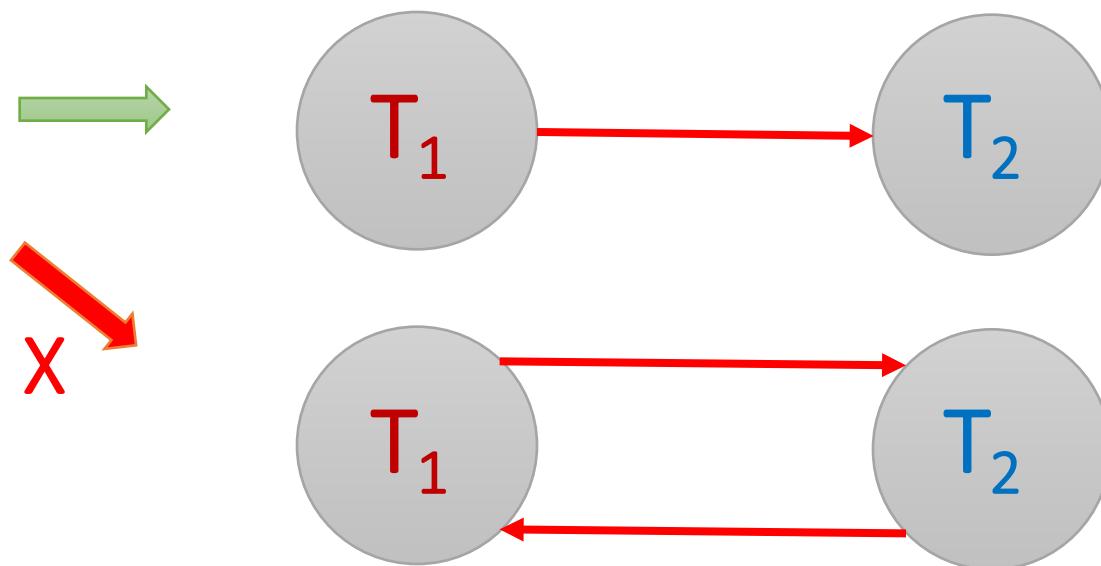
What can we say about “good” vs. “bad” conflict graphs?

Serial Schedule:



Simple!

Interleaved Schedules:



Theorem: Schedule is **conflict serializable** if and only if its conflict graph is acyclic

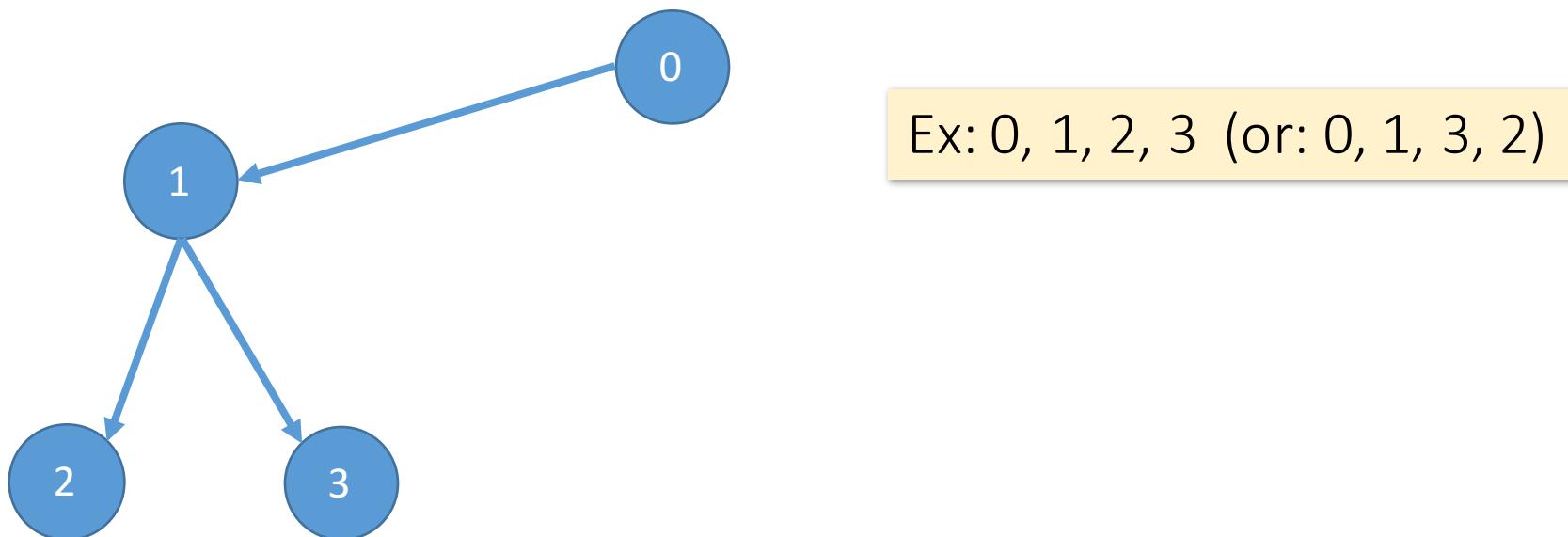
Let's unpack this notion of acyclic
conflict graphs...

DAGs & Topological Orderings

- A **topological ordering** of a directed graph is a linear ordering of its vertices that respects all the directed edges
- A directed acyclic graph (DAG) always has one or more **topological orderings**
 - (And there exists a topological ordering *if and only if* there are no directed cycles)

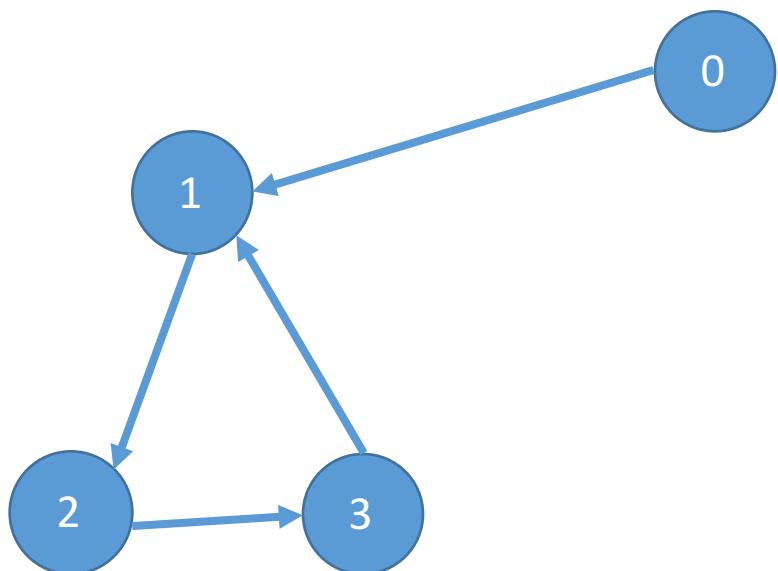
DAGs & Topological Orderings

- Ex: What is one possible topological ordering here?



DAGs & Topological Orderings

- Ex: What is one possible topological ordering here?



There is none!

Connection to conflict serializability

- In the conflict graph, a topological ordering of nodes corresponds to a **serial ordering of TXNs**
- Thus an acyclic conflict graph → conflict serializable!

Theorem: Schedule is **conflict serializable** if and only if its conflict graph is acyclic

Strict Two-Phase Locking

- We consider **locking**- specifically, *strict two-phase locking*- as a way to deal with concurrency, because it **guarantees conflict serializability (if it completes- see upcoming...)**
- Also (*conceptually*) straightforward to implement, and transparent to the user!

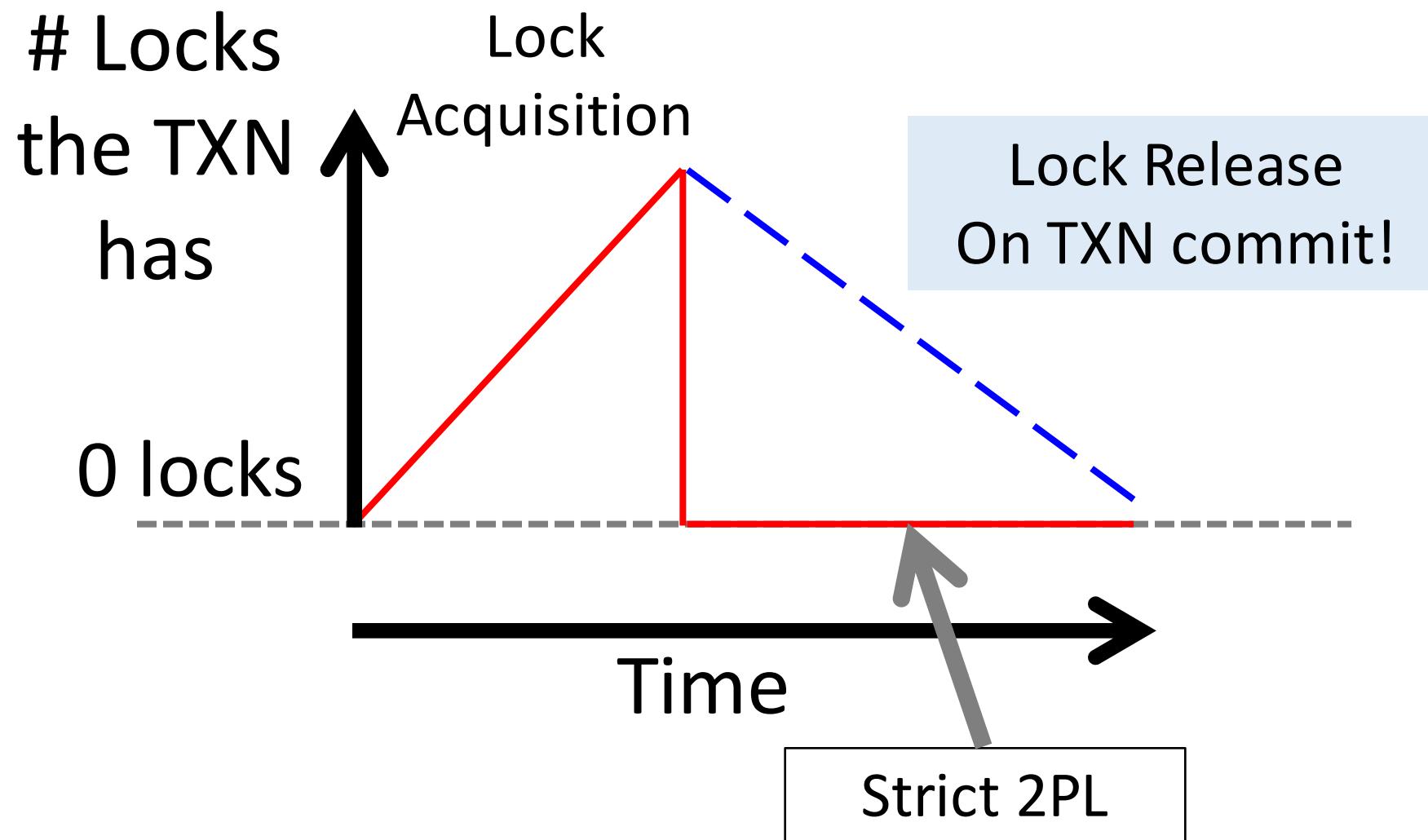
Strict Two-phase Locking (Strict 2PL) Protocol:

TXNs obtain:

- An **X (*exclusive*) lock** on object before **writing**.
 - If a TXN holds, no other TXN can get a lock (S or X) on that object.
- An **S (*shared*) lock** on object before **reading**
 - If a TXN holds, no other TXN can get *an X lock* on that object
- All locks held by a TXN are released when TXN completes.

Note: Terminology here- “exclusive”, “shared”- meant to be intuitive- no tricks!

Picture of 2-Phase Locking (2PL)



Strict 2PL

Theorem: Strict 2PL allows only schedules whose dependency graph is acyclic

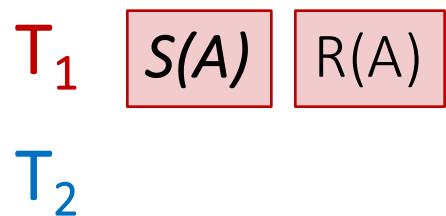
Proof Intuition: In strict 2PL, if there is an edge $T_i \rightarrow T_j$ (i.e. T_i and T_j conflict) then T_j needs to wait until T_i is finished – so *cannot* have an edge $T_j \rightarrow T_i$

Therefore, Strict 2PL only allows conflict serializable \Rightarrow serializable schedules

Strict 2PL

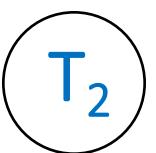
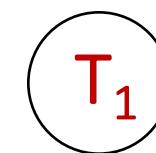
- If a schedule follows strict 2PL and locking, it is conflict serializable...
 - ...and thus serializable
 - ...and thus maintains isolation & consistency!
- Not all serializable schedules are allowed by strict 2PL.
- So let's use strict 2PL, what could go wrong?

Deadlock Detection: Example

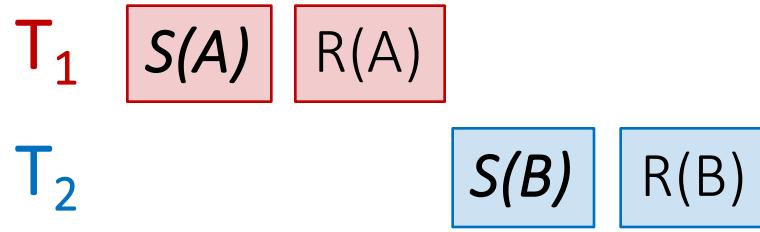


First, T_1 requests a shared lock on A to read from it

Waits-for graph:



Deadlock Detection: Example

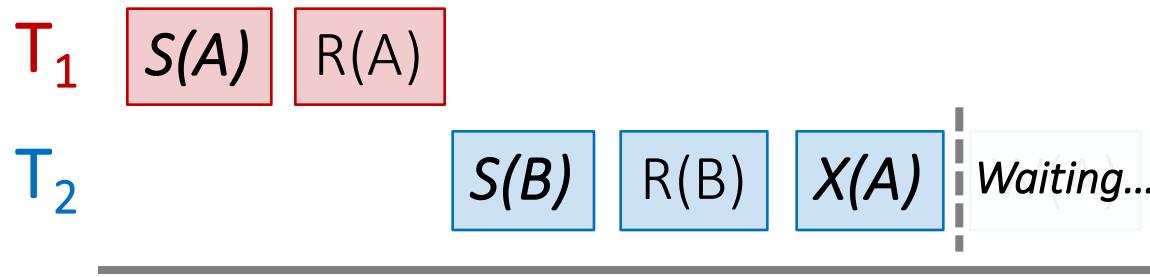


Waits-for graph:



Next, T_2 requests a shared lock on B to read from it

Deadlock Detection: Example

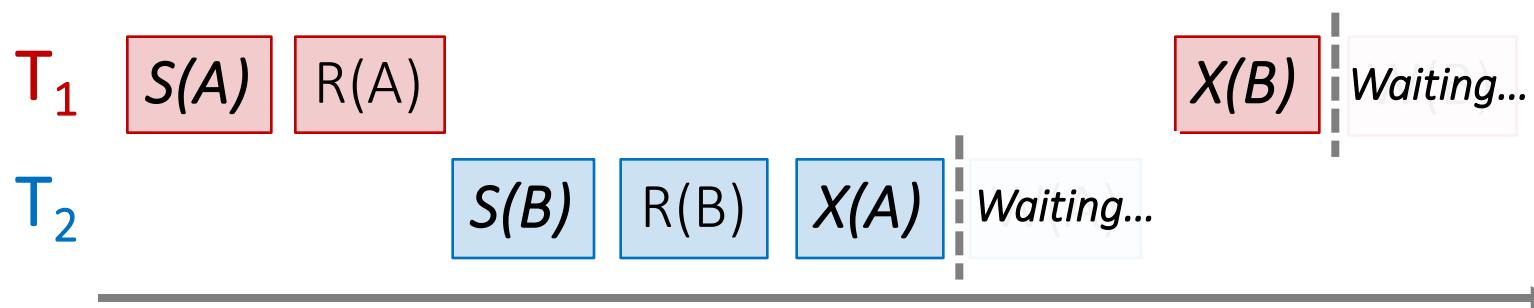


Waits-for graph:

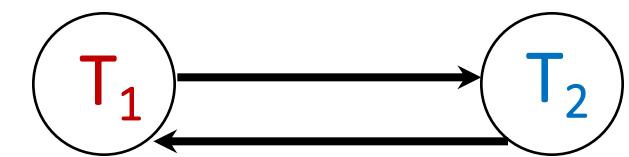


T_2 then requests an exclusive lock on A to write to it- now T_2 is waiting on T_1 ...

Deadlock Detection: Example



Waits-for graph:

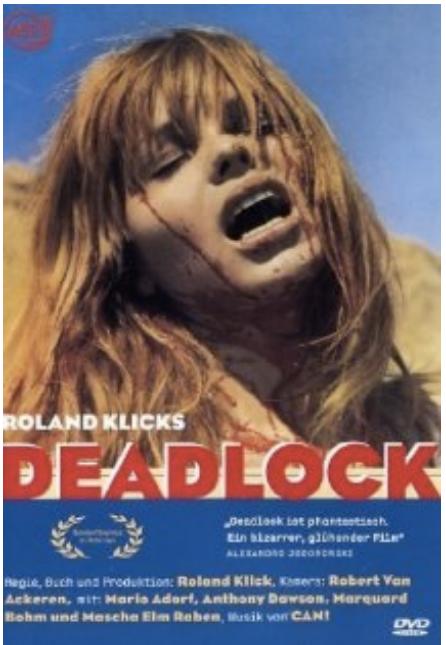


Cycle =
DEADLOCK

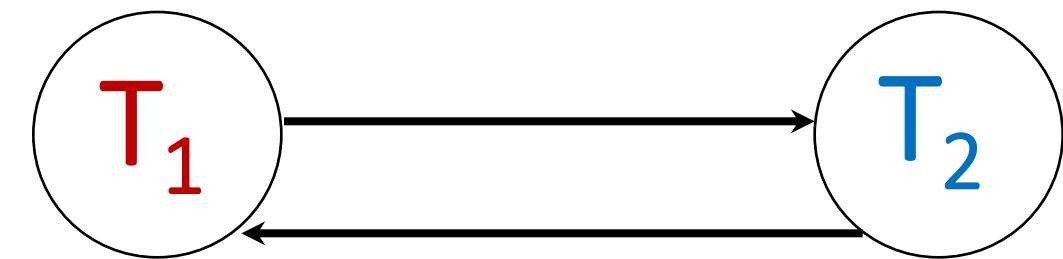
Finally, T₁ requests an exclusive lock on B to write to it- now T₁ is waiting on T₂... DEADLOCK!

sqlite3.OperationalError: database is locked

```
ERROR: deadlock detected
DETAIL: Process 321 waits for ExclusiveLock on tuple of
relation 20 of database 12002; blocked by process 4924.
Process 404 waits for ShareLock on transaction 689; blocked
by process 552.
HINT: See server log for query details.
```



The problem?
Deadlock!??!



NB: Also movie called wedlock
(deadlock) set in a futuristic prison...
I haven't seen either of them...

Deadlocks

- **Deadlock:** Cycle of transactions waiting for locks to be released by each other.
- Two ways of dealing with deadlocks:
 1. Deadlock prevention
 2. Deadlock detection

Deadlock Detection

- Create the **waits-for graph**:
 - Nodes are transactions
 - There is an edge from $T_i \rightarrow T_j$ if T_i is *waiting for T_j to release a lock*
- Periodically check for (**and break**) cycles in the waits-for graph

Summary

- *Last lecture:* Concurrency achieved by **interleaving TXNs** such that **isolation & consistency** are maintained
 - We formalized a notion of **serializability** that captured such a “good” interleaving schedule
- We defined **conflict serializability**, which implies serializability
 - There are other, more general issues!
- **Locking** allows only conflict serializable schedules
 - If the schedule completes- it may deadlock!



Candy Break

2. The Buffer

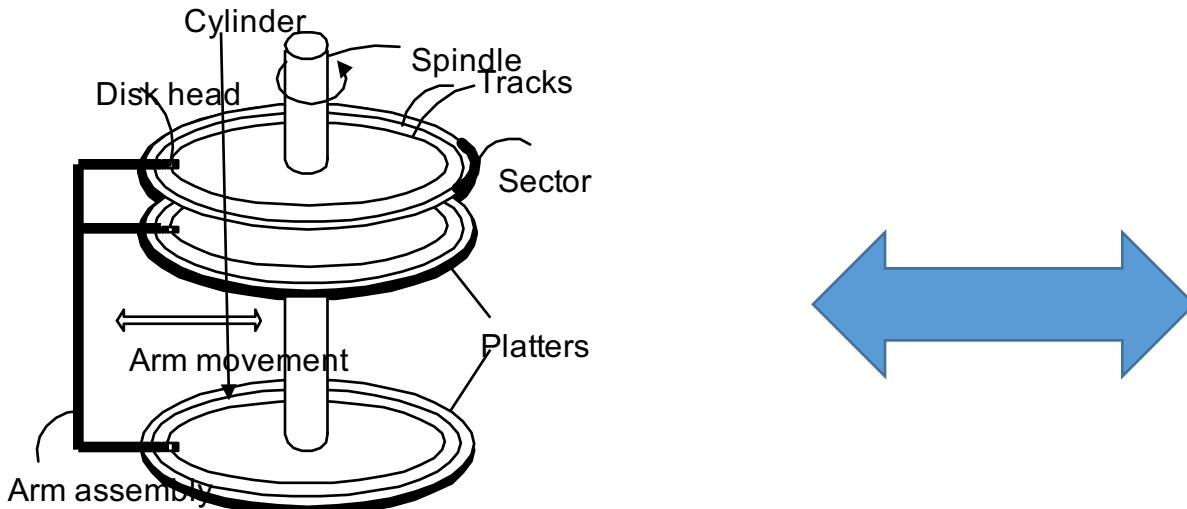
Transition to Mechanisms

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)

What you will learn about in this section

1. RECAP: Storage and memory model
2. Buffer primer

High-level: Disk vs. Main Memory



Disk:

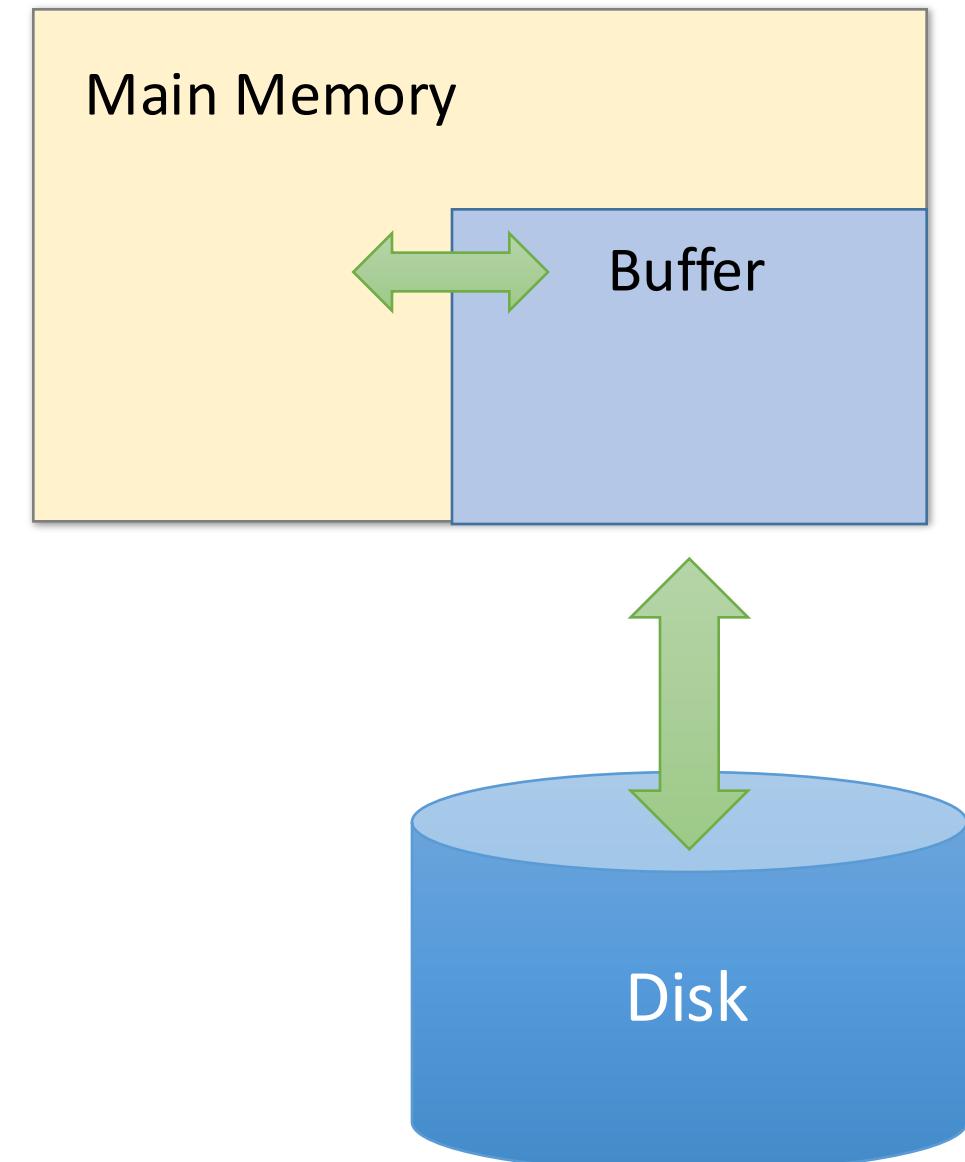
- **Slow:** Sequential *block* access
 - Read a blocks (not byte) at a time, so sequential access is cheaper than random
 - **Disk read / writes are expensive!**
- **Durable:** We will assume that once on disk, data is safe!
- **Cheap**

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!

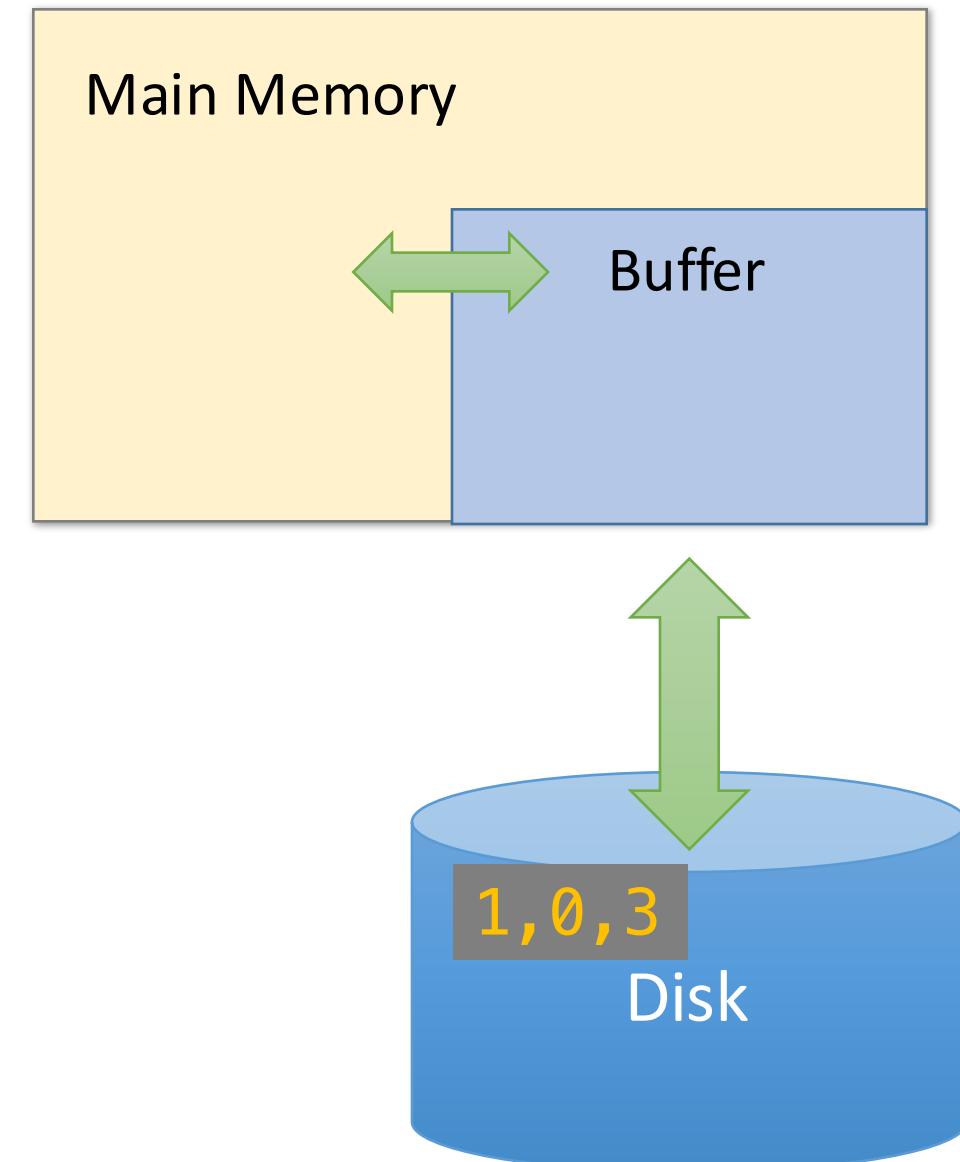
The Buffer

- 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

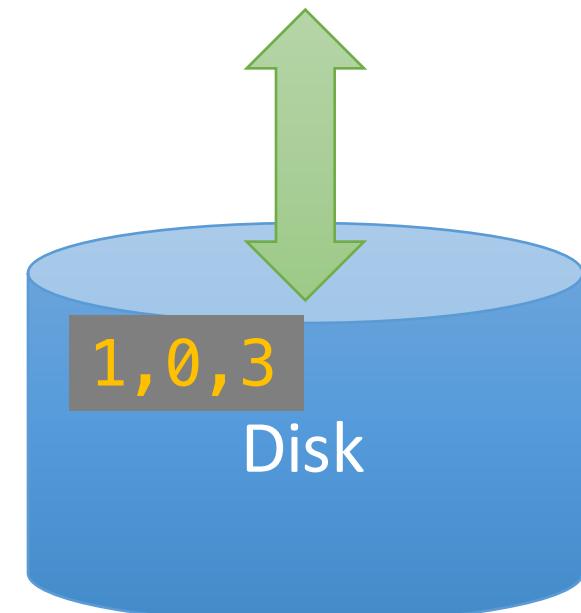
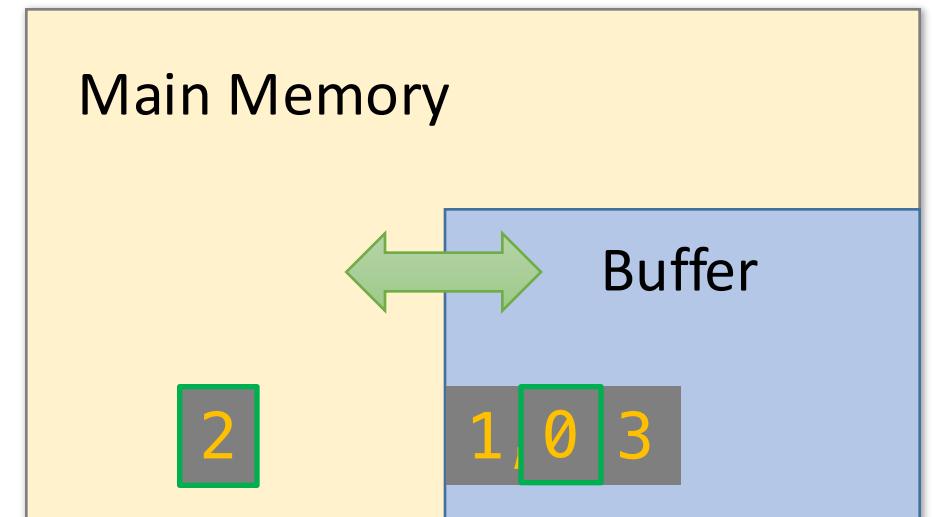
- In this class: We'll consider a buffer located in **main memory** that operates over **pages** and **files**:
 - **Read(page)**: Read page from disk -> buffer *if not already in buffer*



The (Simplified) Buffer

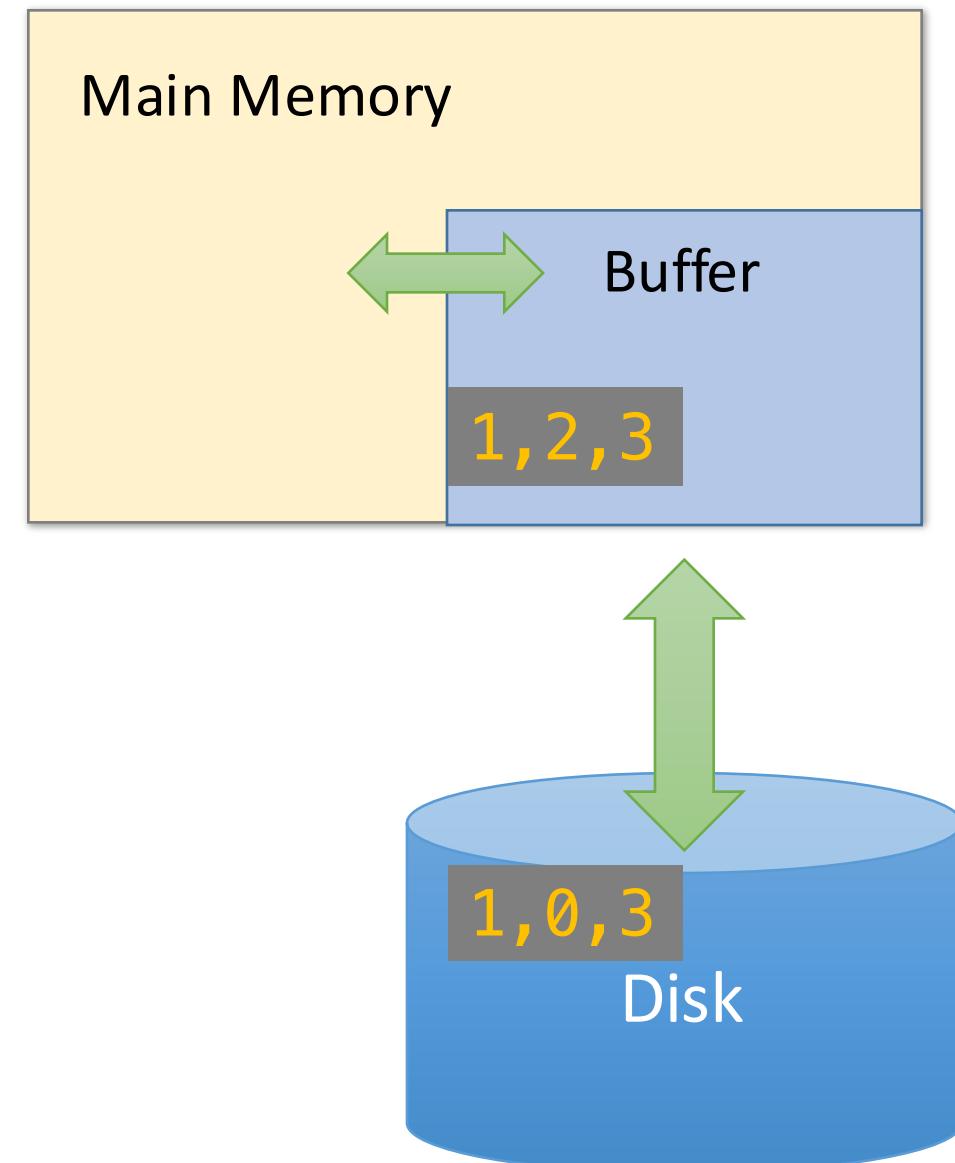
- In this class: We'll consider a buffer located in **main memory** that operates over **pages** and **files**:
 - **Read(page)**: Read page from disk -> buffer *if not already in buffer*

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



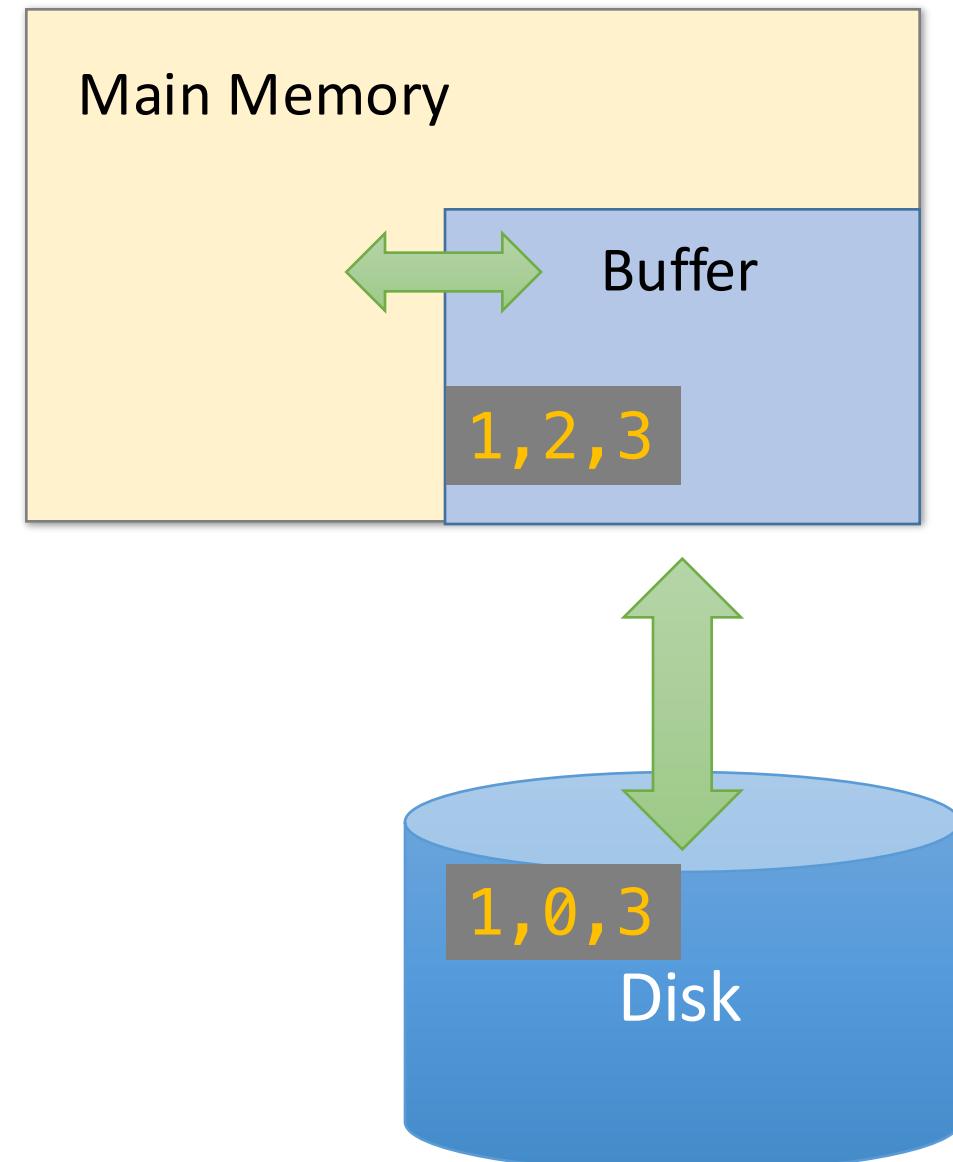
The (Simplified) Buffer

- 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



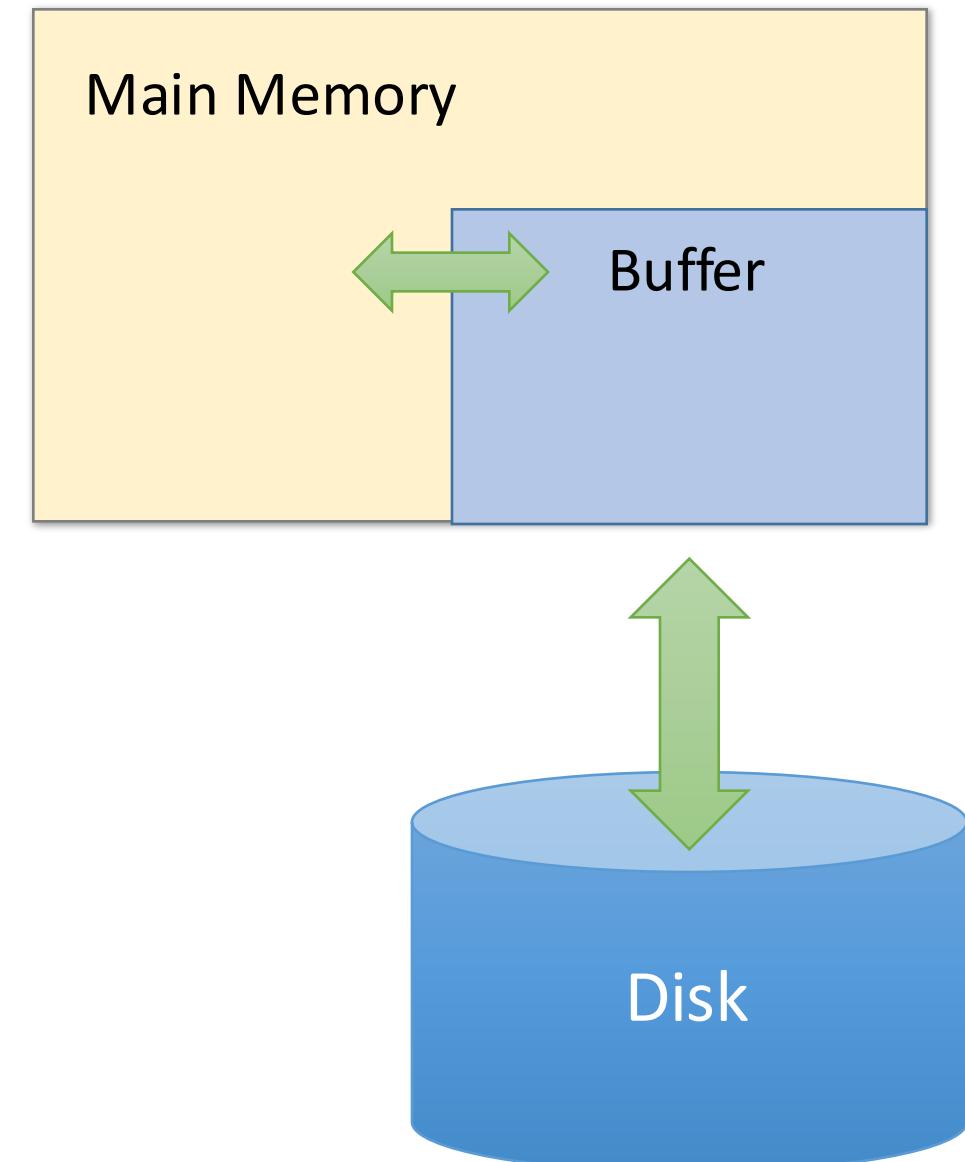
The (Simplified) Buffer

- 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



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.

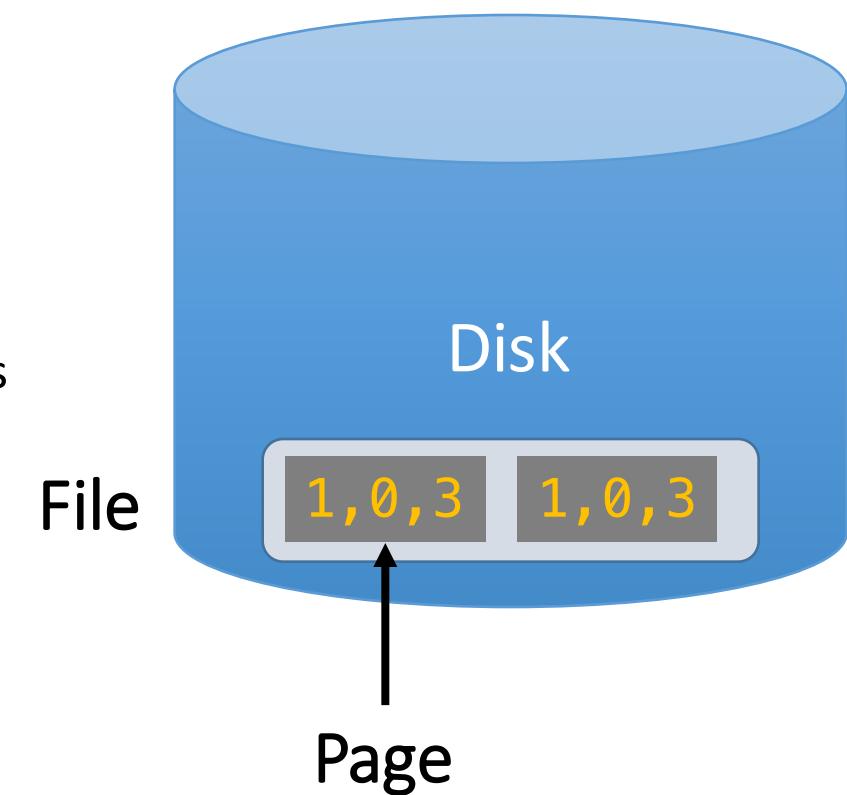


The Buffer Manager

- 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

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 (see 346)
- And a file is a ***variable-length list*** of pages
 - Interface: create / open / close; next_page(); etc.



2. External Merge & Sort

What you will learn about in this section

1. External Merge- Basics
2. External Merge- Extensions
3. External Sort

External Merge

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?

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:

$$A_1 \leq A_2 \leq \dots \leq A_N$$

$$B_1 \leq B_2 \leq \dots \leq B_M$$

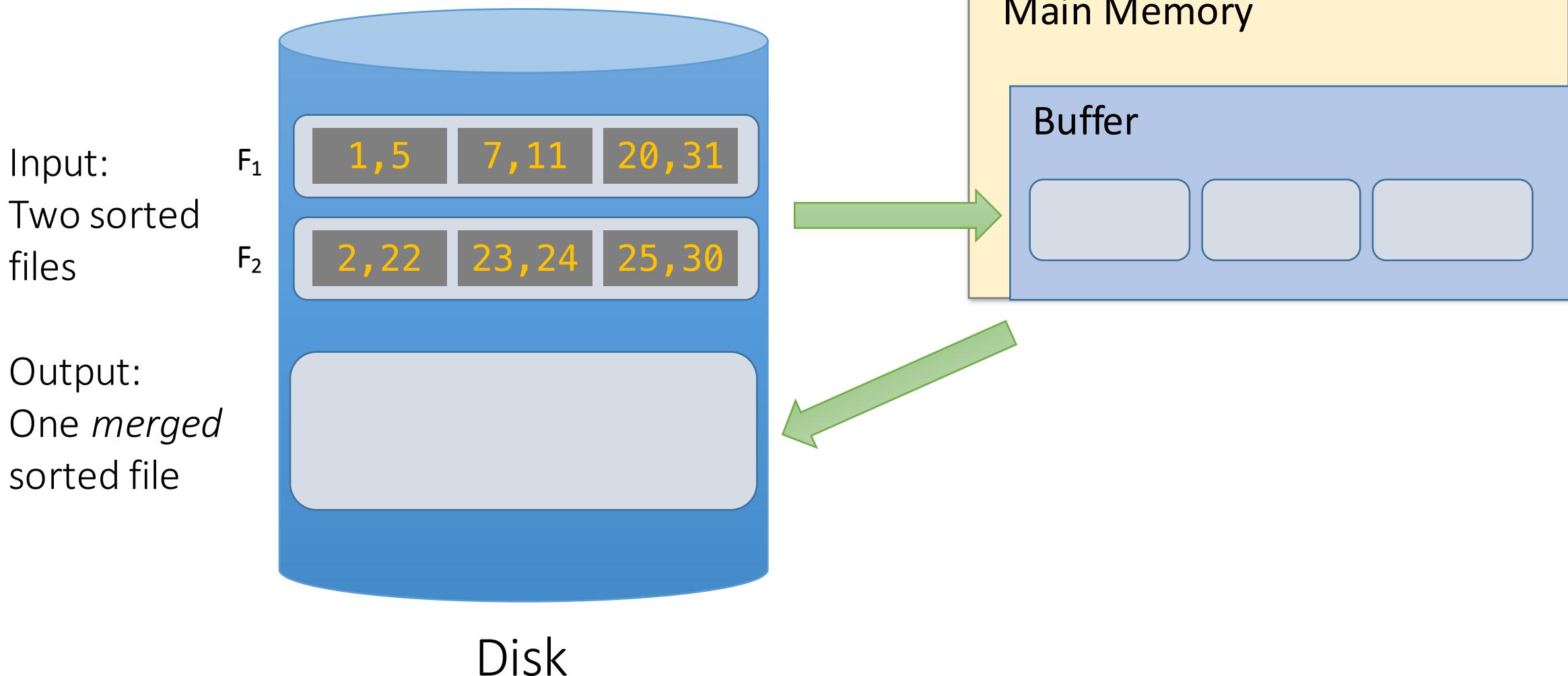
Then:

$$\text{Min}(A_1, B_1) \leq A_i$$

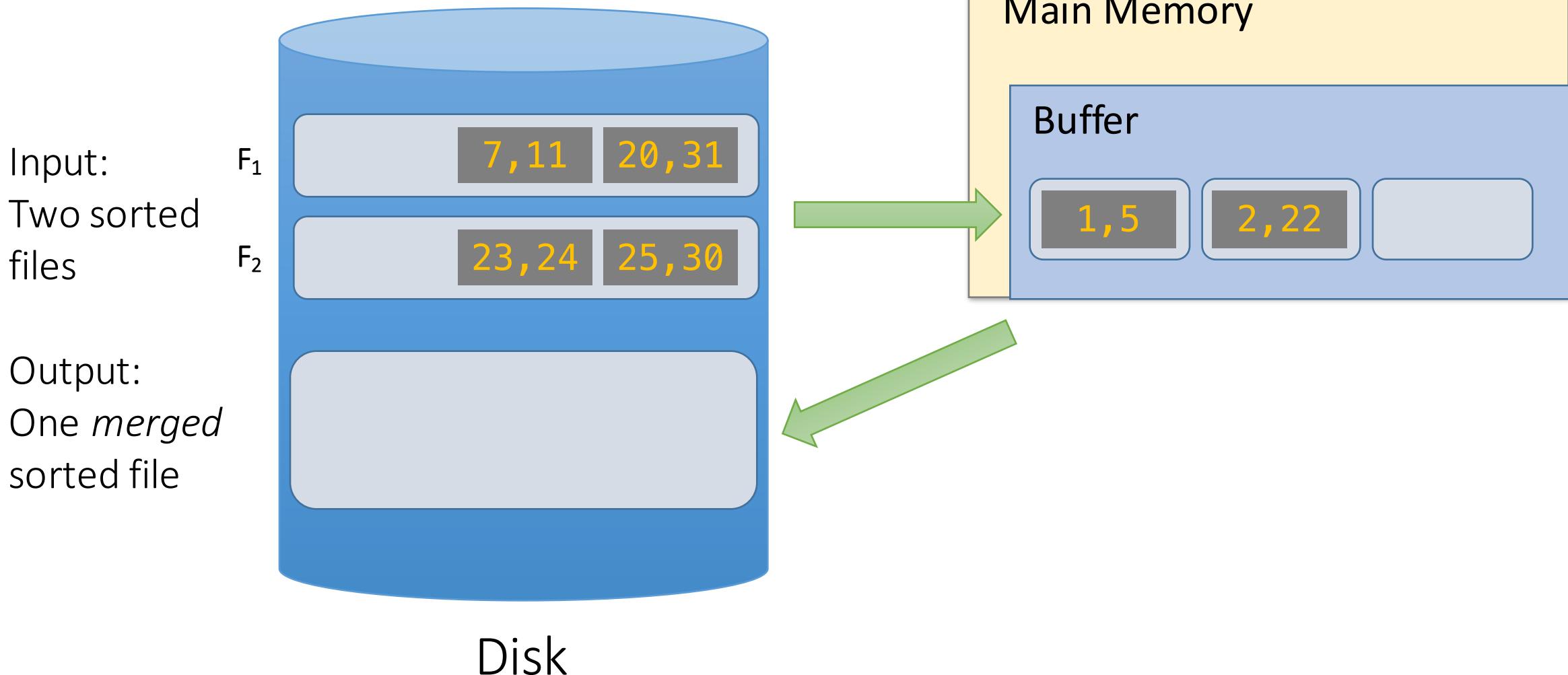
$$\text{Min}(A_1, B_1) \leq B_j$$

for $i=1 \dots N$ and $j=1 \dots M$

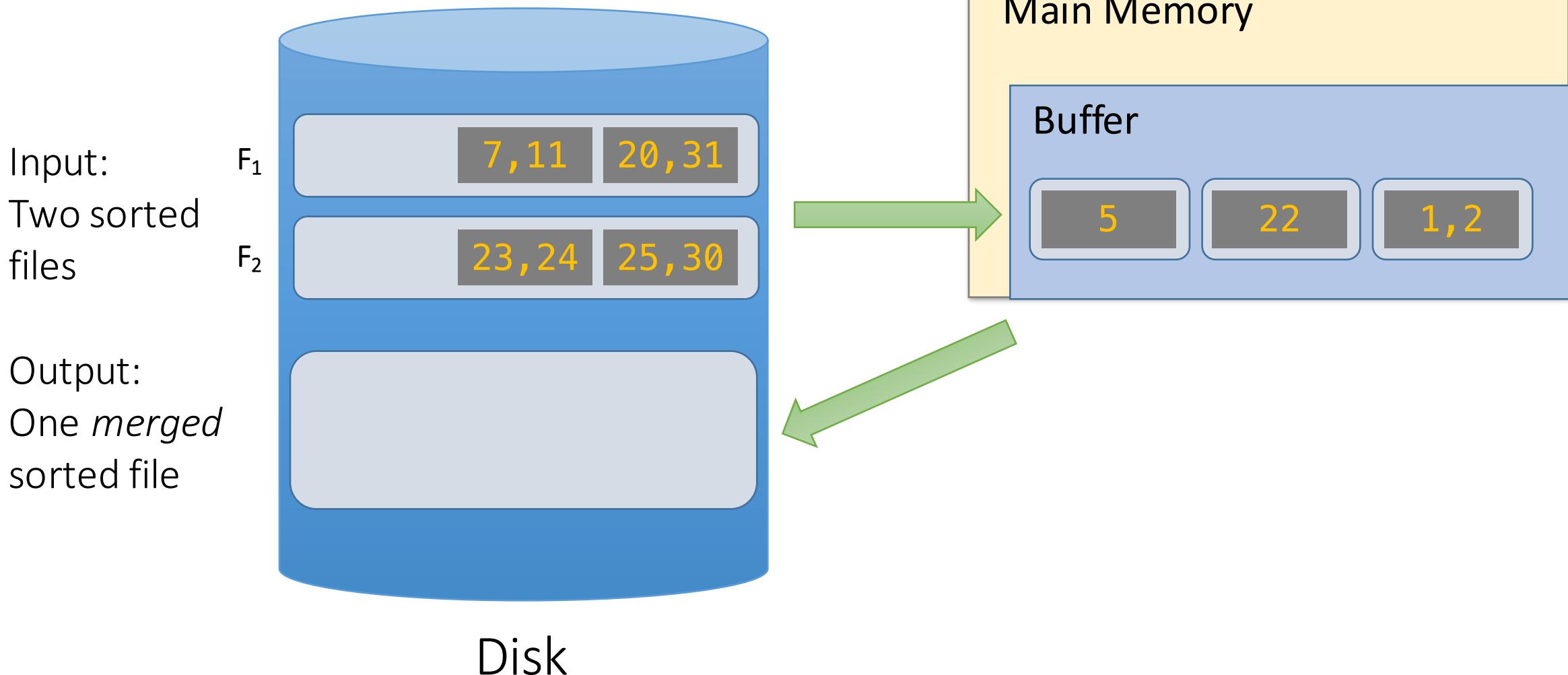
External Merge Algorithm



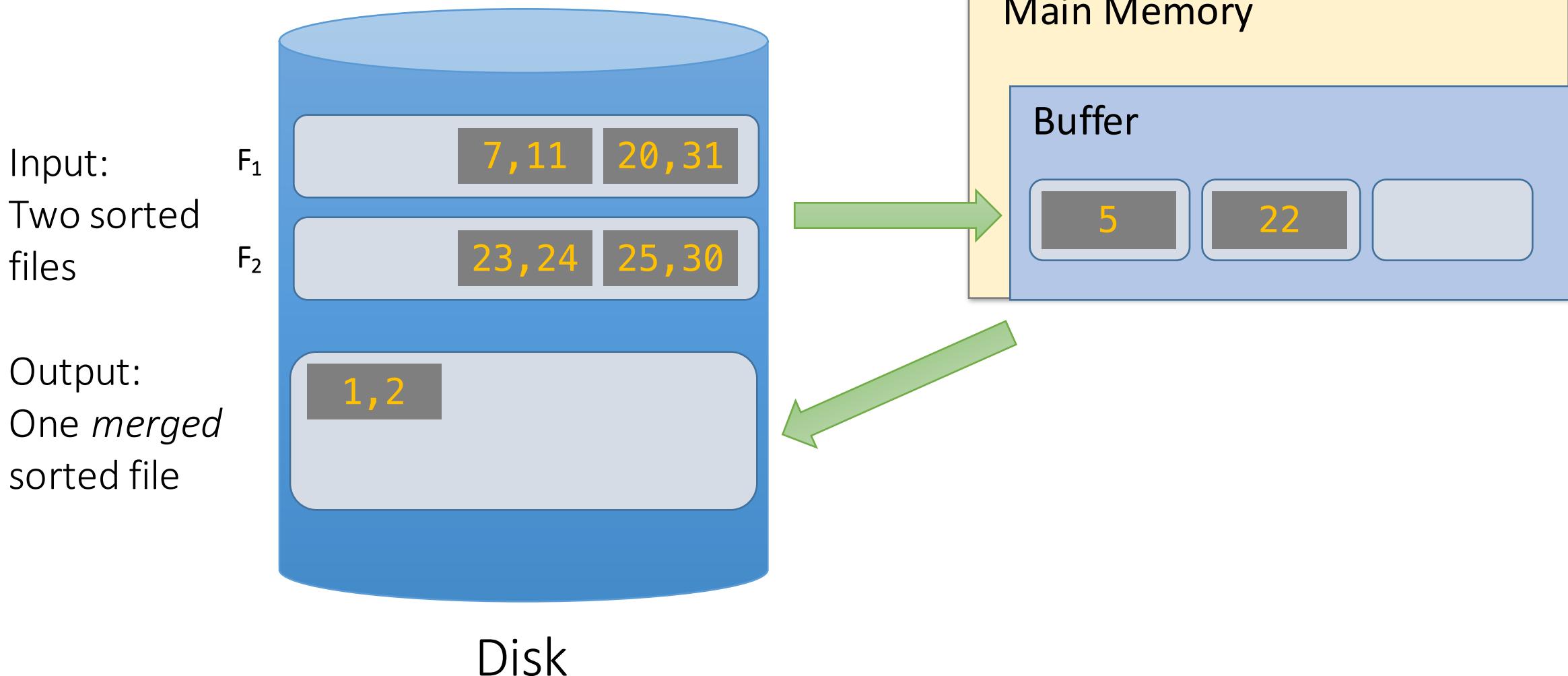
External Merge Algorithm



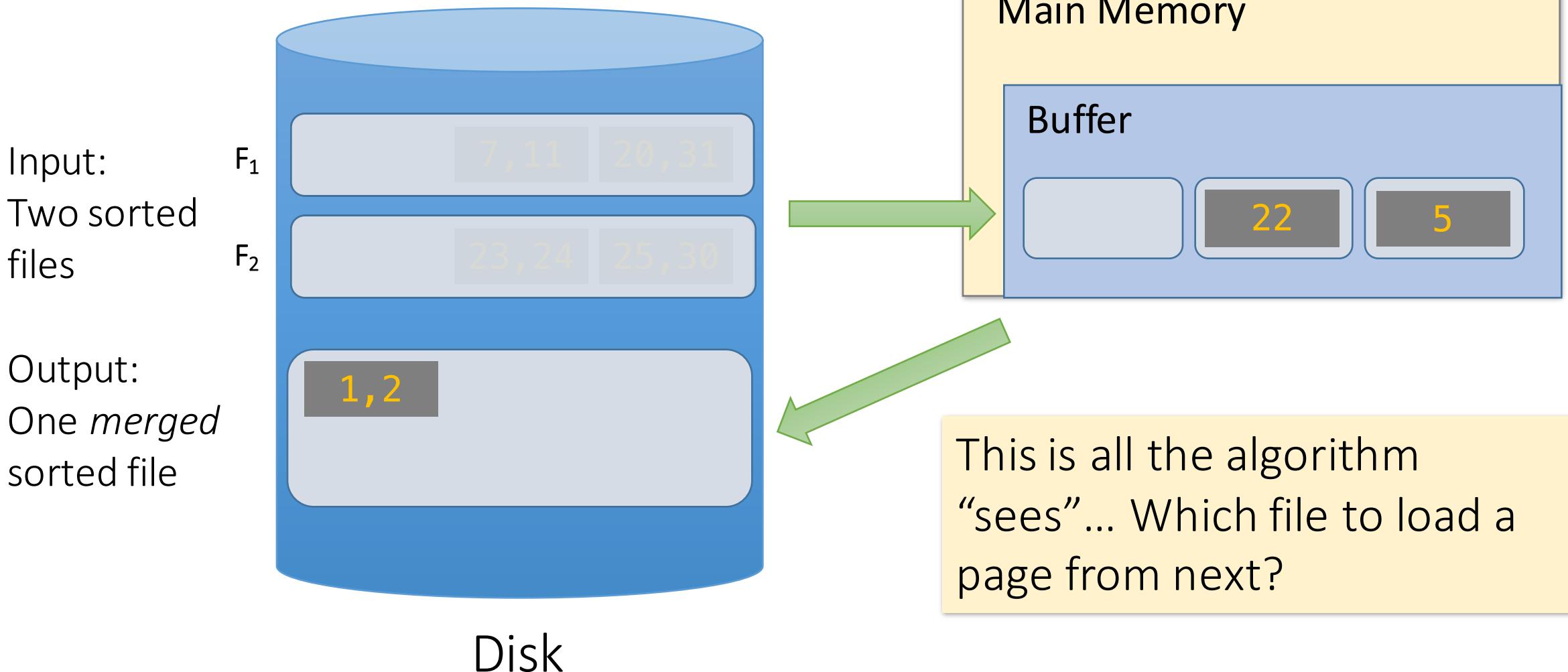
External Merge Algorithm



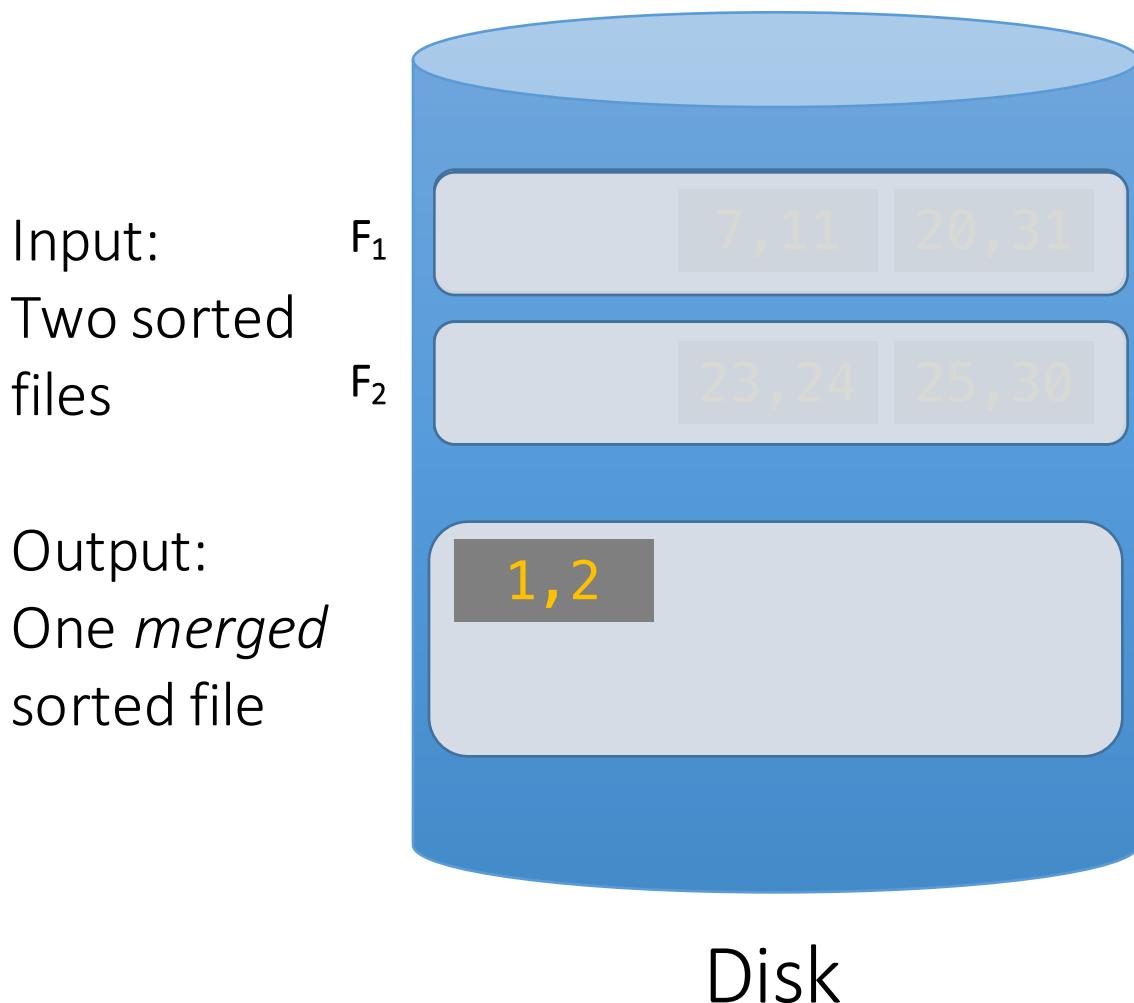
External Merge Algorithm



External Merge Algorithm

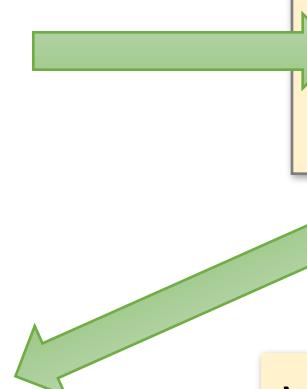


External Merge Algorithm



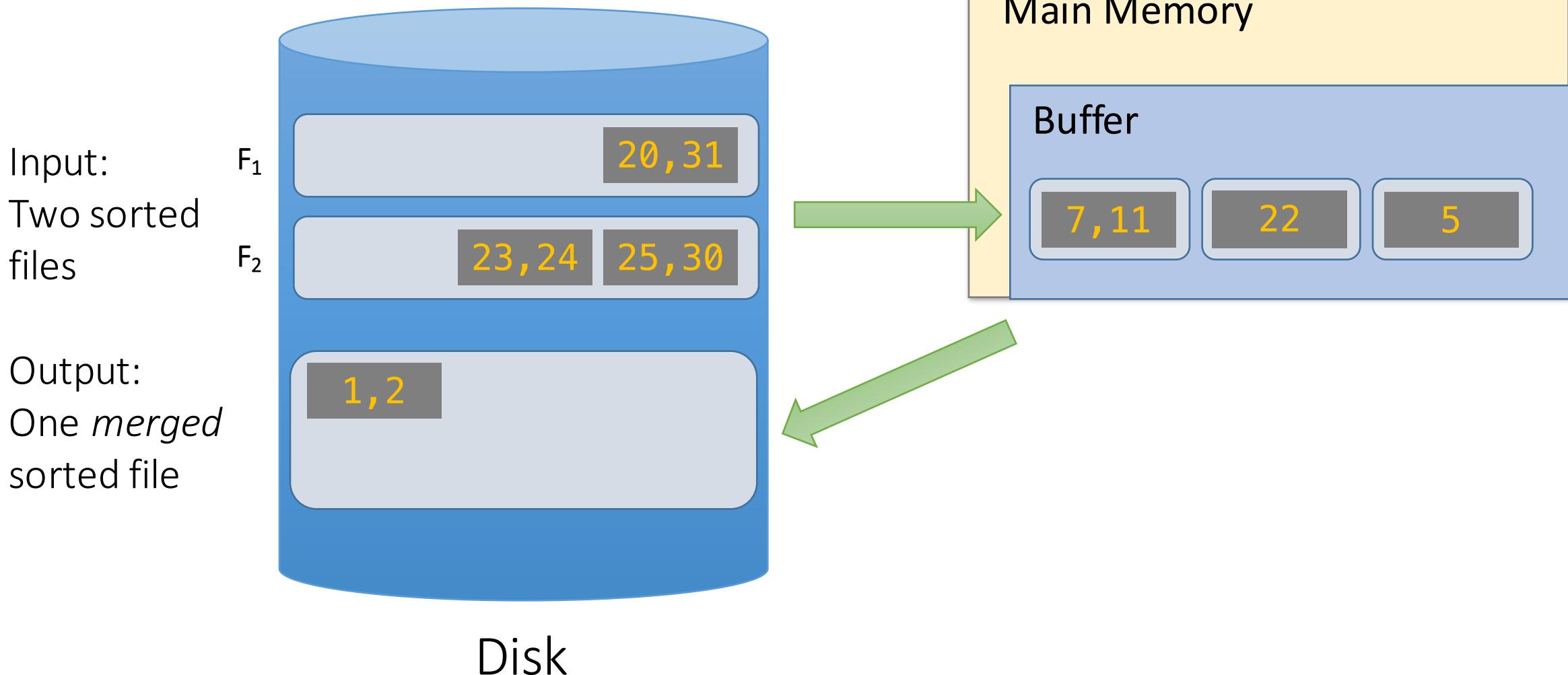
Main Memory

Buffer

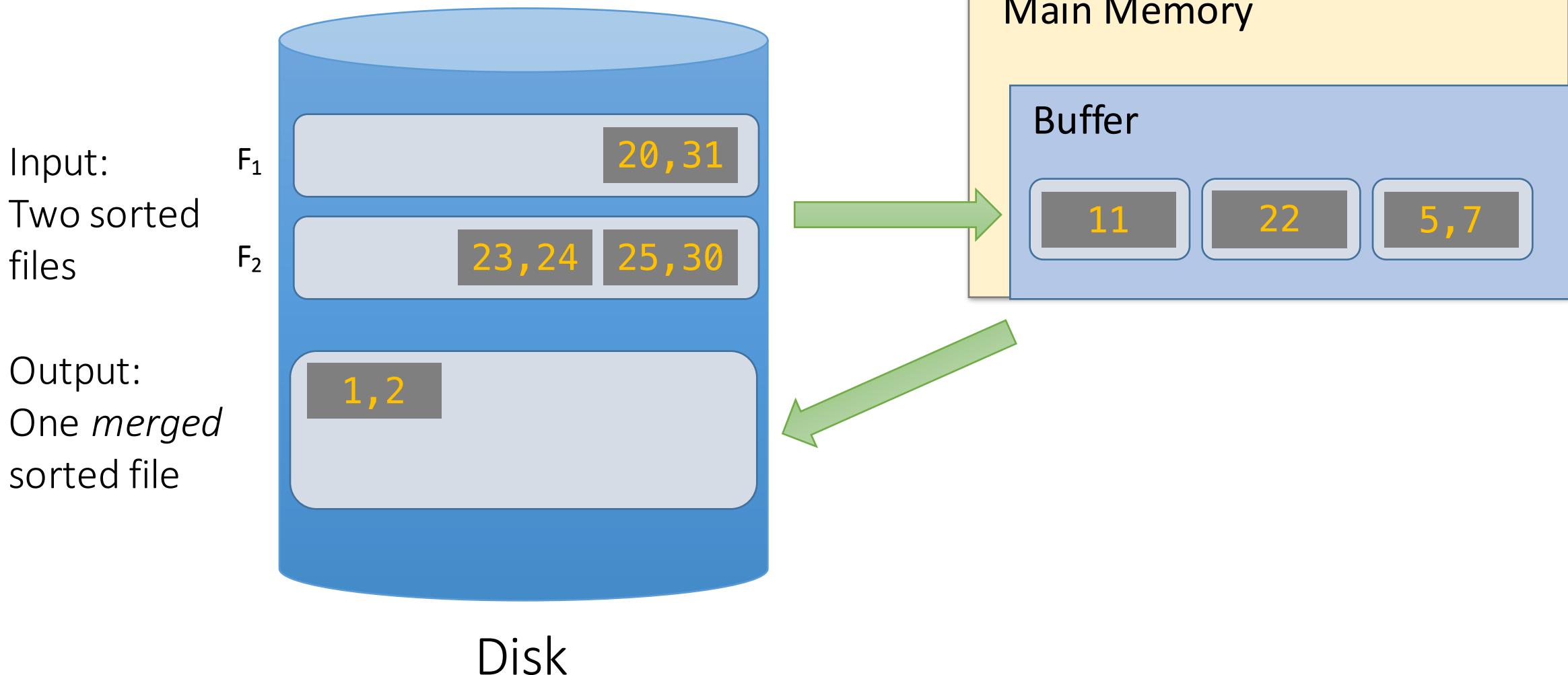


We know that F_2 only contains values ≥ 22 ... so we should load from F_1 !

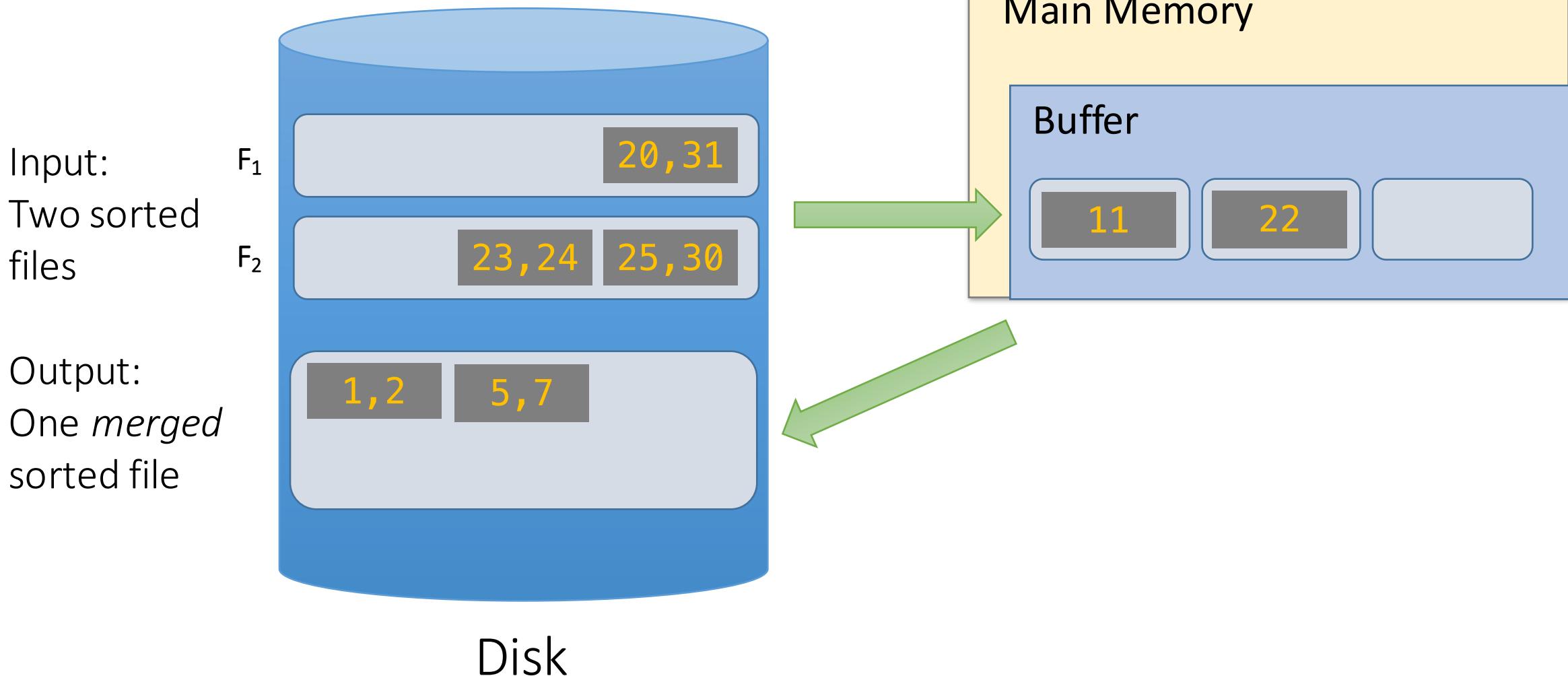
External Merge Algorithm



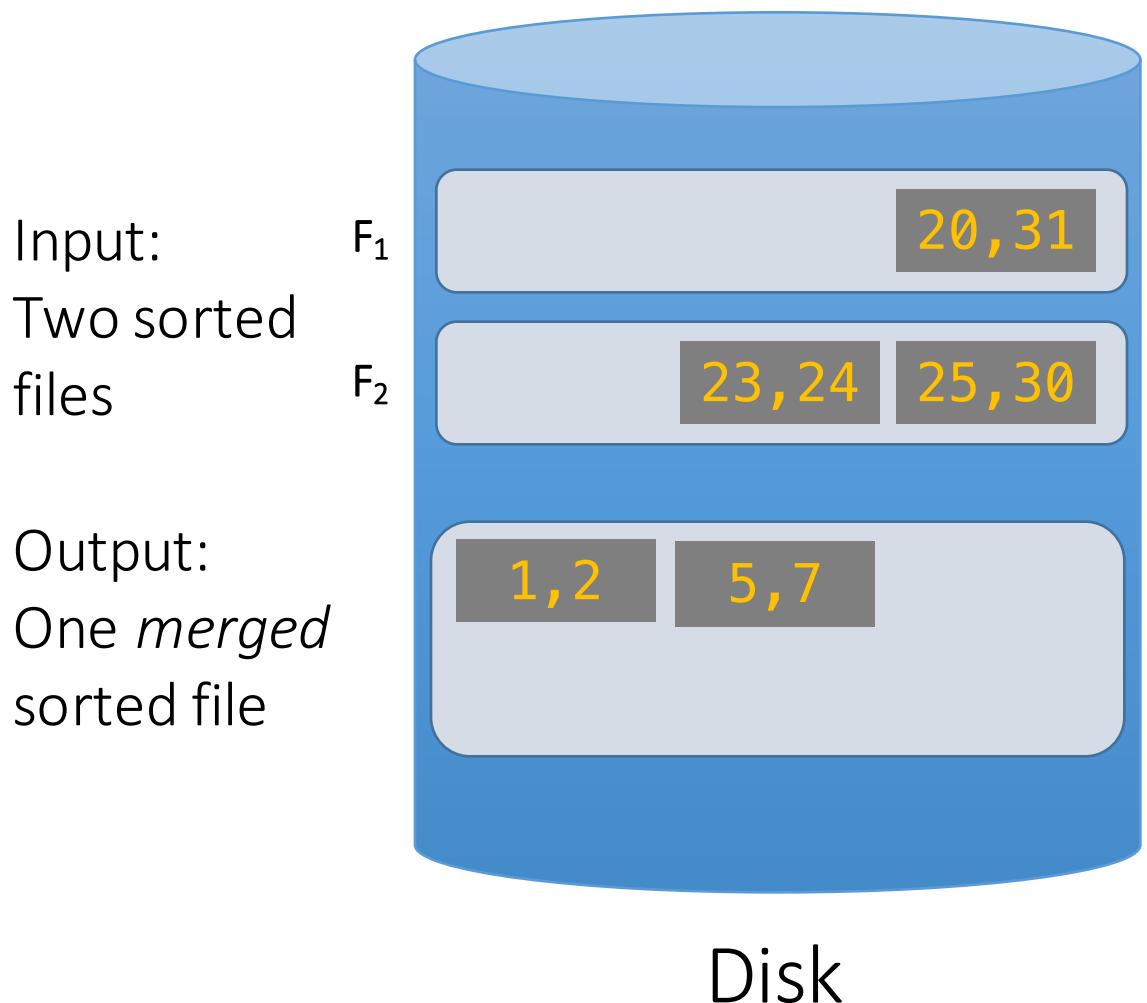
External Merge Algorithm



External Merge Algorithm



External Merge Algorithm



Main Memory

Buffer

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?

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...

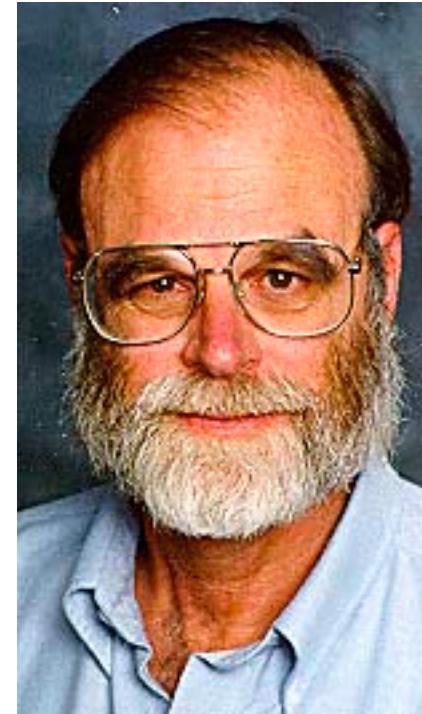
- Sorting useful for eliminating *duplicate copies* in a collection of records (Why?)
- Sorting is first step in *bulk loading* B+ tree index.
- *Sort-merge* join algorithm involves sorting

Next lecture!

The lecture
after next...

Do people care?

<http://sortbenchmark.org>



Sort benchmark bears his name

Simplified External Sorts.

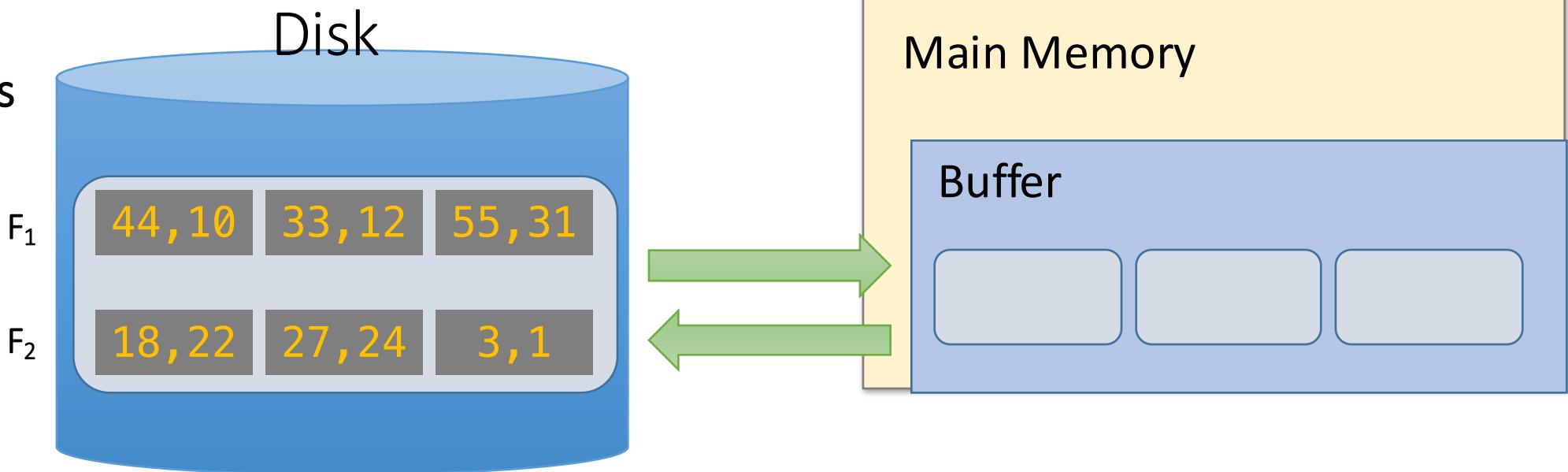
So how do we sort big files?

1. Split into chunks small enough to **sort in memory**
2. **Merge** each pair of sorted chunks *using the external merge algorithm*
3. **Merge** pairs of these resulting chunks...
4. **Repeat** until all merged (and sorted!)

External Merge Sort Algorithm

Example:

- 3 Buffer pages
- 6-page file

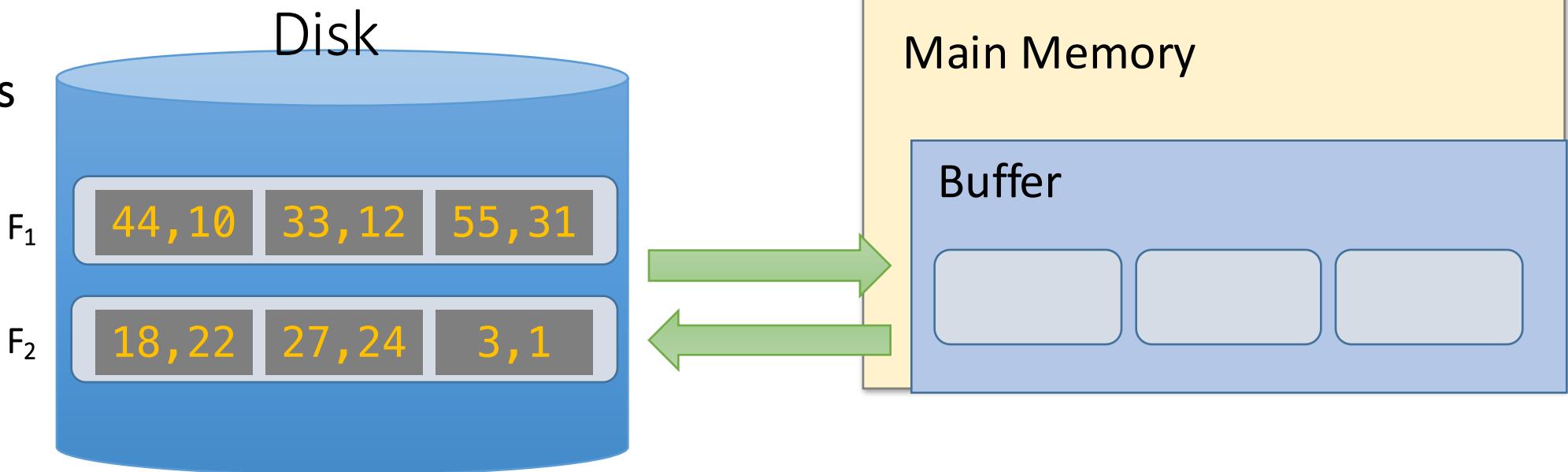


1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm

Example:

- 3 Buffer pages
- 6-page file

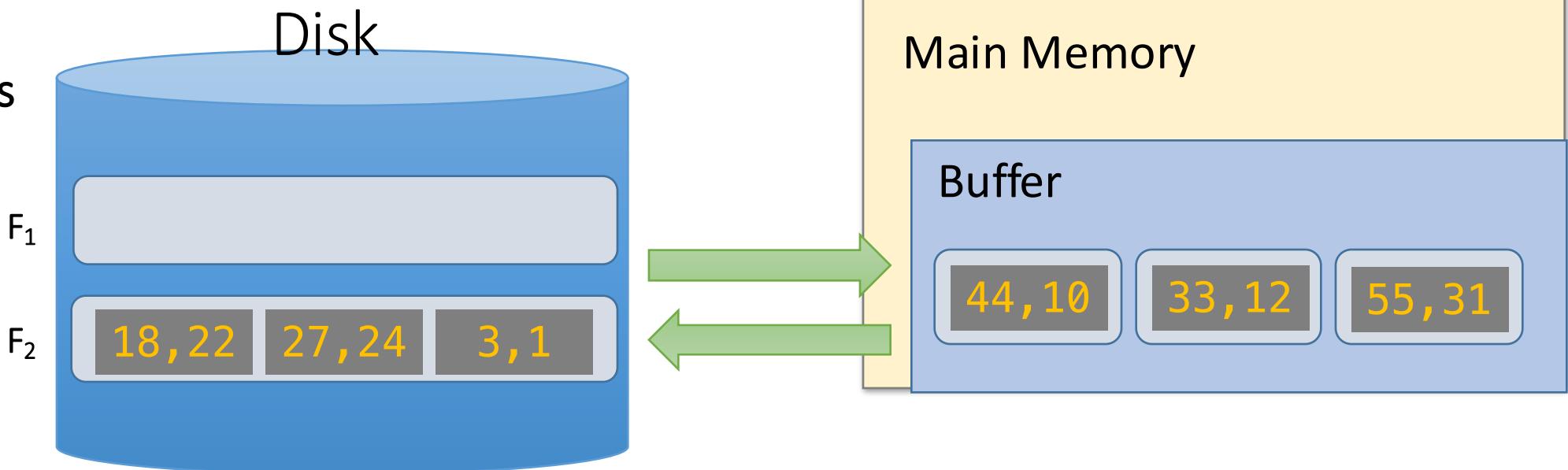


1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm

Example:

- 3 Buffer pages
- 6-page file

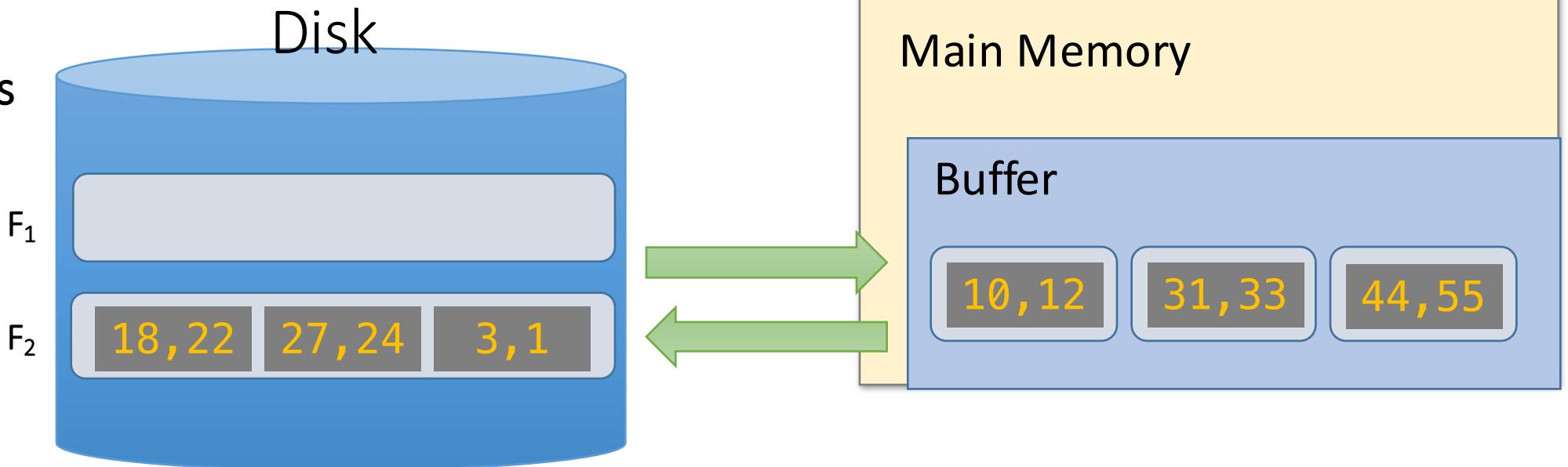


1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm

Example:

- 3 Buffer pages
- 6-page file

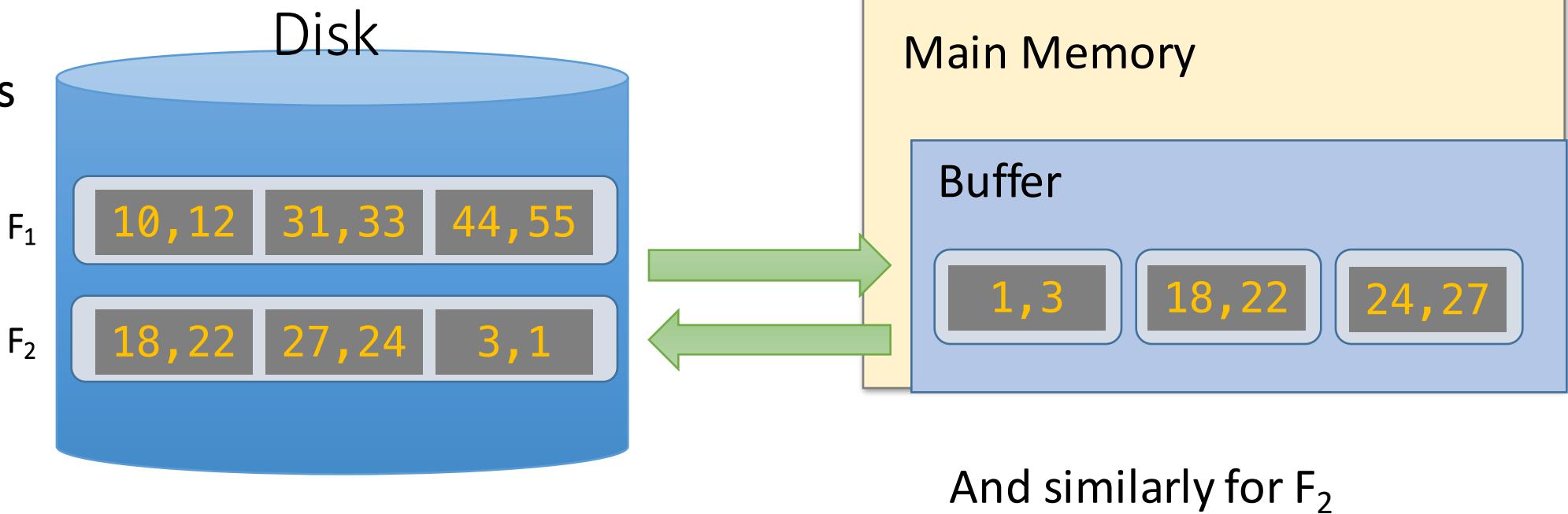


1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm

Example:

- 3 Buffer pages
- 6-page file

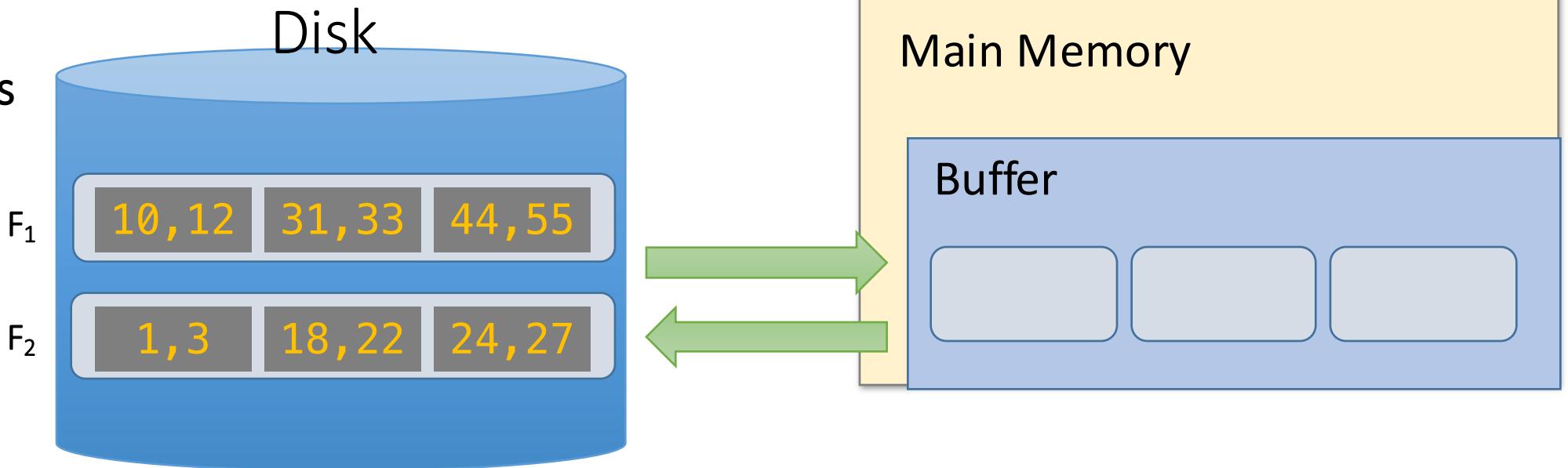


1. Split into chunks small enough to **sort in memory**

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. = $1 \text{ R} + 1 \text{ W}$ for each file = $2 * 2 = 4$ IO operations
2. Merge each pair of sorted chunks ***using the external merge algorithm***
 1. = $2 * (3 + 3) = 12$ IO operations

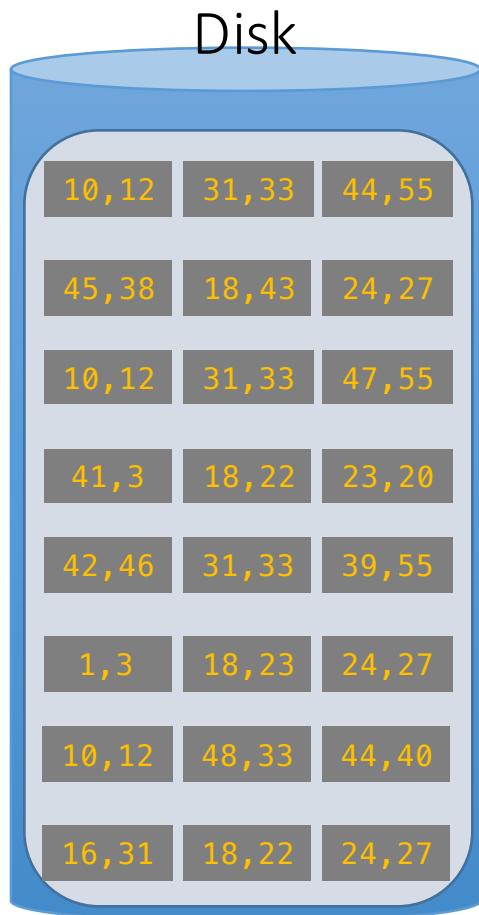
Calculating IO Cost

For B buffer pages, $2*B$ page file:

1. Split into two B-page files and **sort in memory**
 1. = 1 R + 1 W for each file = $2*2 = 4$ IO operations
2. Merge each pair of sorted chunks *using the external merge algorithm*
 1. = $2*(B + B) = 4B$ IO operations

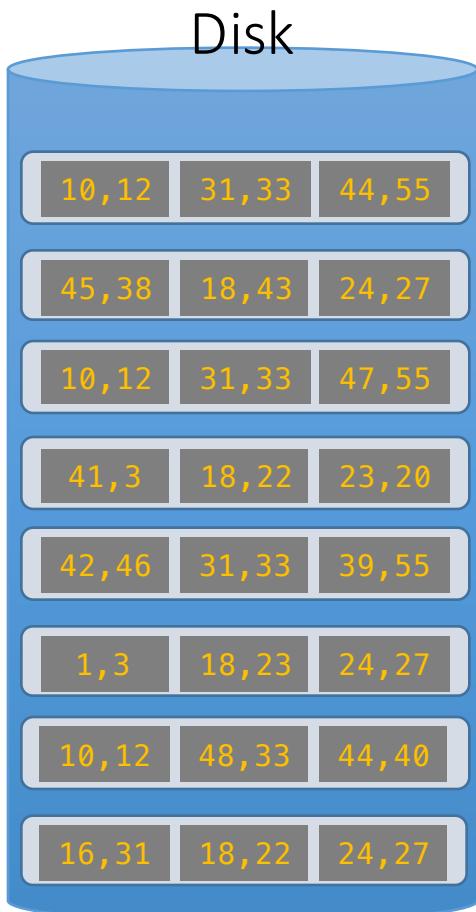
$$= 4*(B+1) \text{ IO Operations}$$

Running External Merge Sort on Larger Files



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

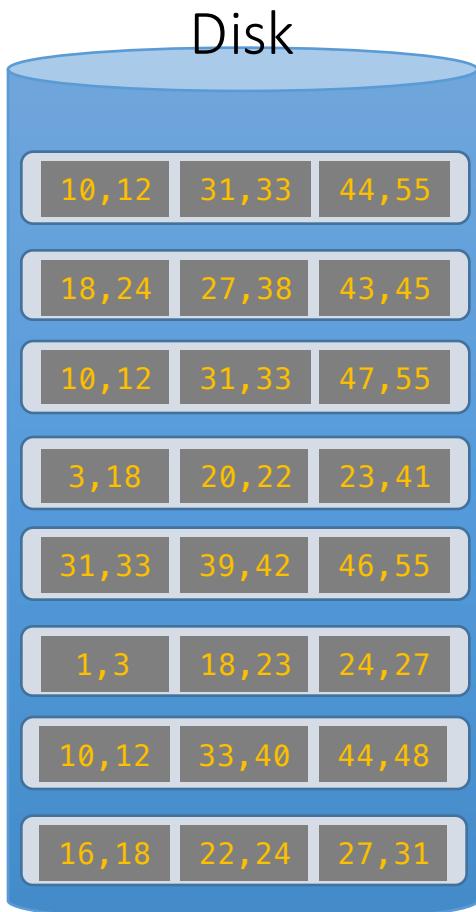
Running External Merge Sort on Larger Files



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

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

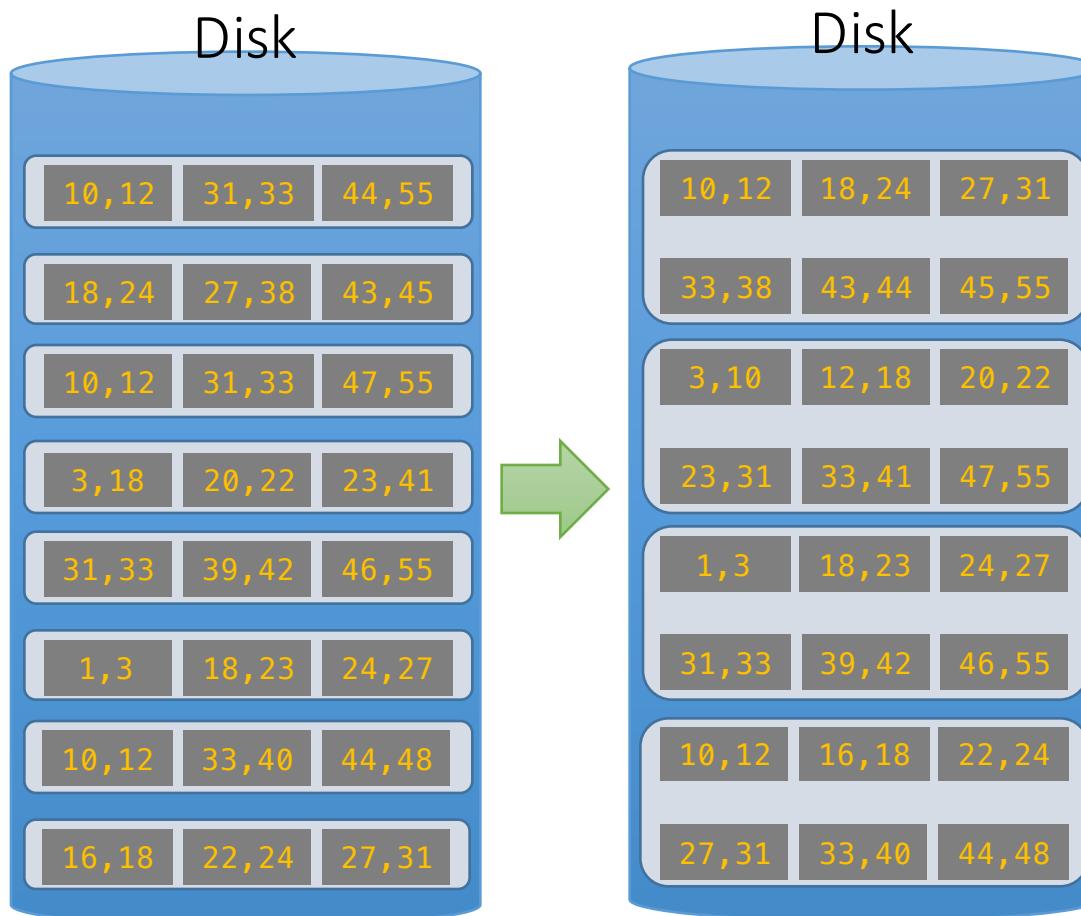
Running External Merge Sort on Larger Files



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

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

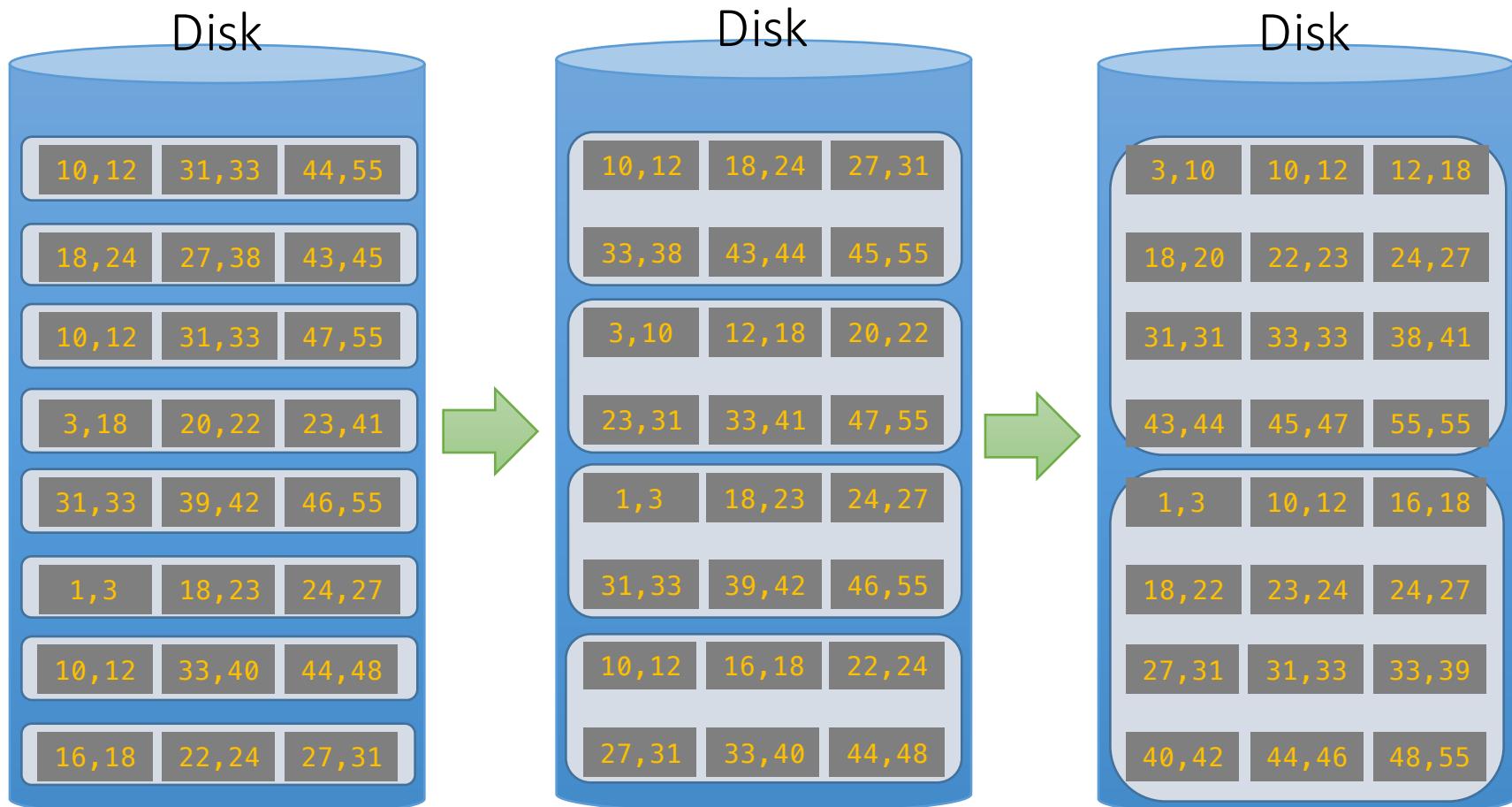
Running External Merge Sort on Larger Files



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

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

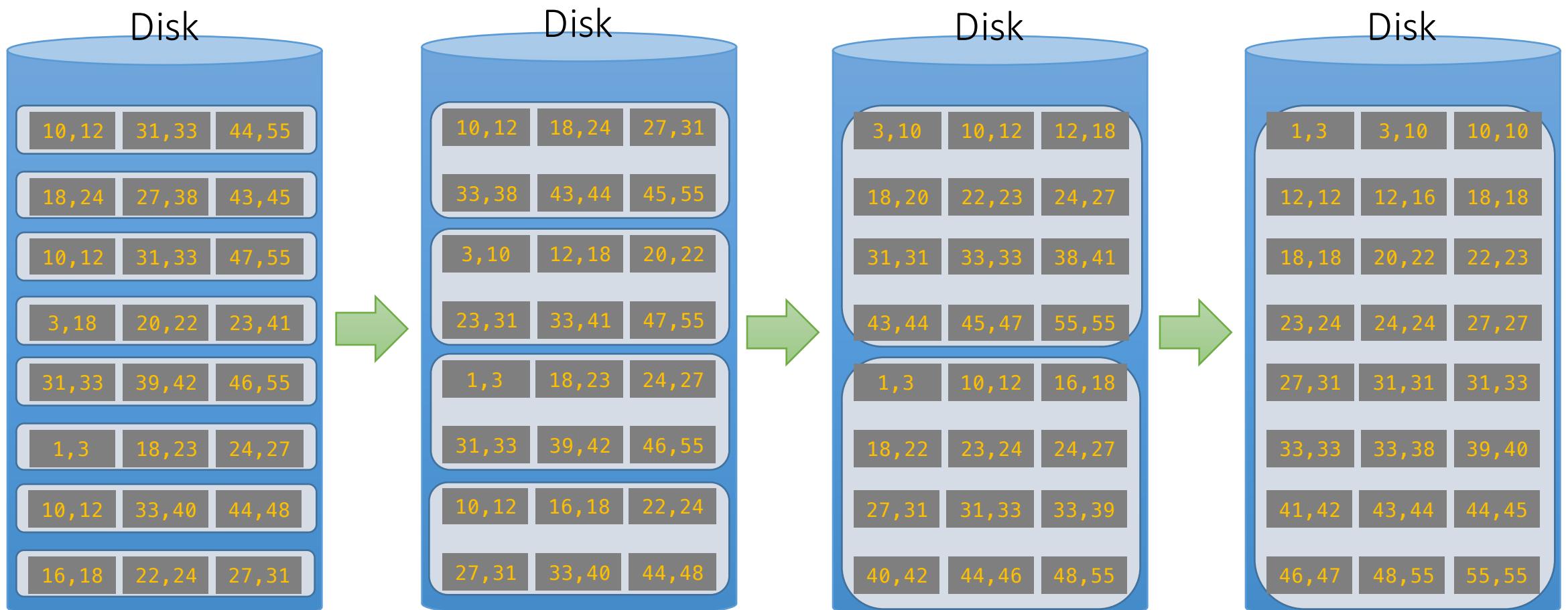
Running External Merge Sort on Larger Files



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

3. And repeat...

Running External Merge Sort on Larger Files

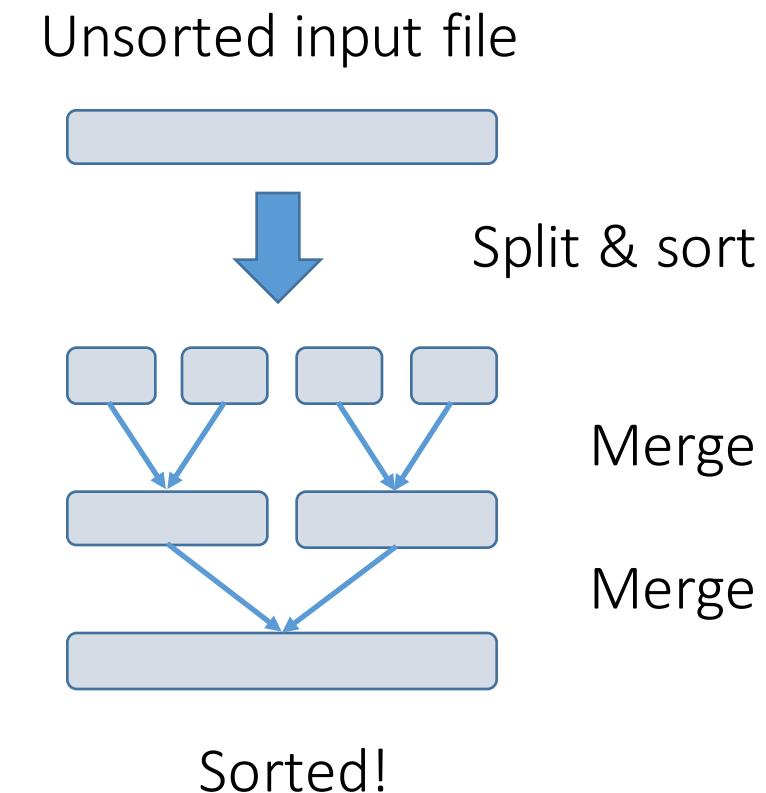


4. And repeat!

Simplified 3-page Buffer Version

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

- We merge **$N/2$ pairs of length 1 page**
 - Each one takes $2*(1+1) = 4$ IO
- Then we merge **$N/4$ pairs of length 2 pages**
 - Each one takes $2*(2+2) = 8$ IO
- ... In general, each step will take **$2N$ IO!**



For N pages, there will be $\lceil \log_2 N \rceil + 1$ steps $\rightarrow 2N * (\lceil \log_2 N \rceil + 1)$ total IO cost!

Using $B+1$ buffer pages to reduce passes

Goal: Reduce *the number of passes*.

- Suppose we have $B+1$ buffer pages and a file with N pages.
- Each pass still takes $2N$ IOs.

IO Cost:

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



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



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

1. **Increase length of initial runs.** Sort B at a time!
 - From N of length 1 to $N/(B+1)$ runs each of length B

2. **Perform a B way merge.**

- Given M runs each of length L , produce M/B runs of length LB .

Summary

- Basics of IO and buffer management.
 - See notebook for more fun! (Learn about *sequential flooding*)
- 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

Number of Passes of External Sort

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

Engineer's rule of thumb:
You sort in 3 passes

Even longer initial runs

Suppose we have $B+1$ buffer pages.

- Produce longer runs by “merging” as we read.
- Read B buffer pages and merge them.
- **Twist:** when input page is empty, read in a new page.
- **Issue:** Some values on new page may be smaller than already output value.
 - Can’t output, these values call them **frozen**.
- Try to consolidate frozen values in pages, to free up more pages.
 - **Alternatively:** Just use a priority queue—called tournament sort!
- **How long are the runs if input file is sorted in desired order?**
 - Nothing is frozen!
- **How long are the runs if input file is sorted in reverse order?**
 - All new values are frozen

Engineers approximation: Runs will have $\sim 2(B+1)$ length.

Double Buffering

Which IOs are sequential in sorting?

- Initial runs, both reading and writing are sequential writes.
- **Merge phase:** writing is sequential, reading is **not!**

Can Improve performance by using bigger buffers to “prefetch” or “double buffer” to hide reads in merge-phase.

- **Prefetch:** Hide latency
- **Bigger Batch Sizes:** Amortize expensive random reads and writes, but reduce available buffer pages.

Surprisingly useful trick.