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

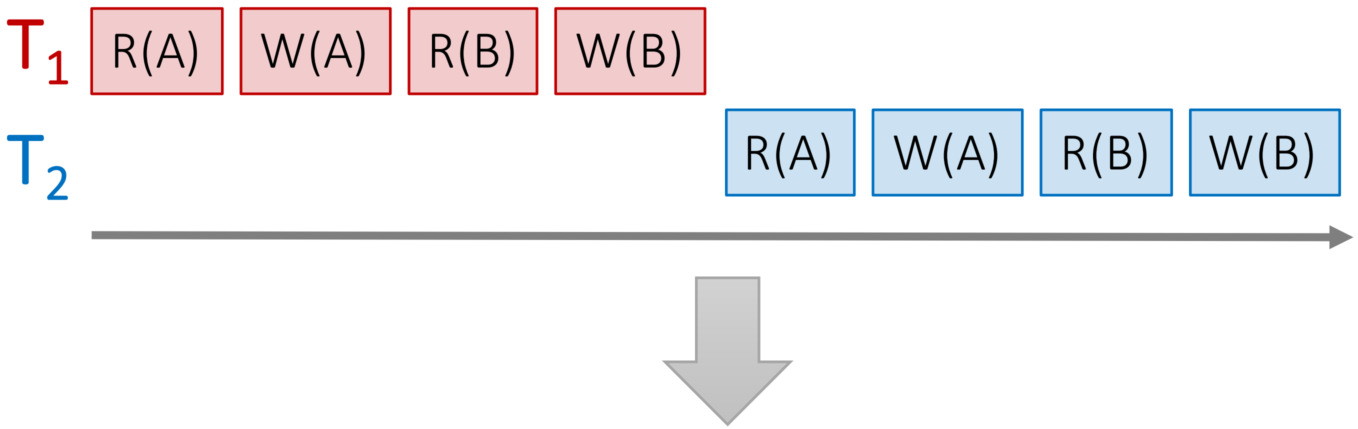
Recap from Lecture 9-2

What you will learn about in this section

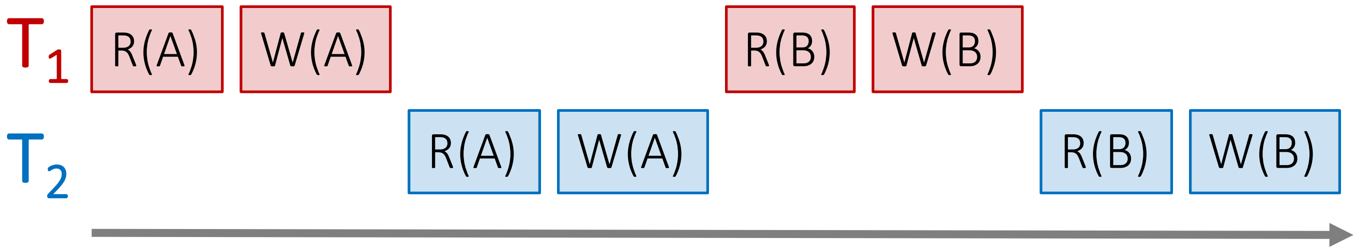
1. RECAP: Concurrency
2. Conflict Serializability
3. DAGs & Topological Orderings
4. Strict 2PL
5. Deadlocks

Recall: Concurrency as Interleaving TXNs

Serial Schedule:



Interleaved Schedule:



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

We call the particular order of interleaving a <u>schedule</u>

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

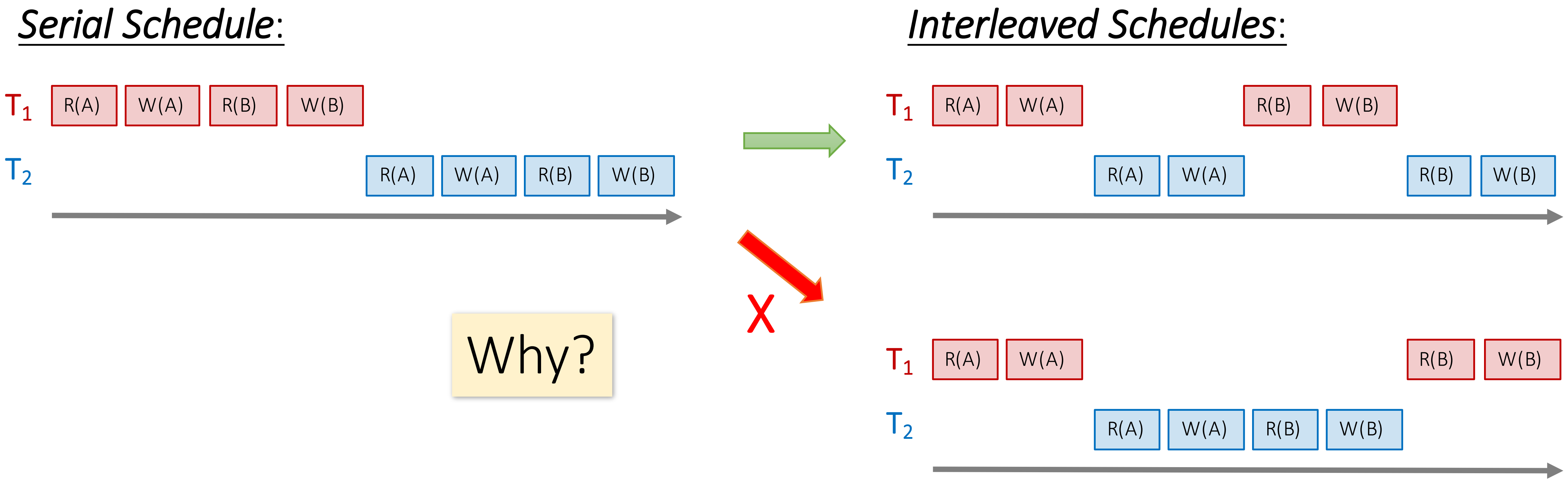
- The DBMS has freedom to interleave TXNs
- However, it must pick an interleaving or **schedule** such that isolation and consistency are maintained
  - Must be *as if* the TXNs had executed serially!

"With great power comes great responsibility"

A<u>CI</u>D

DBMS must pick a schedule which maintains isolation & consistency

Recall: "Good" vs. "bad" schedules



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

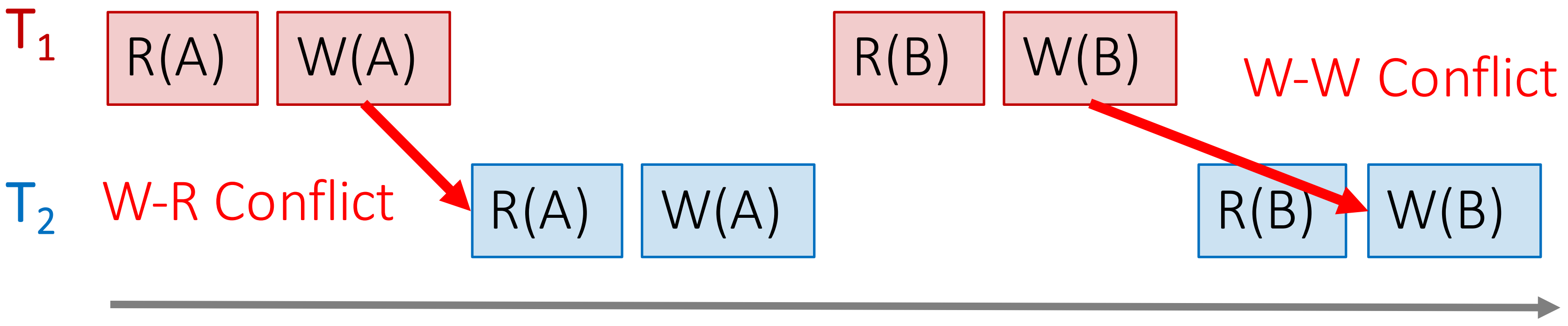
Ways of Defining "Good" vs. "Bad" Schedules

- 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..

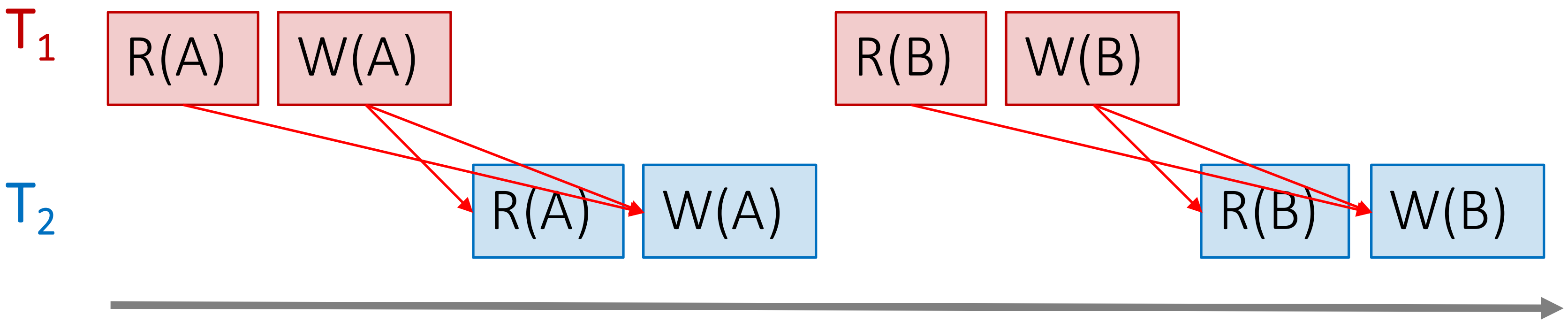
Conflicts

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

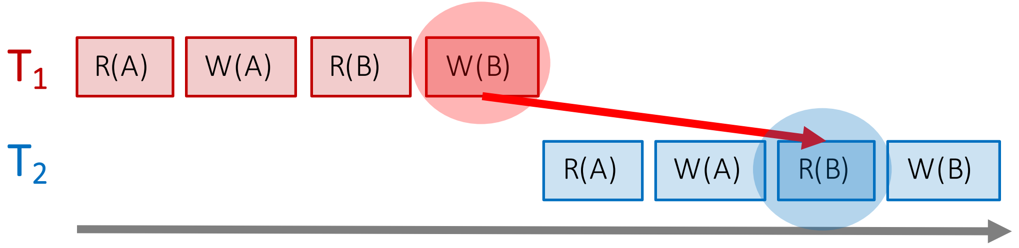
- Two schedules are **conflict equivalent** if:
  - They involve *the same actions of the same TXNs*
  - Every *pair of conflicting actions* of two TXNs are *ordered in the same way*
- Schedule S is **conflict serializable** if S is *conflict equivalent* to some serial schedule

Conflict serializable ⇒ serializable

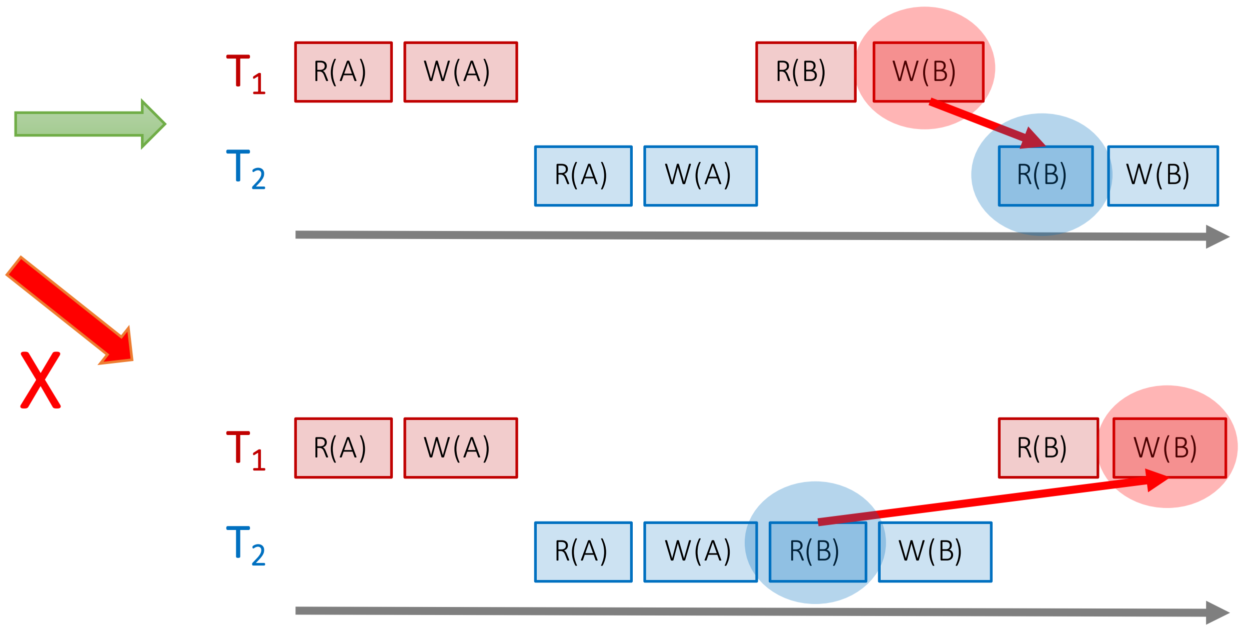
So if we have conflict serializable, we have consistency & isolation!

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

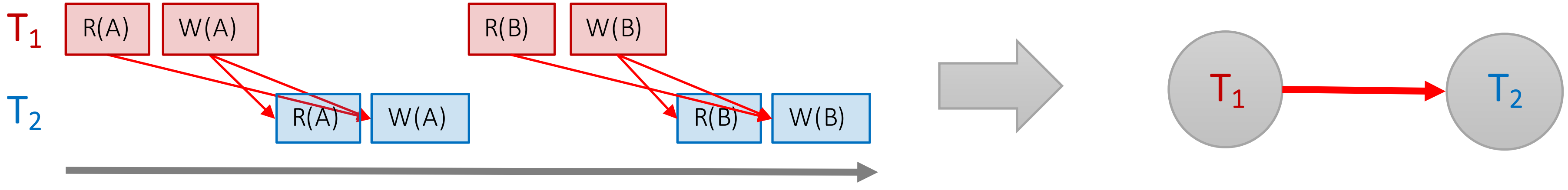
- **Conflicts** are things we talk about to help us characterize different schedules
  - Present in both “good” and “bad” schedules

- **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

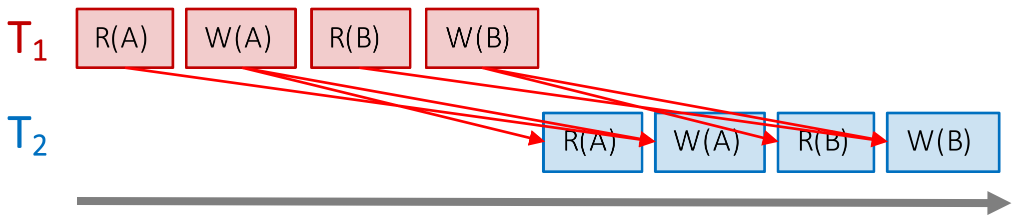
- Let's now consider looking at conflicts **at the TXN level**

- 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 <u>precede and conflict with</u> any actions in T_j**

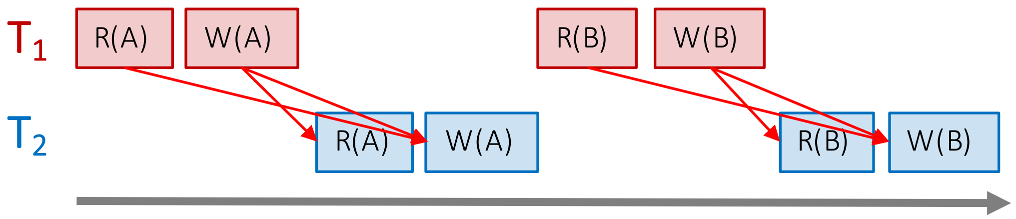


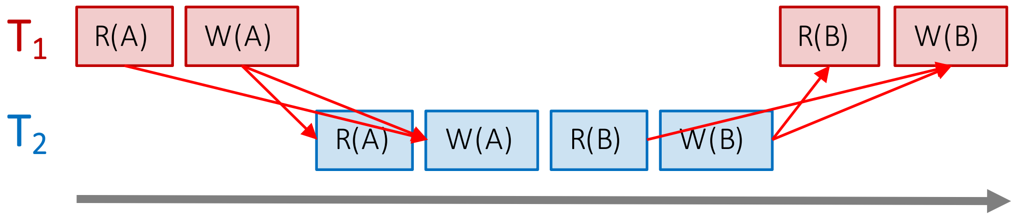
What can we say about "good" vs. "bad" conflict graphs?

Serial Schedule:



Interleaved Schedules:

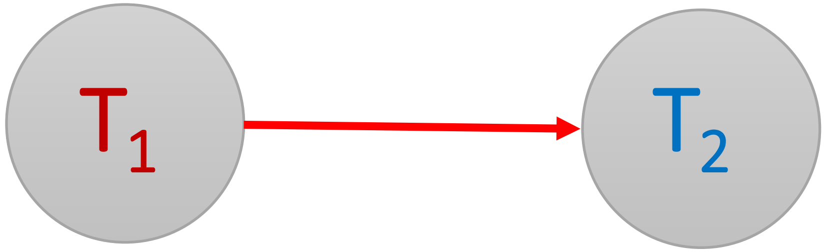




A bit complicated…

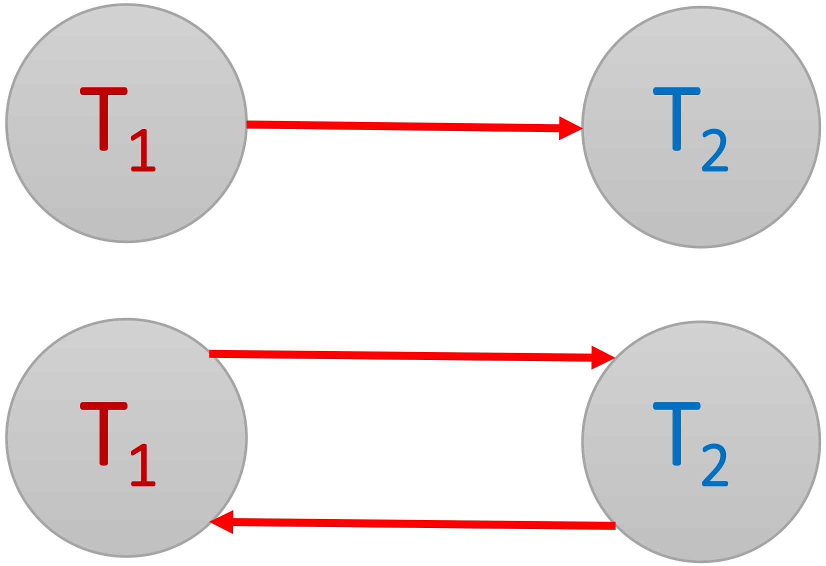
What can we say about "good" vs. "bad" conflict graphs?

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…

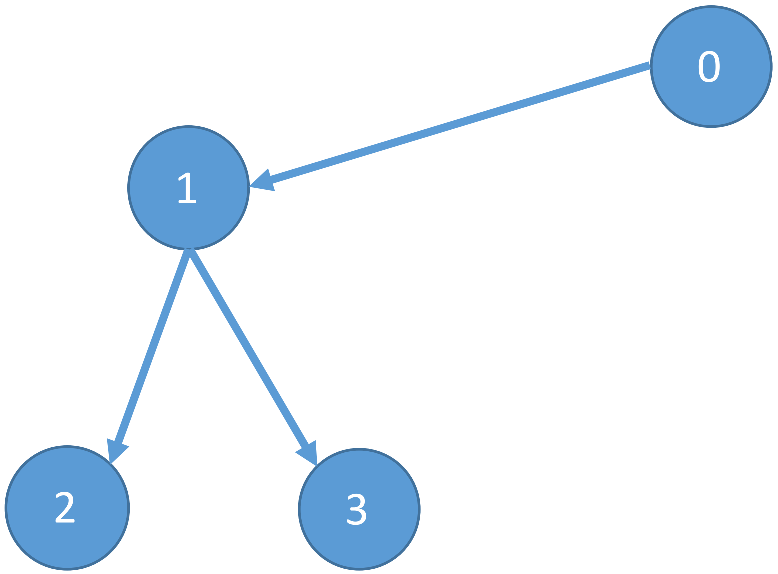
DAGs & Topological Orderings

- 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

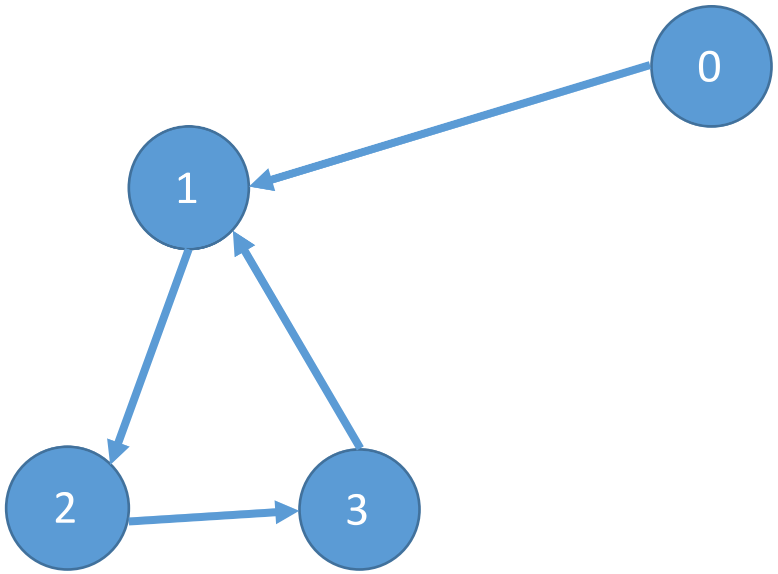
- Ex: What is one possible topological ordering here?



Ex: 0, 1, 2, 3 (or: 0, 1, 3, 2)

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 is **guarantees conflict serializability (if it completes- see upcoming…)**

- Also (*conceptually*) straightforward to implement, and transparent to the user!

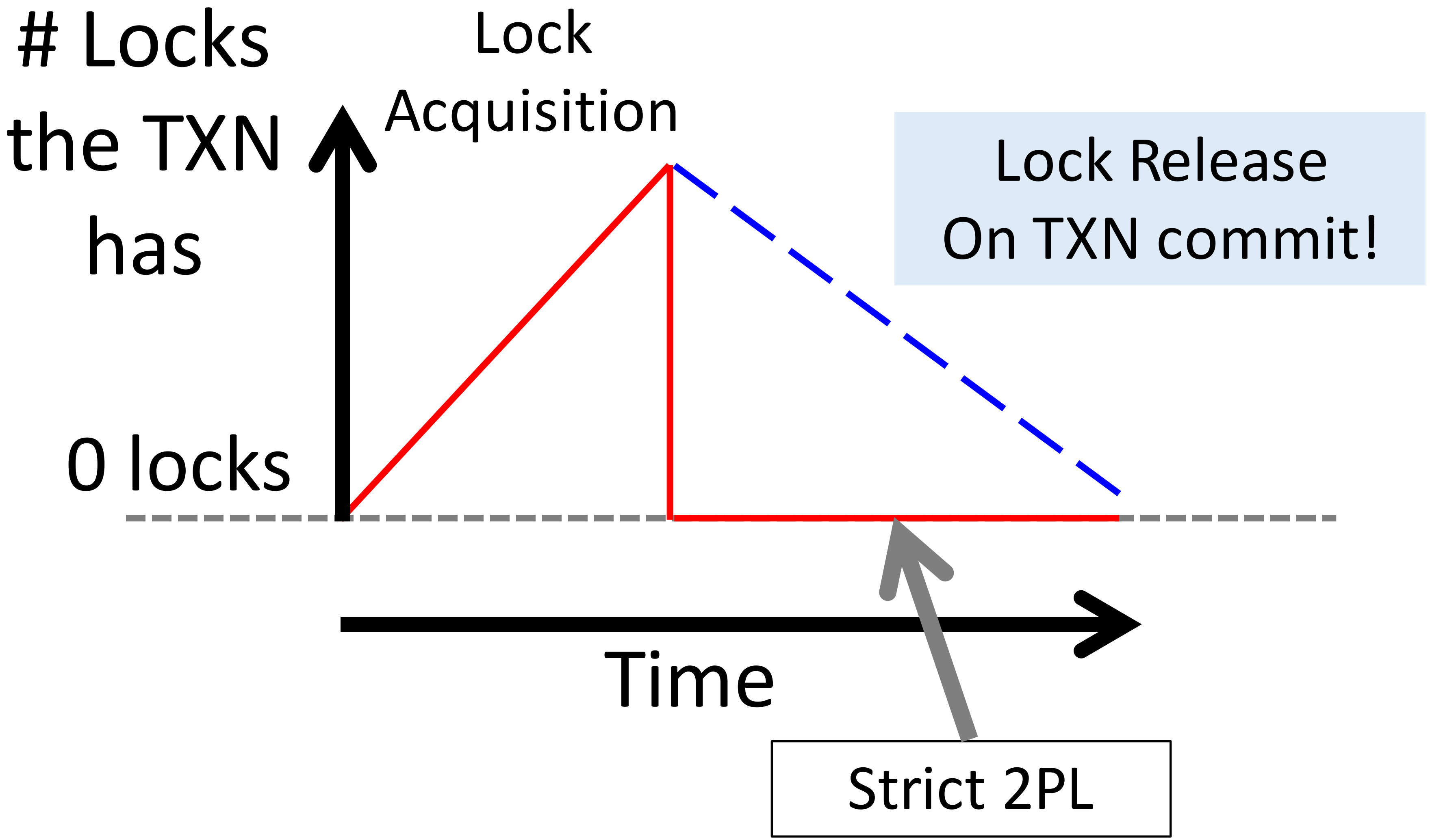
Strict Two-phase Locking (Strict 2PL) Protocol:

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

<u>Theorem:</u> 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 ⇒ 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

Waits-for graph:

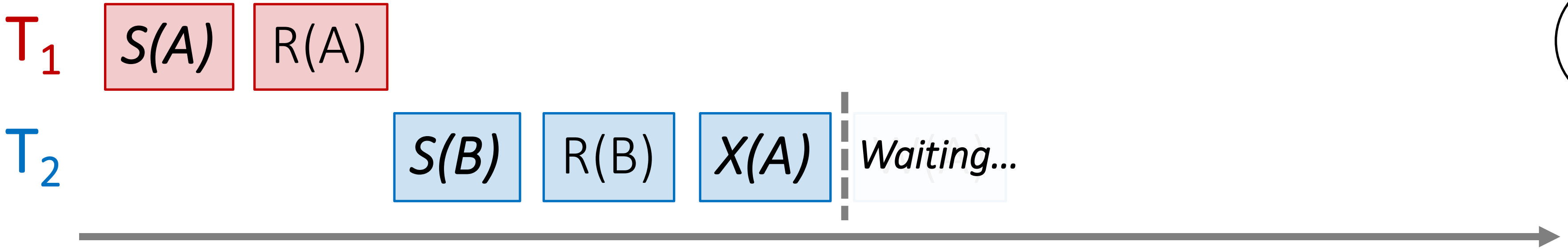
First, T_1 requests a shared lock on A to read from it

Deadlock Detection: Example

Waits-for graph:

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

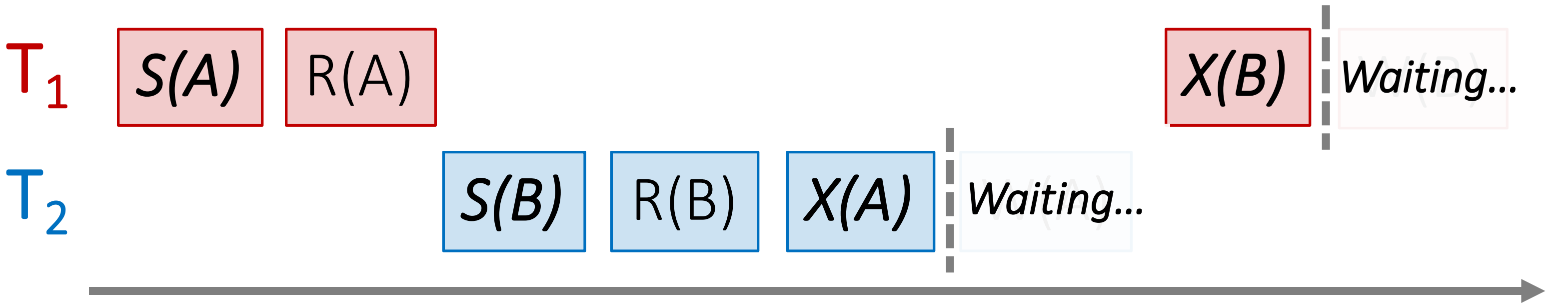
Deadlock Detection: Example



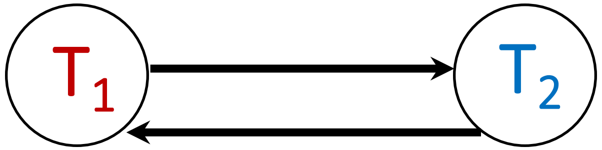
Waits-for graph:

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:

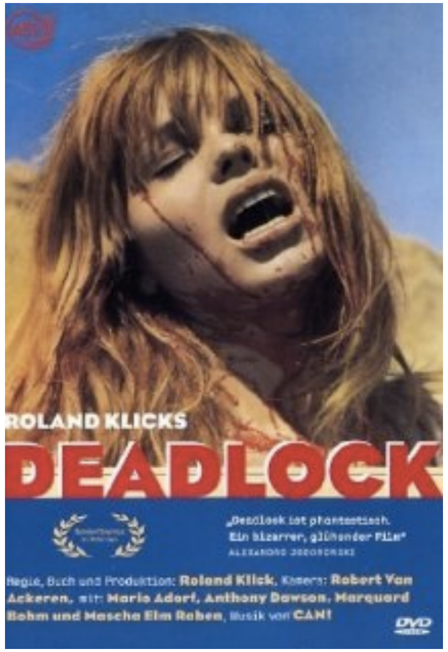


Cycle = DEADLOCK

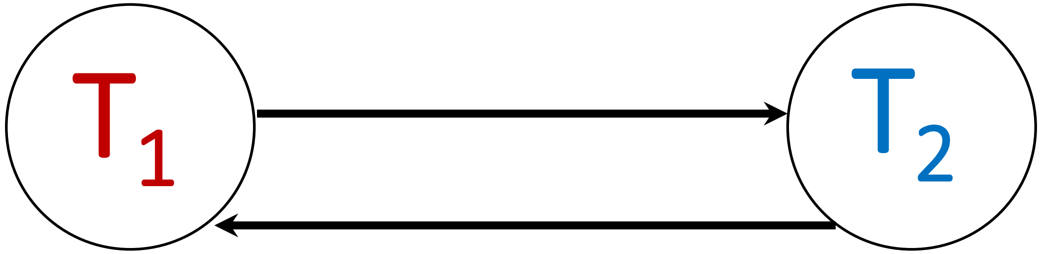
Finally, T_1 requests an exclusive lock on B to write to it- **now T_1 is waiting on T_2… DEADLOCK!**

```
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

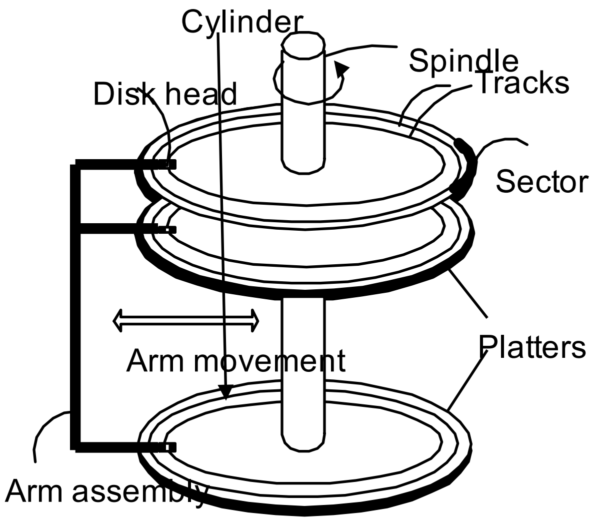
2. Many **mechanisms** have become **stand-alone systems**
   - **Indexing** to Key-value stores
   - Embedded join processing
   - SQL-like languages take some aspect of what we discuss (PIG, Hive)

What you will learn about in this section

1. RECAP: Storage and memory model

2. Buffer primer

High-level: Disk vs. Main Memory



Disk:

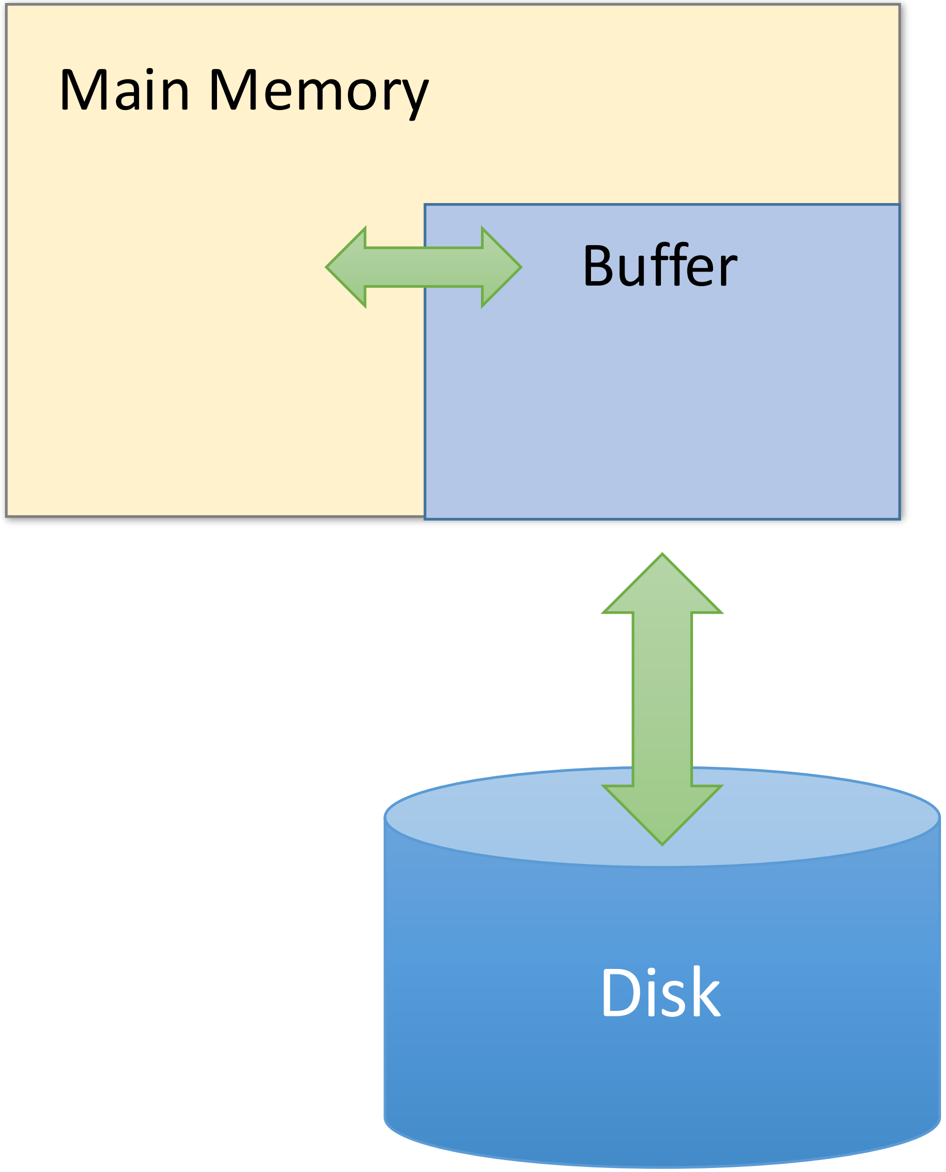
- ***Slow:*** Sequential *block* access
  - Read a blocks (not byte) at a time, so sequential access is cheaper than random
  - **Disk read / writes are expensive!**
- ***Durable:*** We will assume that once on disk, data is safe!
- ***Cheap***

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

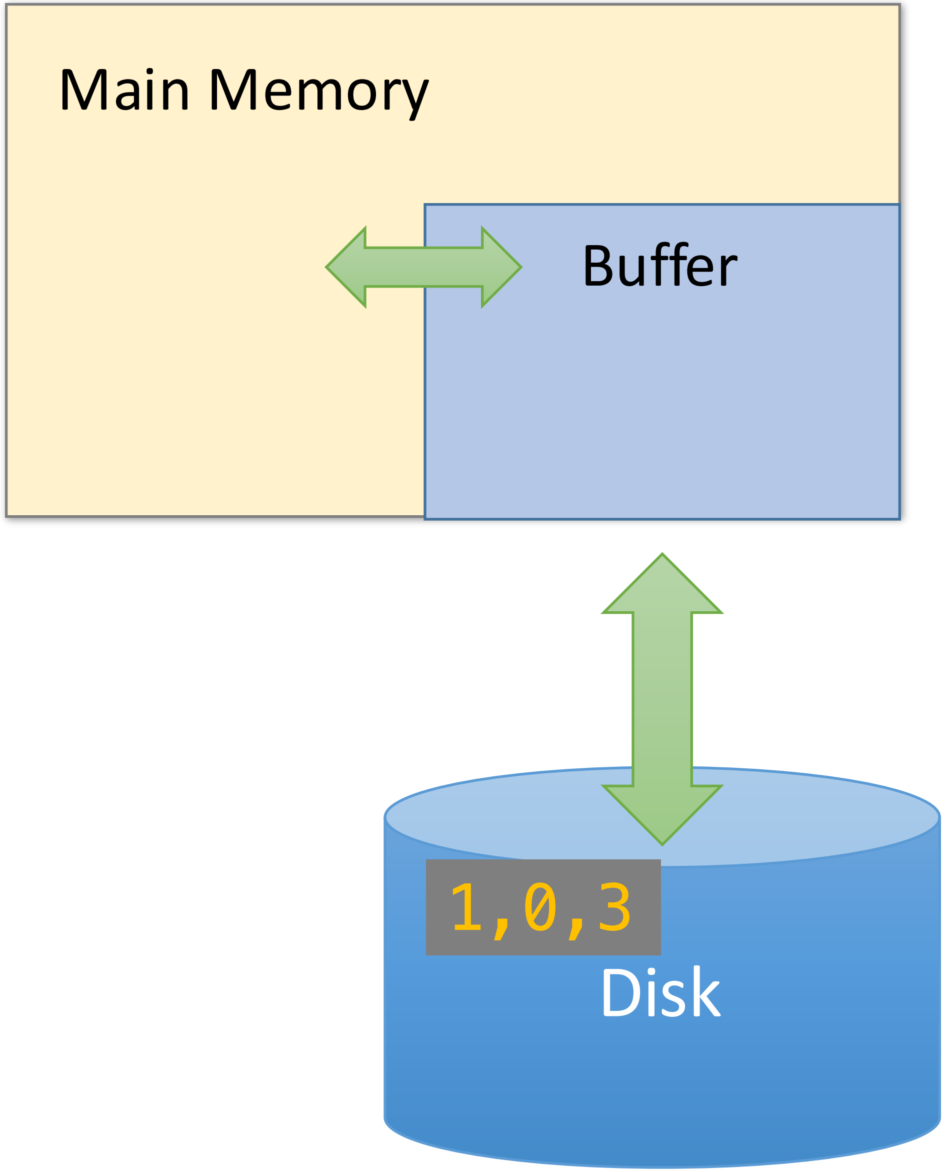
- A **buffer** is a region of physical memory used to store *temporary data*
  - *In this lecture:* a region in main memory used to store **intermediate data between disk and processes**
- *Key idea:* Reading / writing to disk is slow- need to cache data!



The (Simplified) Buffer

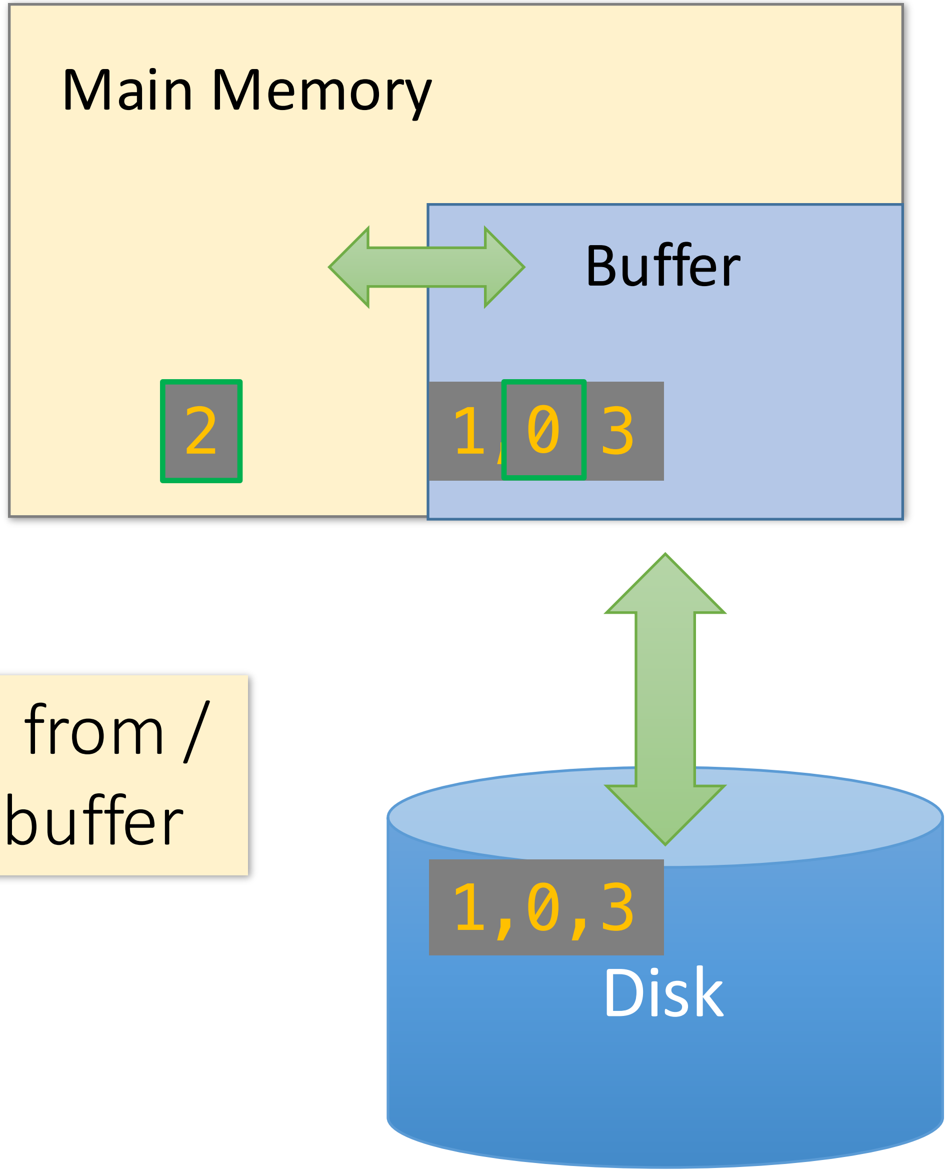
- In this class: We'll consider a buffer located in **main memory** that operates over **pages** and **files**:

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



The (Simplified) Buffer

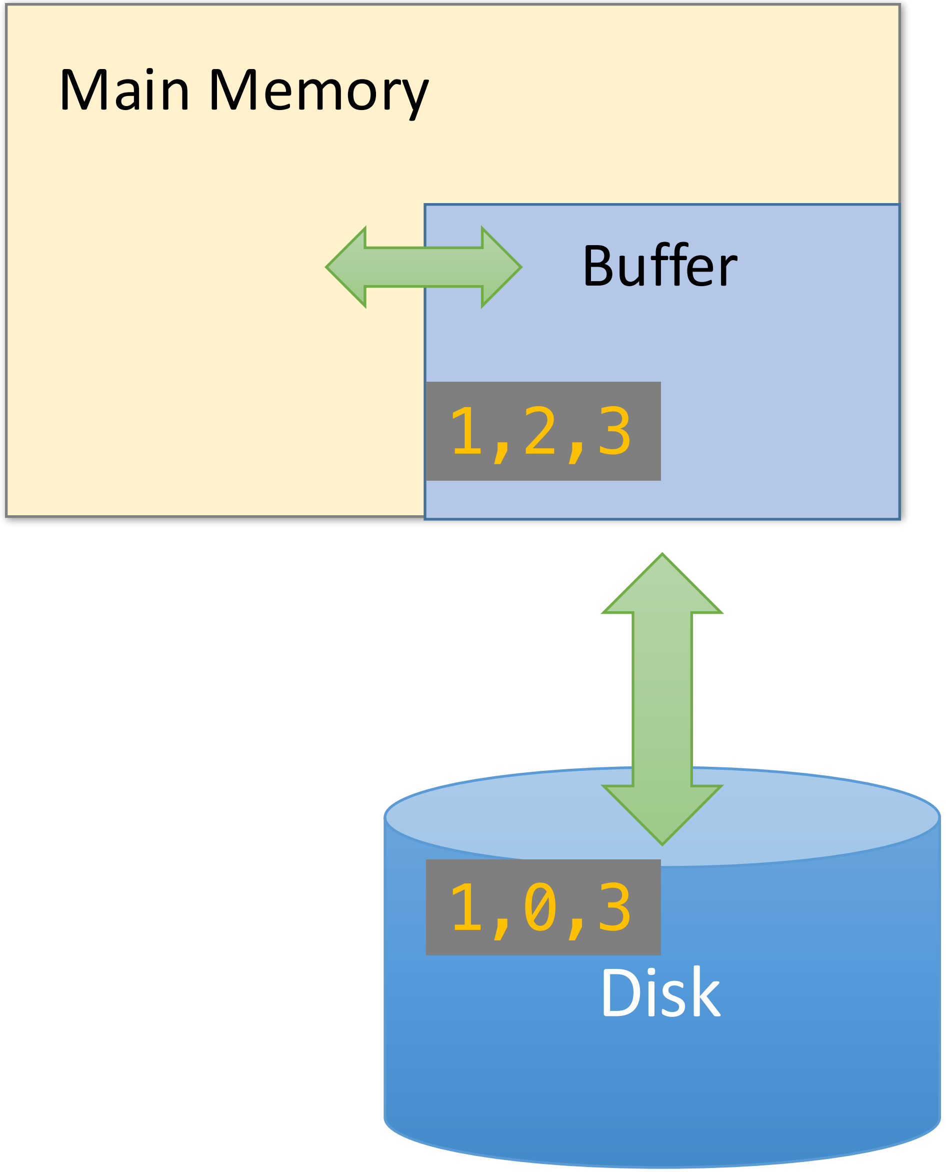
- In this class: We'll consider a buffer located in **main memory** that operates over **pages** and **files**:
  - **<u>Read(page):</u>** Read page from disk -> buffer *if not already in buffer*



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

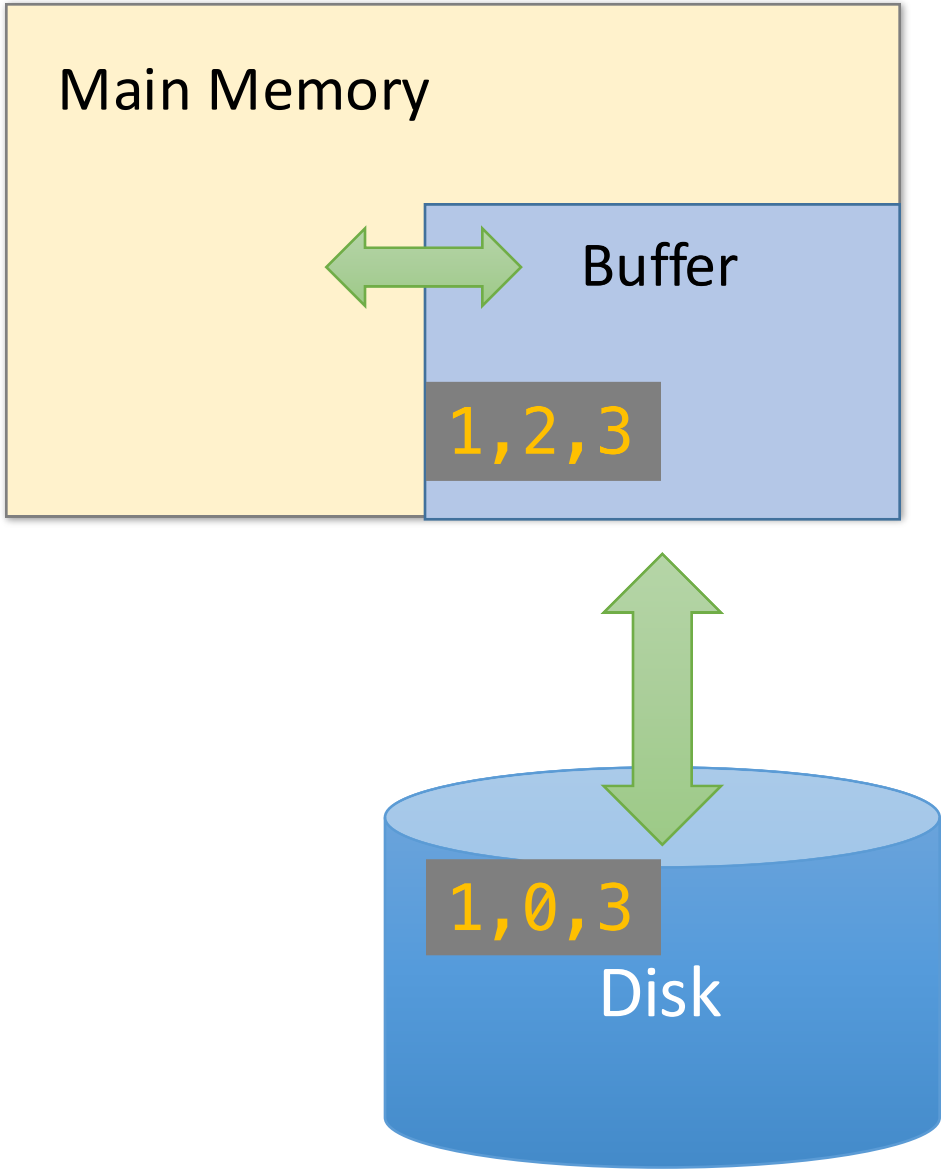
The (Simplified) Buffer

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



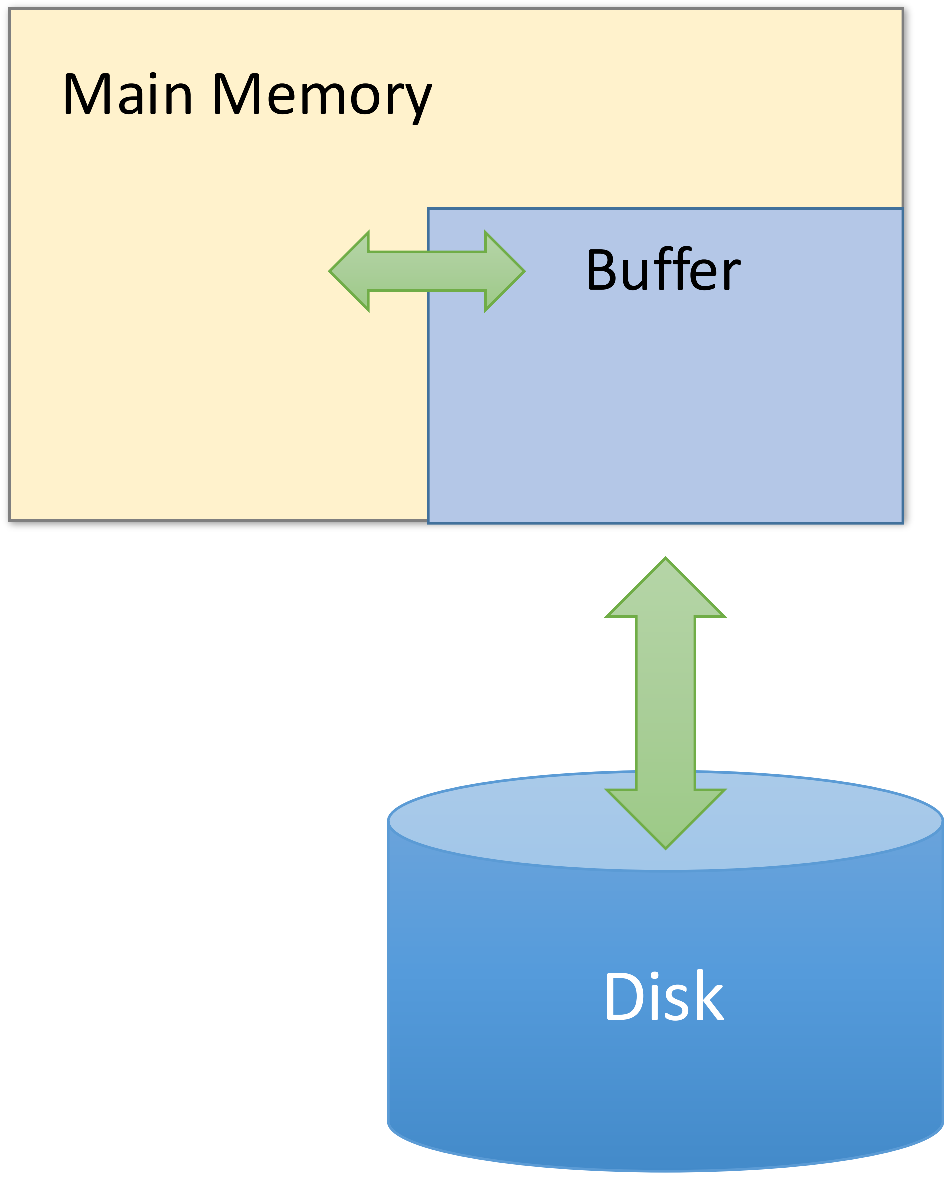
The (Simplified) Buffer

- 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

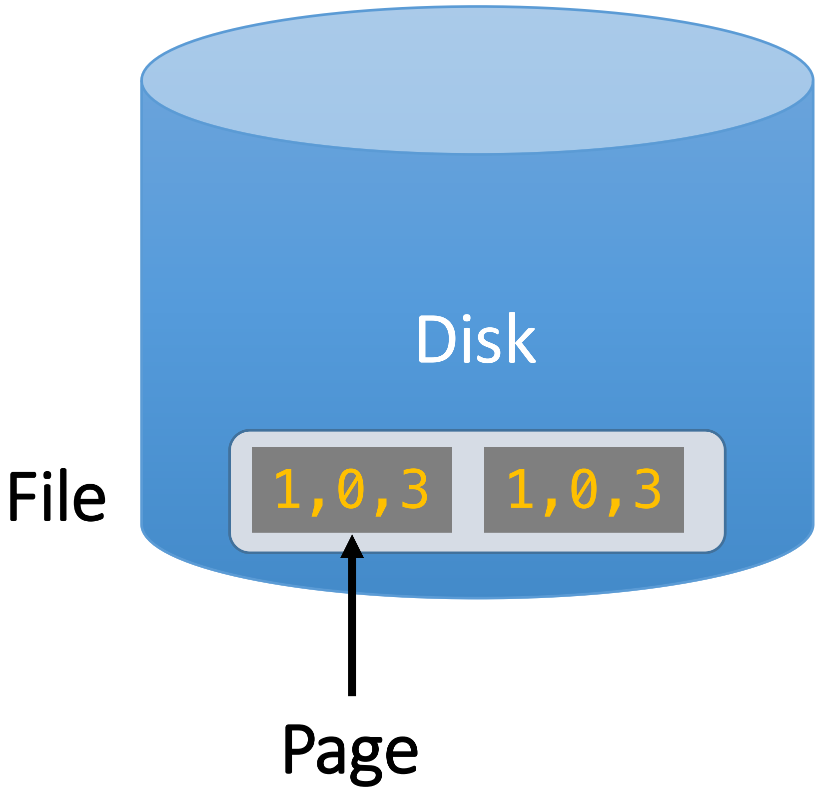
- A **buffer manager** handles supporting operations for the buffer:

  - 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:

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

Then:

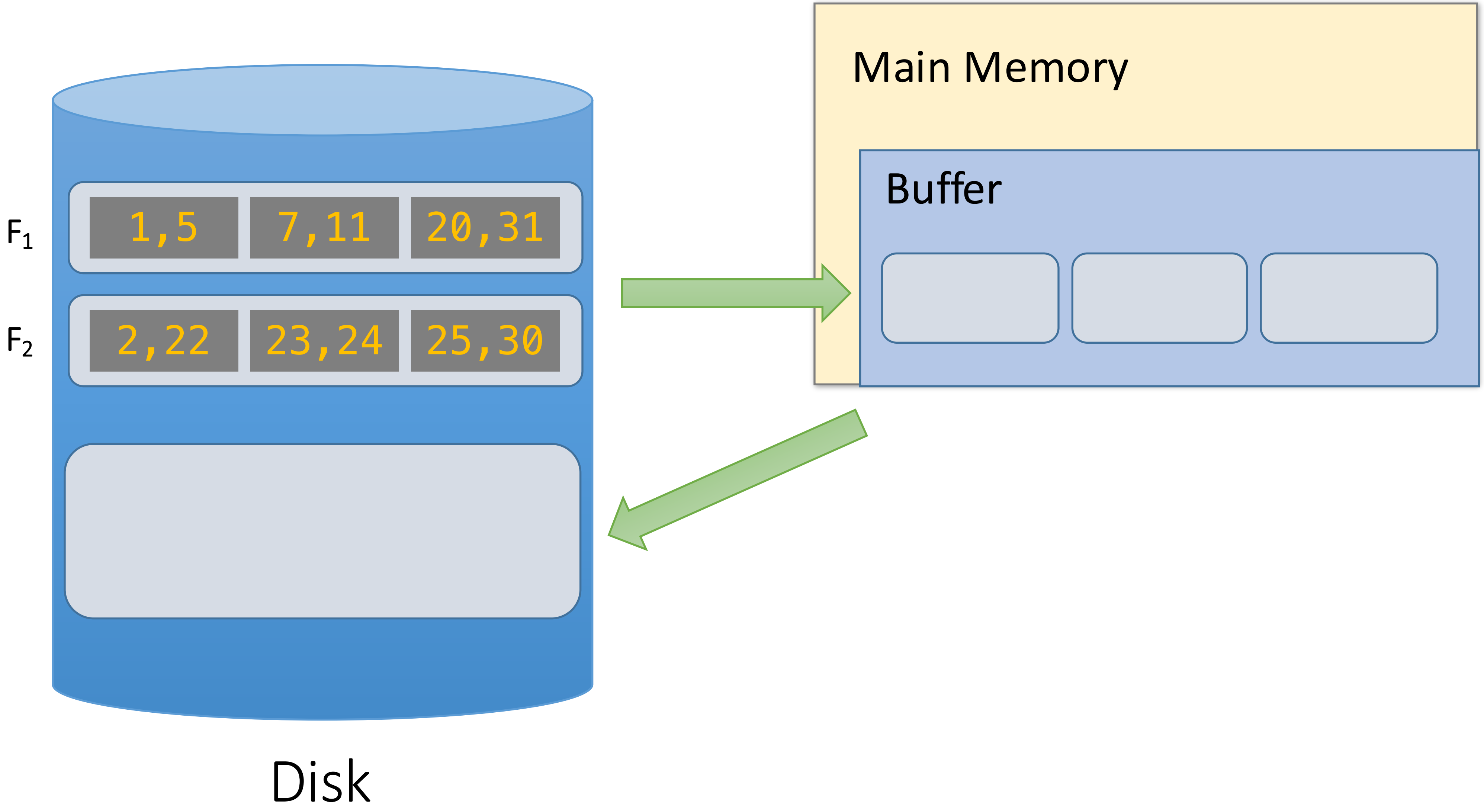
$$Min(A_1, B_1) \le A_i$$
$$Min(A_1, B_1) \le B_j$$

for i=1….N and j=1….M

External Merge Algorithm

Input:
Two sorted
files

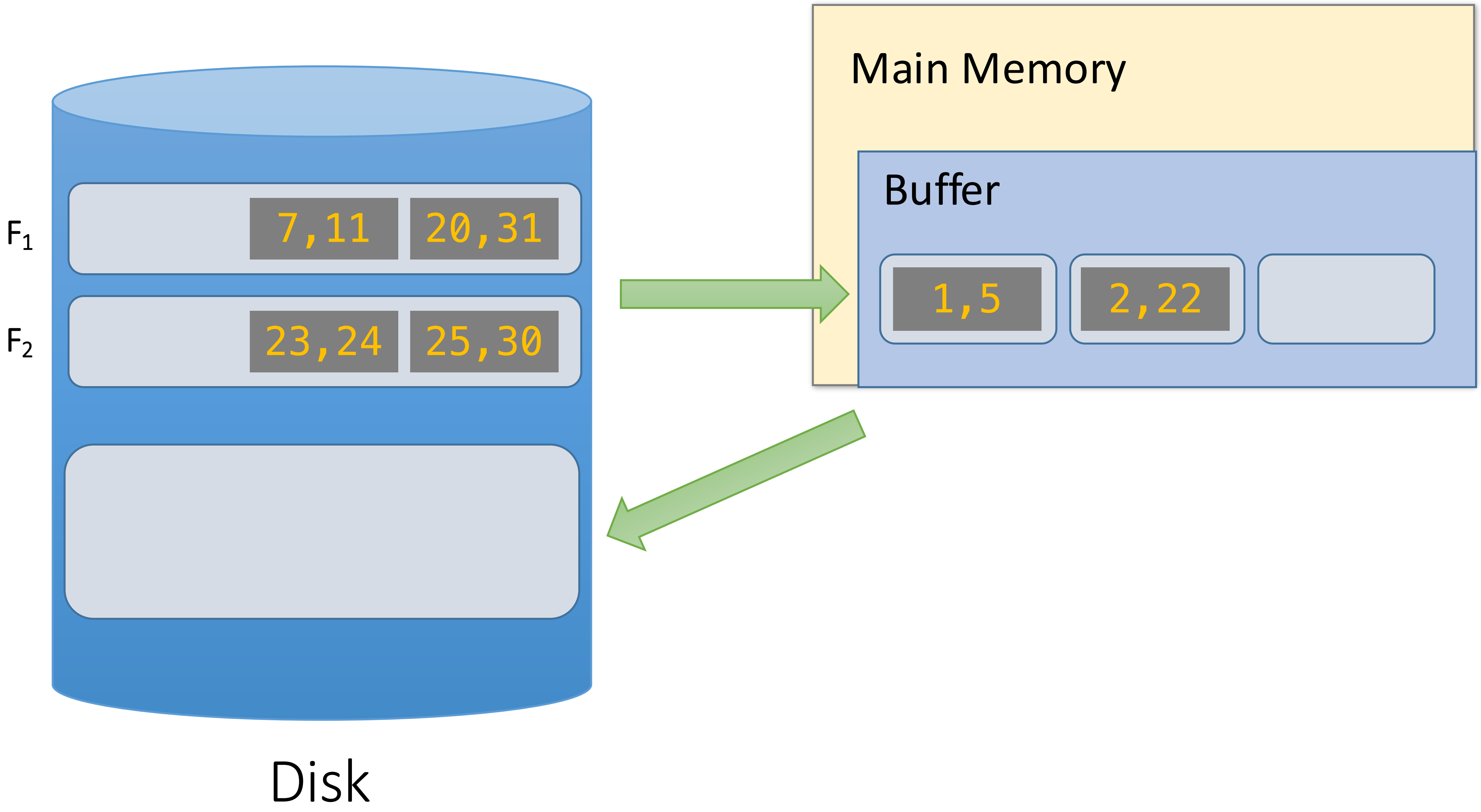
Output:
One *merged*
sorted file



External Merge Algorithm

Input:
Two sorted
files

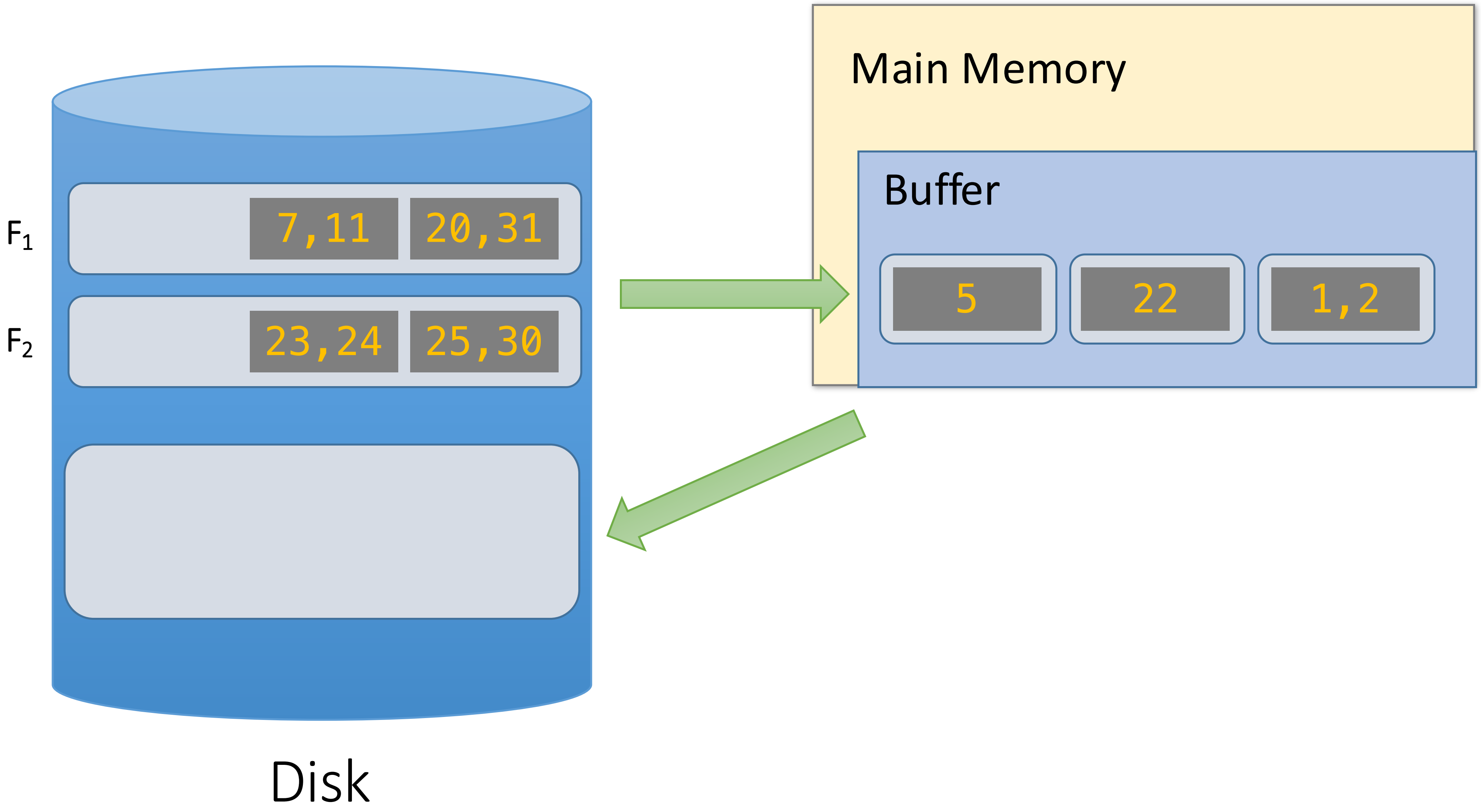
Output:
One *merged*
sorted file



External Merge Algorithm

Input:
Two sorted files

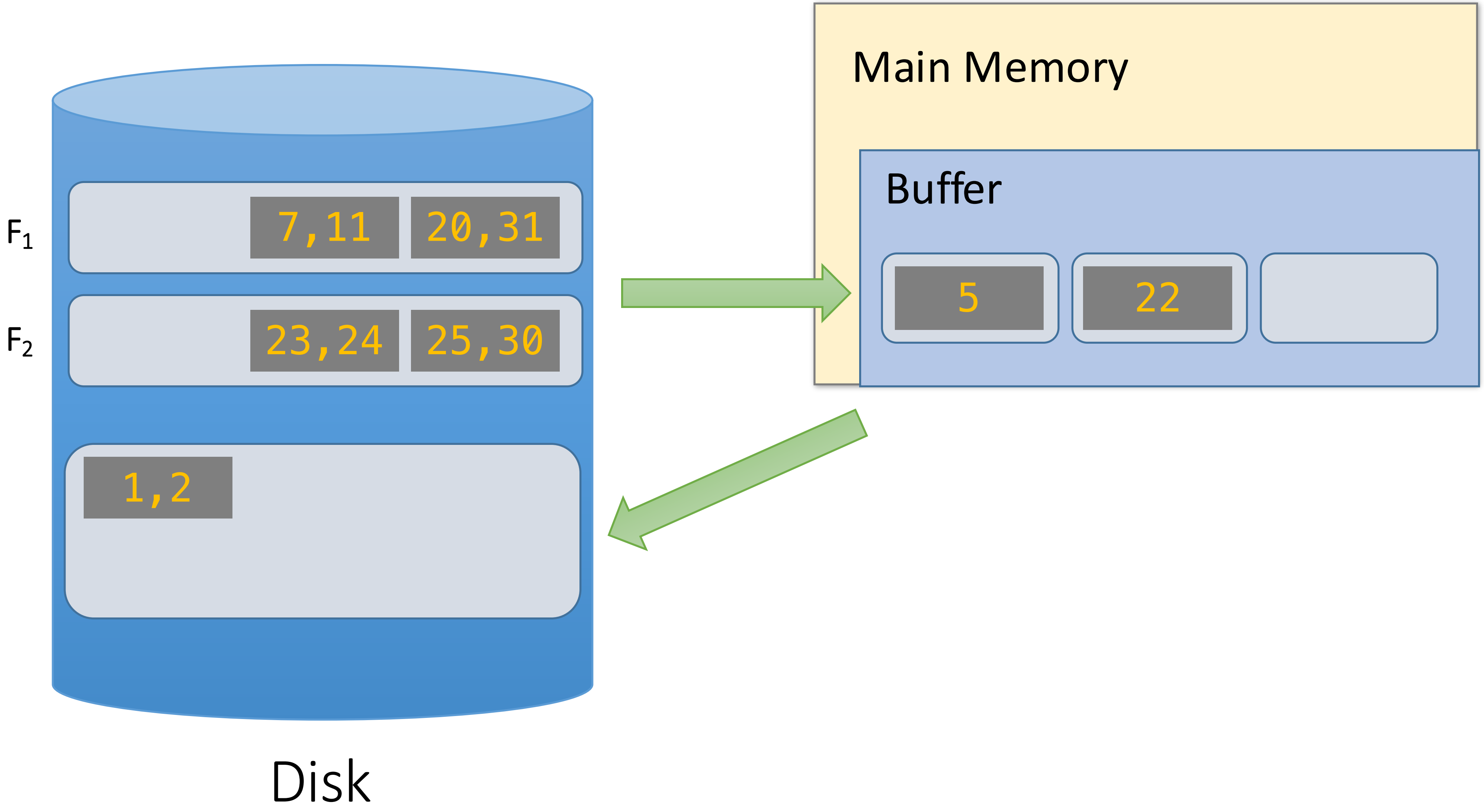
Output:
One *merged* sorted file



External Merge Algorithm

Input:
Two sorted
files

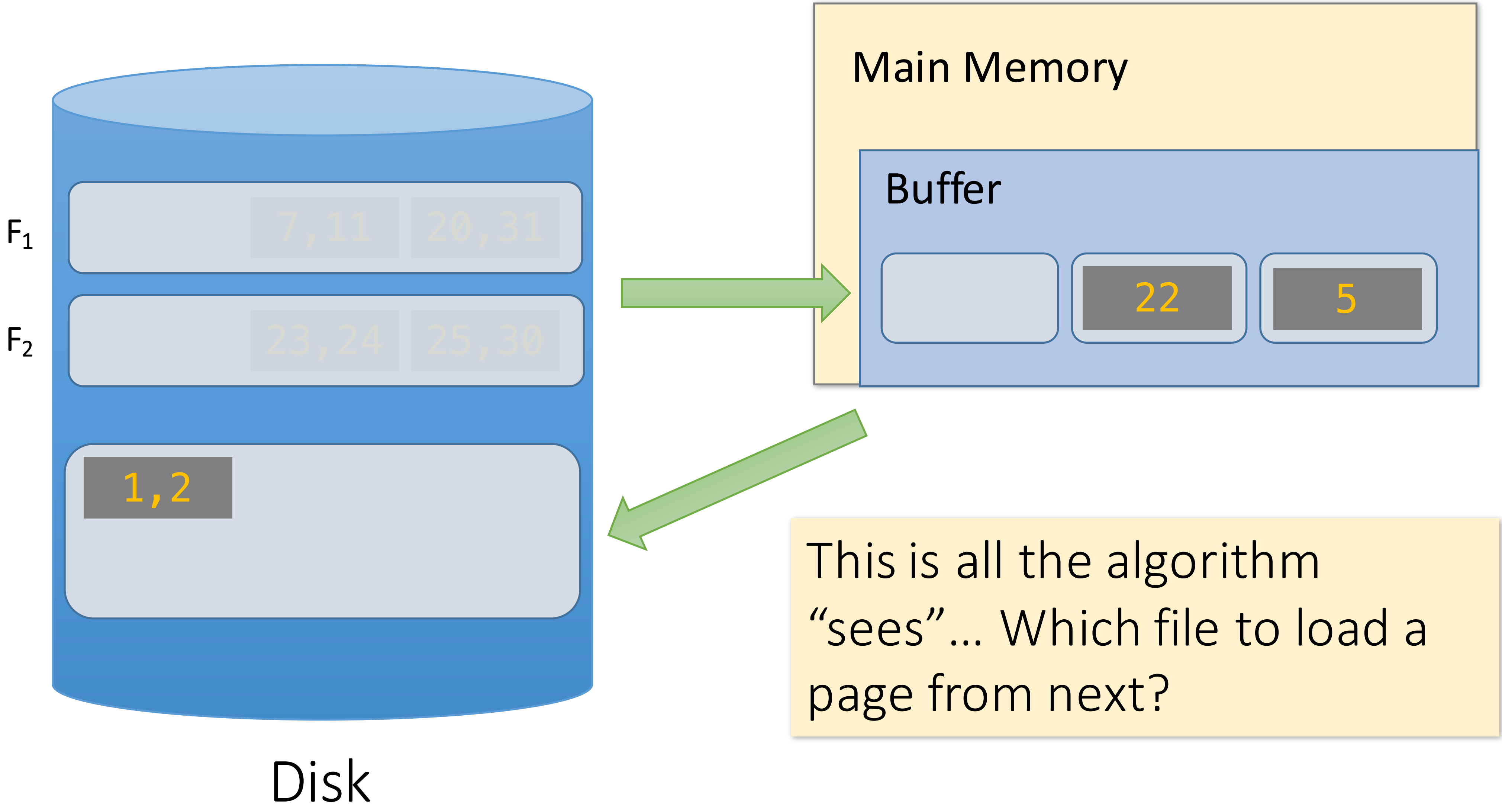
Output:
One *merged*
sorted file



External Merge Algorithm

Input:
Two sorted
files

Output:
One *merged*
sorted file

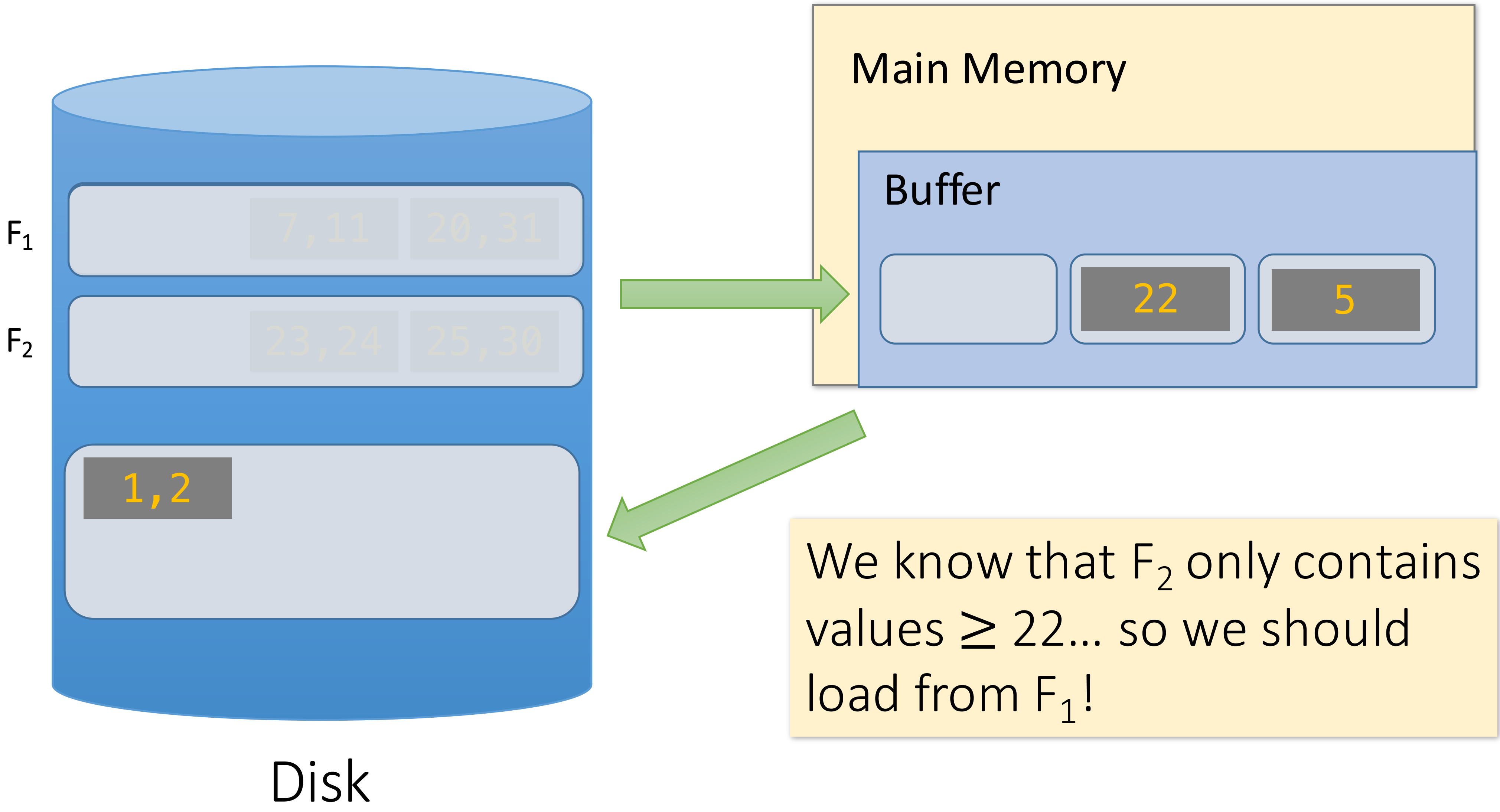


This is all the algorithm "sees"… Which file to load a page from next?

External Merge Algorithm

Input:
Two sorted files

Output:
One *merged* sorted file

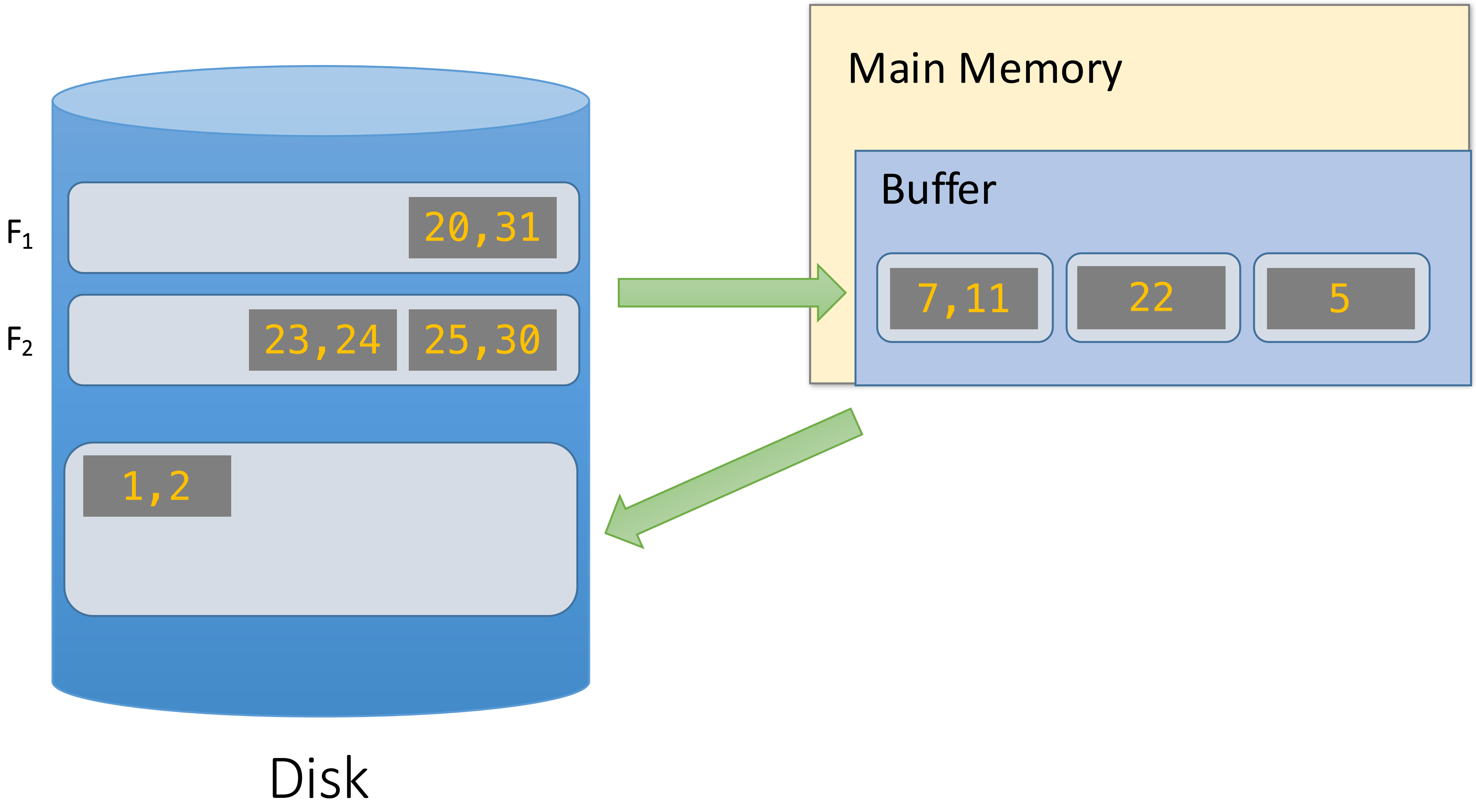


We know that F_2 only contains values ≥ 22… so we should load from F_1!

External Merge Algorithm

Input:
Two sorted
files

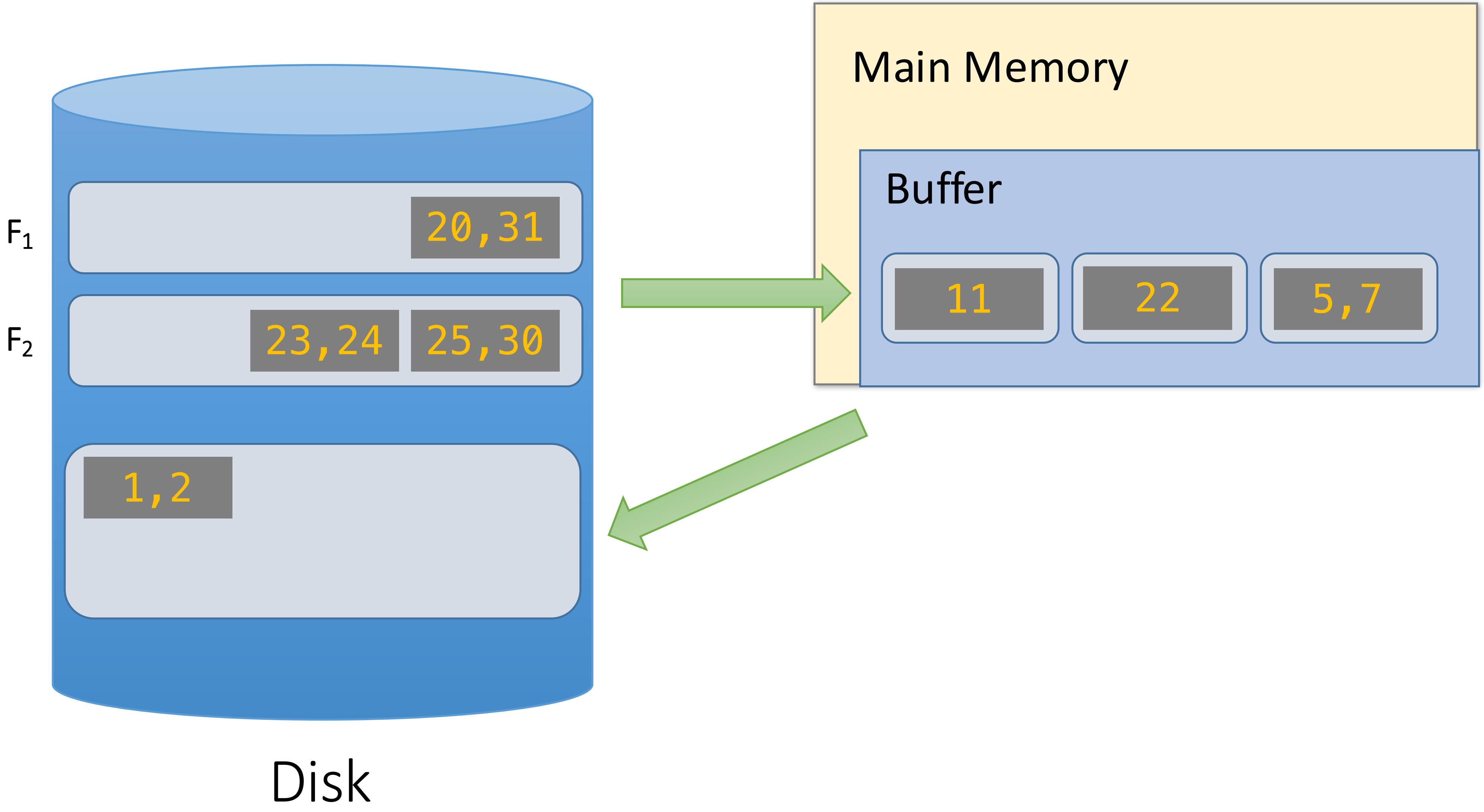
Output:
One *merged*
sorted file



External Merge Algorithm

Input:
Two sorted
files

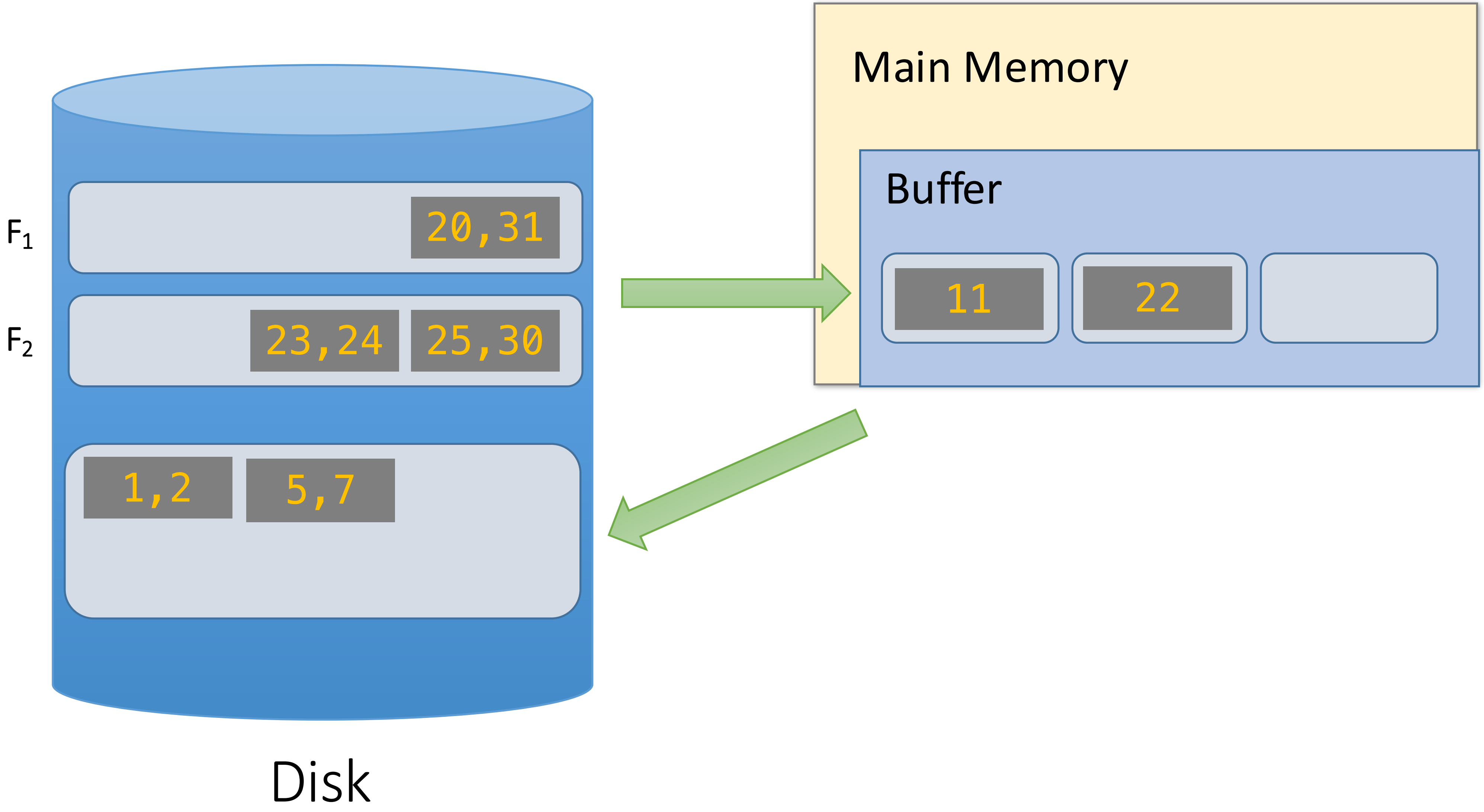
Output:
One *merged*
sorted file



External Merge Algorithm

Input:
Two sorted files

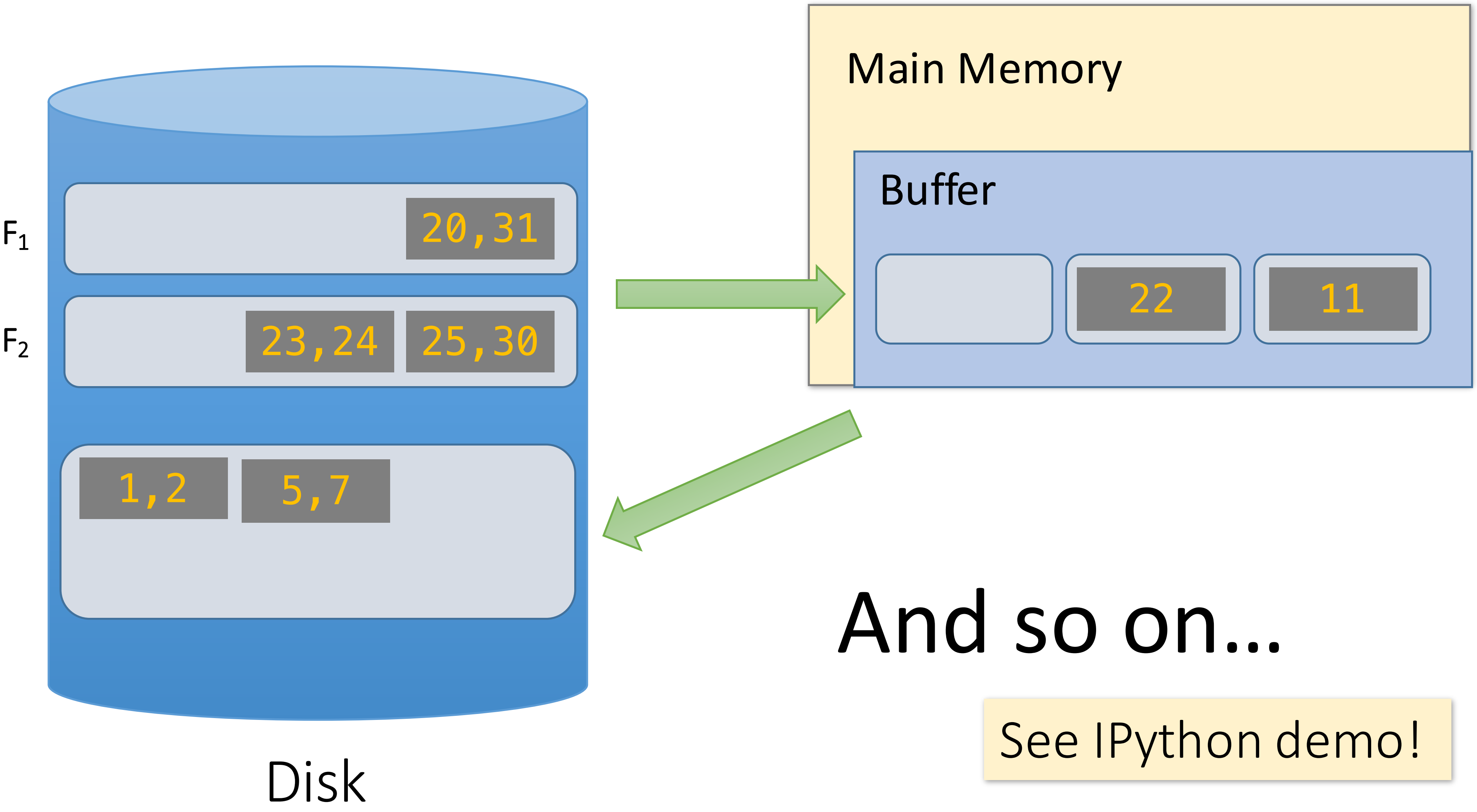
Output:
One *merged* sorted file



External Merge Algorithm

Input:
Two sorted files

Output:
One *merged* sorted file



And so on…

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?