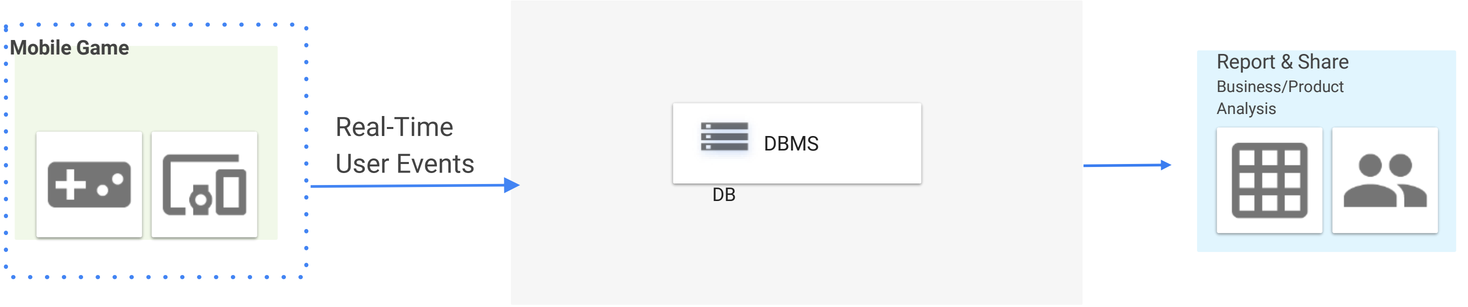
Lecture 12:
Indexing \ IO Model \ External Merge

How?

Example Game App

DB v0

(Recap lectures)



Q1: 1000 users/sec writing?
Q2: Offline?
Q3: Support v1, v1' versions?

Q7: How to model/evolve game data?
Q8: How to scale to millions of users?
Q9: When machines die, restore game state gracefully?

Q4: Which user cohorts?
Q5: Next features to build?
Experiments to run?
Q6: Predict ads demand?

App designer

Systems designer

Product/Biz designer

Today's Lecture

1. Indexing
2. IO Model
3. External Merge

1. Indexing

Example

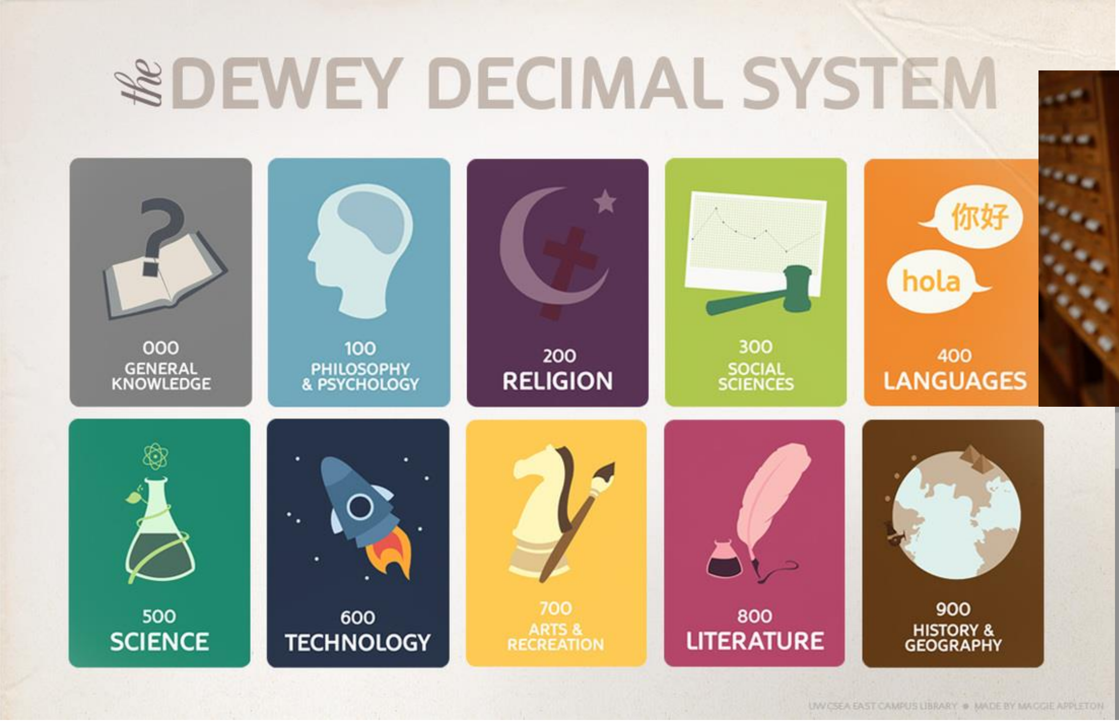
Find Book in Library

Design choices?

- Scan through each aisle
- Lookup pointer to book location, with librarian's organizing scheme

Example

Find Book in Library With Index



Index Cards

Algorithm for book titles

- Find right category
- Lookup Index, find location
- Walk to aisle. Scan book titles. Faster if books are sorted

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

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

Indexes on a table

- An index speeds up selections on search key (s)
  - Any subset of fields

- Example

Books(BID, name, author, price, year, text)

On which attributes would you build indexes?

Example

Billion_Books

BID	Title	Author	Published	Full_text
1001	*War and Peace*	Tolstoy	1869	…
1002	*Crime and Punishment*	Dostoyevsky	1866	…
1003	*Anna Karenina*	Tolstoy	1877	…

```
SELECT *
FROM   Billion_Books
WHERE Published > 1867
```

Example

By_Yr_Index

Published	BID
1866	1002
1869	***1001***
1877	***1003***
...	

Billion_Books

BID	Title	Author	Published	Full_text
1001	*War and Peace*	Tolstoy	1869	…
1002	*Crime and Punishment*	Dostoyevsky	1866	…
1003	*Anna Karenina*	Tolstoy	1877	…
...				

Maintain an index for this, and search over that!

Why might just keeping the table sorted by year not be good enough?

Example

By_Yr_Index

Published	BID
1866	1002
1869	1001
1877	1003

By_Author_Title_Index

Author	Title	BID
Dostoyevsky	Crime and Punishment	1002
Tolstoy	Anna Karenina	1003
Tolstoy	War and Peace	1001

Russian_Novels

BID	Title	Author	Published	Full_text
1001	*War and Peace*	Tolstoy	1869	…
1002	*Crime and Punishment*	Dostoyevsky	1866	…
1003	*Anna Karenina*	Tolstoy	1877	…

Can have multiple indexes to support multiple search keys

Indexes shown here as tables, but in reality we will use more efficient data structures…

Covering Indexes

By_Yr_Index

Published	BID
1866	1002
1869	1001
1877	1003

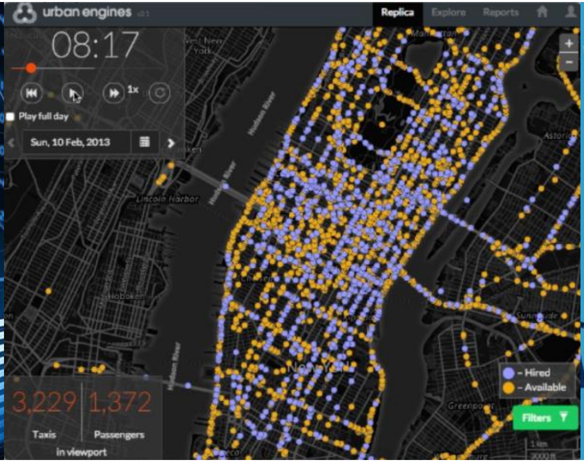
An index **covers** *for a specific query* if the index contains all the needed attributes- ***meaning the query can be answered using the index alone!***

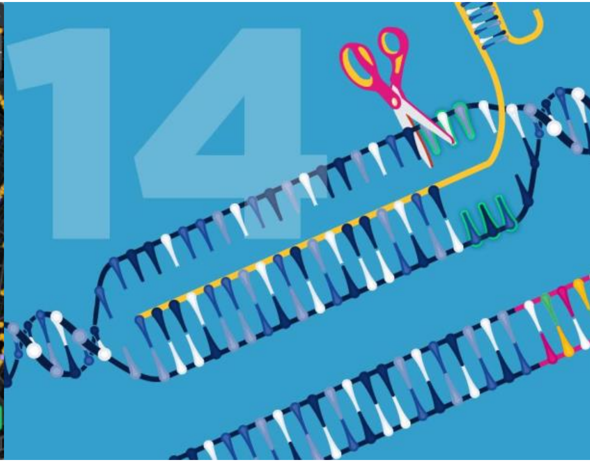
The "needed" attributes are the union of those in the SELECT and WHERE clauses…

Example:

```
SELECT Published, BID
FROM   Billion_Books
WHERE Published > 1867
```

Kinds of Indexes (different data types)





Index for Strings, Integers

Time series, GPS traces, Genomes, Video sequences

Advanced: Equality vs Similarity,
Ranges, Subsequences
Composites of above

Indexes (definition)

An *index* is a **data structure** mapping search keys to sets of rows in table

- Provides efficient lookup & retrieval by search key value (usually much faster than scanning all rows and searching)

An index can store

- full rows it points to (*primary index*), OR
- pointers to rows (*secondary index*) [much of our focus]

Operations on an Index

- Search: Quickly find all records which meet some *condition on the search key attributes*
  - (Advanced: across rows, across tables)
- Insert / Remove entries
  - Bulk Load / Delete. Why?

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

2. IO Model

Transition to Indexing Mechanisms

1. So you can **understand** what the database is doing!
   - Understand the CS challenges of a database and how to use it.
   - 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

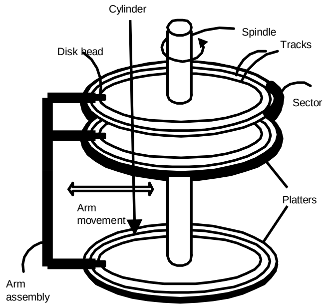
1. RECAP: Storage and memory model

1. Buffer primer

2. Pages

3. Files

High-level: Main Memory vs. Disk



<u>Random Access Memory (RAM) or Main Memory:</u>

- ***Fast:*** Random access, byte addressable
  - ~10x faster for <u>sequential access</u>
  - ~100,000x faster for <u>random access!</u>
- ***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!

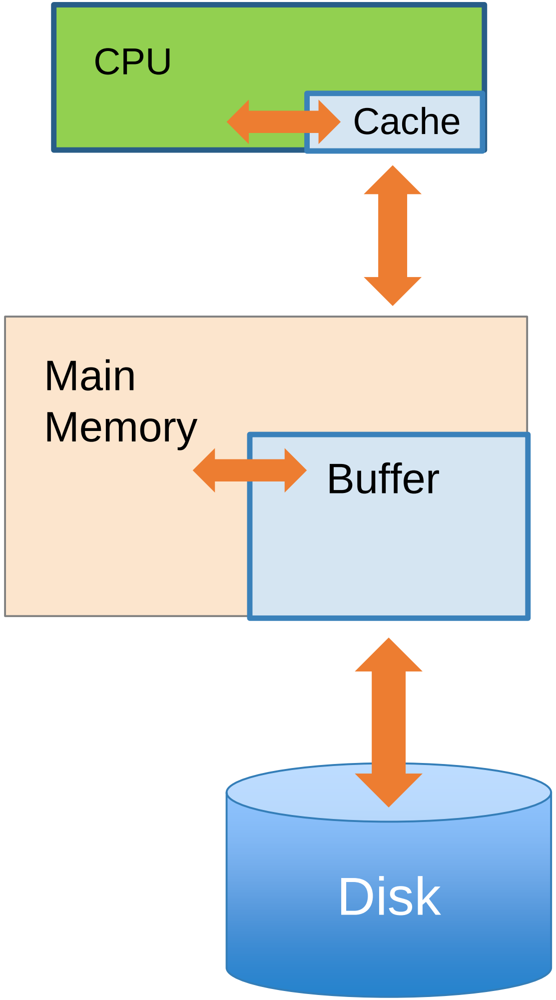
<u>Disk</u>:

- ***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***

The Buffer

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

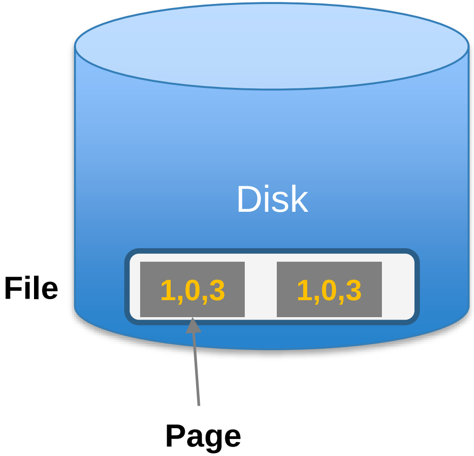
RECAP: Storage and memory model



A Simplified Filesystem Model

- For us, a **page** is ***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 key for good hardware utilization (in cs 346)

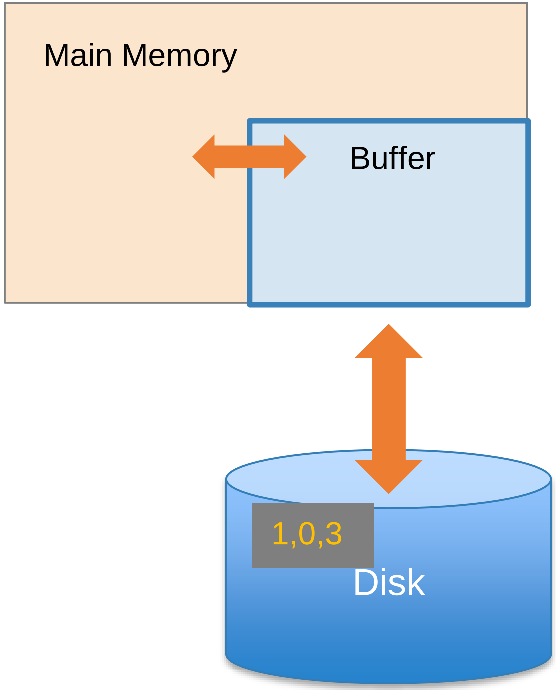


- And a **file** is a *variable-length list* of pages
  - Interface: create / open / close; next_page(); etc.

The (Simplified) Buffer

Buffer located in **main memory** operates over **pages** and **files**:

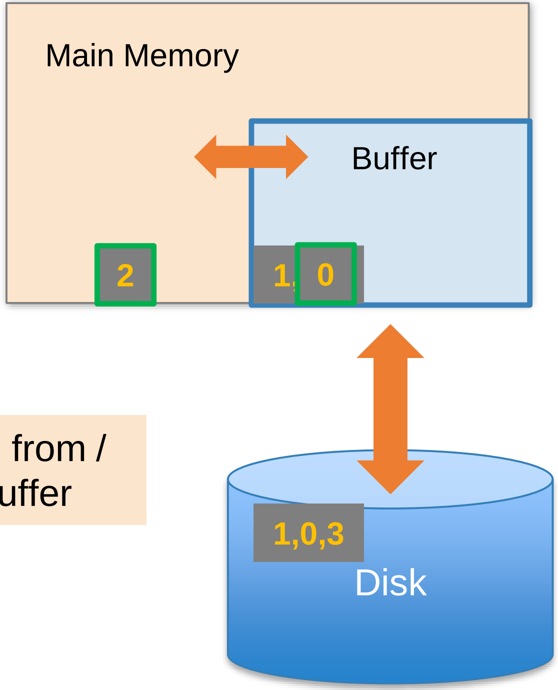
- **<u>Read(page):</u>** Read page from disk --> buffer *if not already in buffer*



The (Simplified) Buffer

Buffer located in **main memory** operates over **pages** and **files**:

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

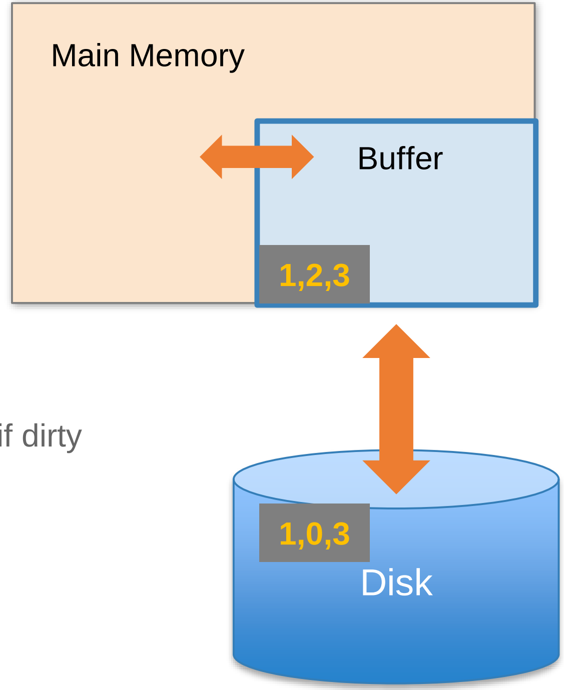


Processes can then read from / write to the page in the buffer

The (Simplified) Buffer

Buffer located in **main memory** operates over **pages** and **files**:

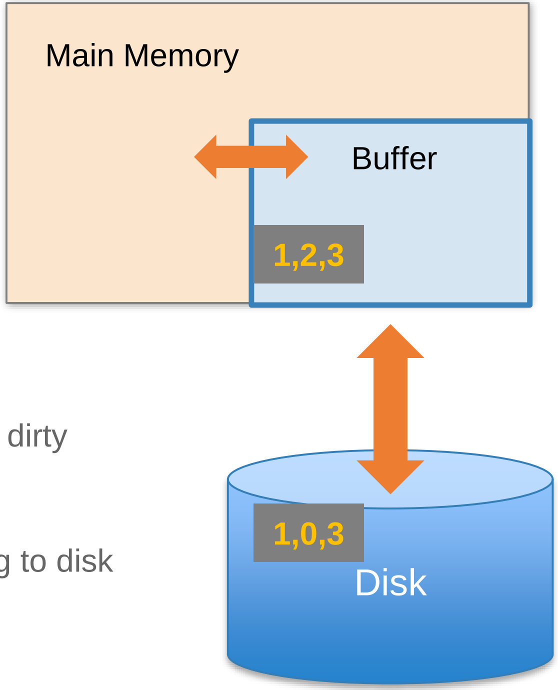
- **<u>Read(page):</u>** Read page from disk --> buffer *if not already in buffer*
- **<u>Flush(page):</u>** Evict page from buffer & write to disk, if dirty (dirty ⇒ modified page)



The (Simplified) Buffer

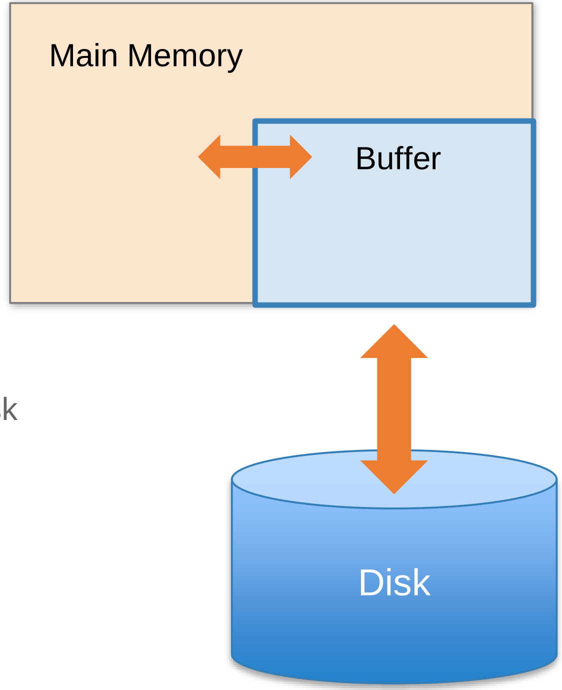
Buffer located in **main memory** operates over **pages** and **files**:

- **<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 if dirty
- **<u>Release(page):</u>** 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
  - Recovery and logging require ability to **flush** to disk



The Buffer Manager

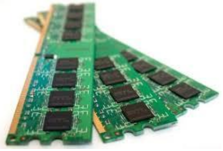
- A **buffer manager** manages operations for the buffer:
  - Primarily, handles & executes the “replacement policy”
    - i.e. finds a page in buffer to flush/release if buffer is full and a new page needs to be read in
  - DBMSs typically implement their own buffer management routines

3. External Merge Algorithm

Big Scaling (with Indexes)

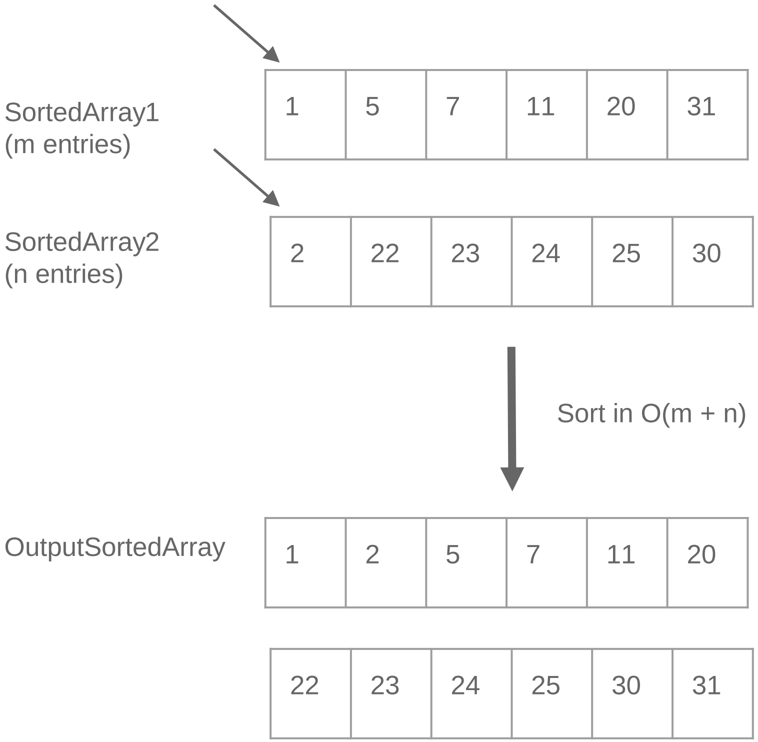
Roadmap

Primary data structures/algorithms

	Hashing	Sorting	Counting
	HashTables ($hash_i$(key) --> value)	BucketSort, QuickSort MergeSort	HashTable + Counter ($hash_i$(key) --> <count>)
		MergeSortedFiles, MergeSort	
			

MergeSortedFiles

(in RAM)



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

- **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:

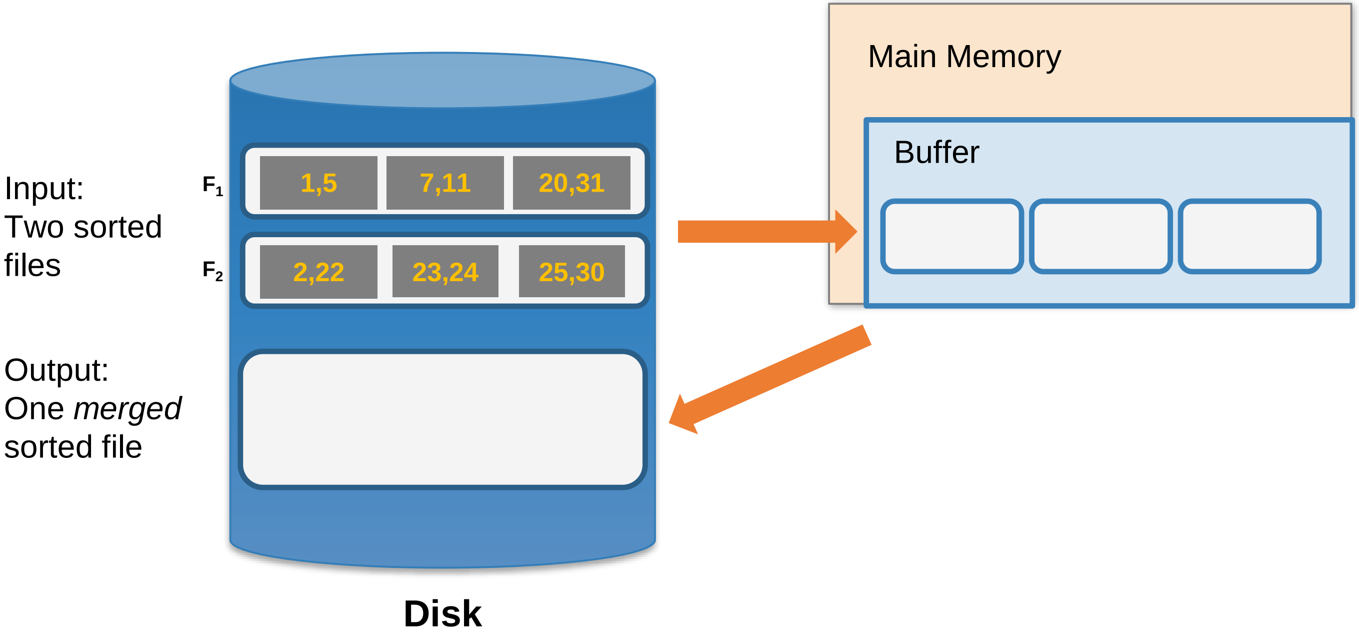
$$A_1 \le A_2 \le \cdots \le A_N$$
$$B_1 \le B_2 \le \cdots \le B_M$$

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

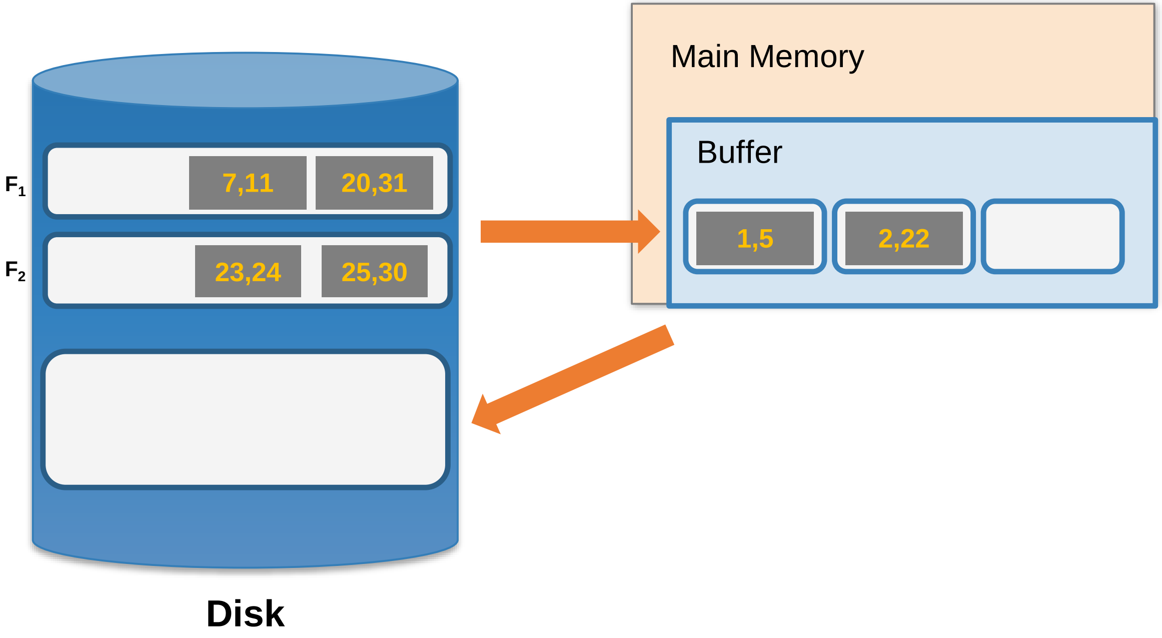
External Merge Algorithm



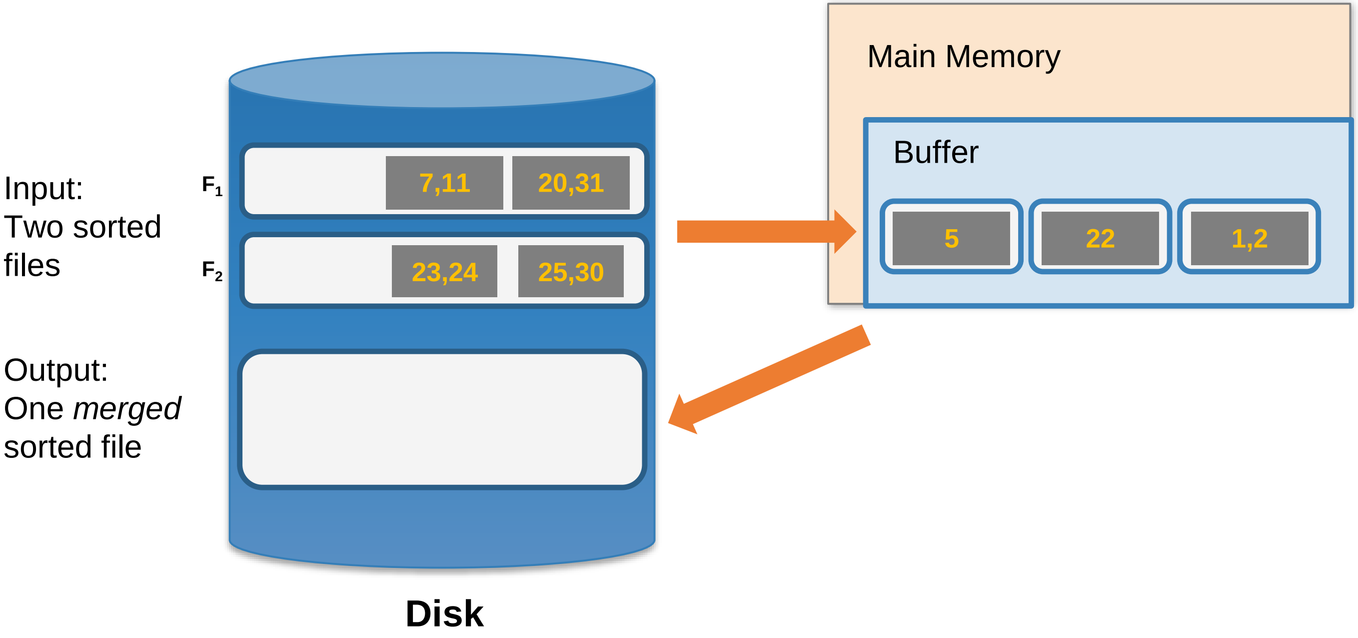
External Merge Algorithm

Input:
Two sorted files

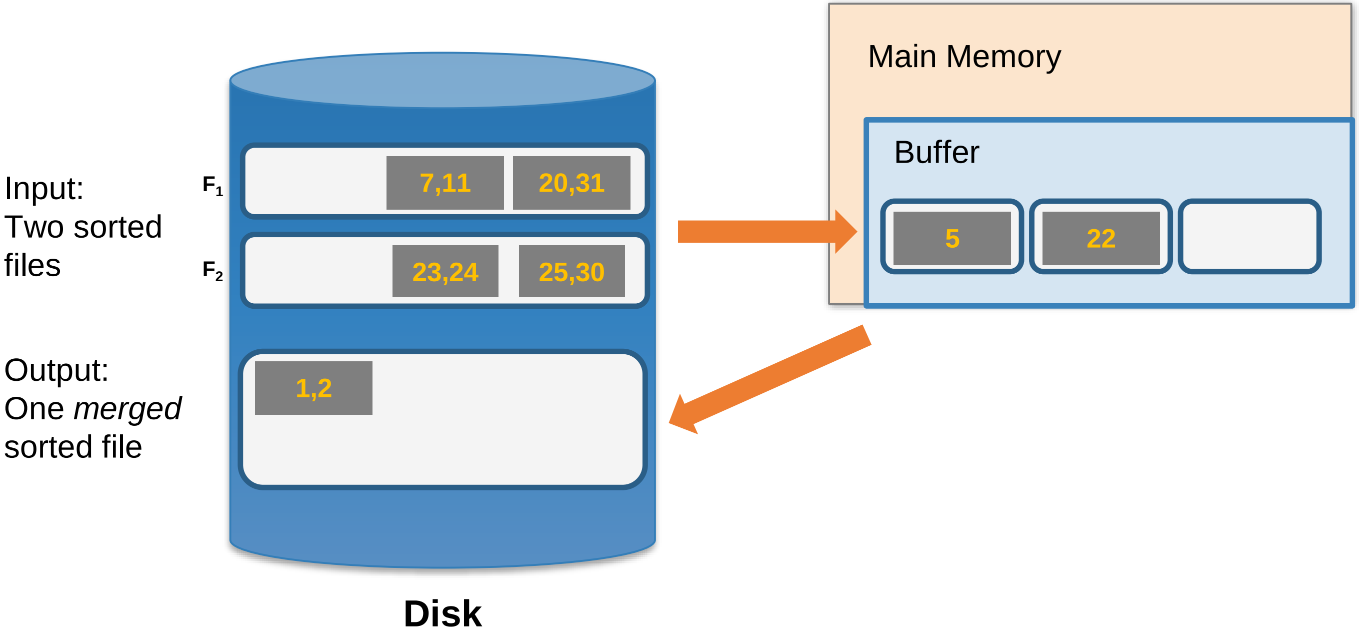
Output:
One *merged* sorted file



External Merge Algorithm



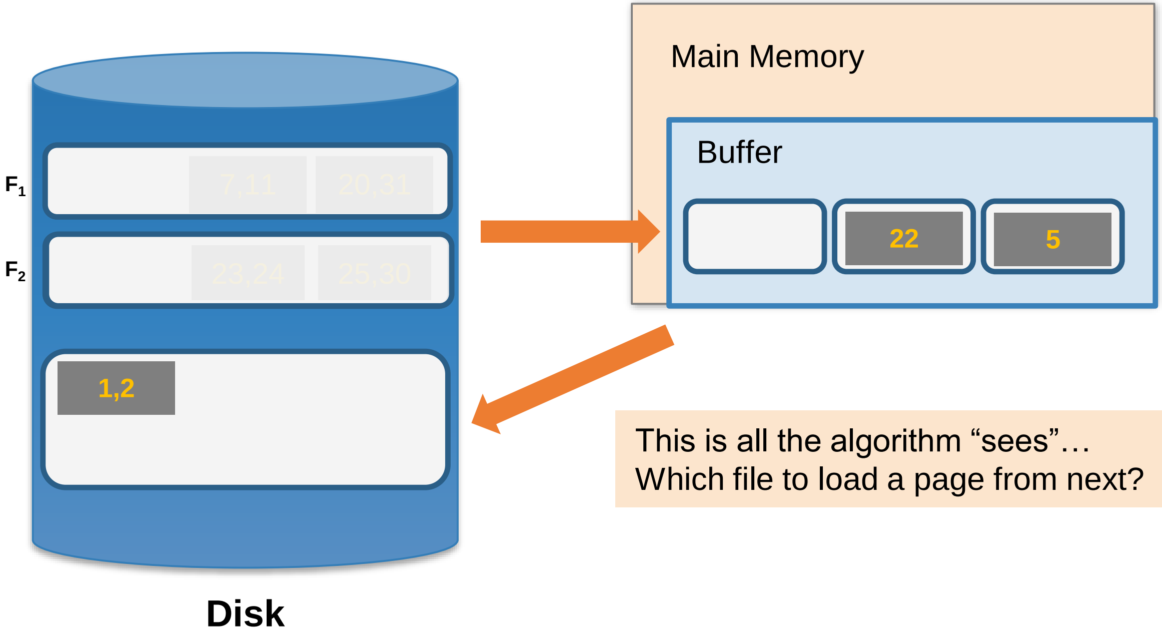
External Merge Algorithm



External Merge Algorithm

Input:
Two sorted files

Output:
One *merged* sorted file

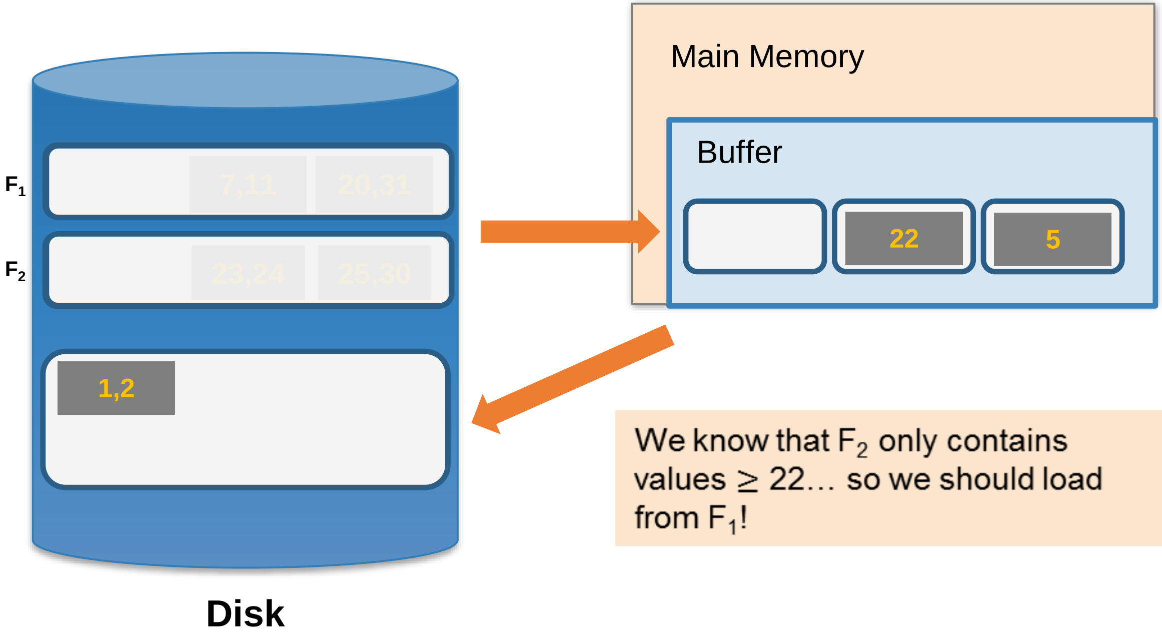


This is all the algorithm "sees"…
Which file to load a page from next?

External Merge Algorithm

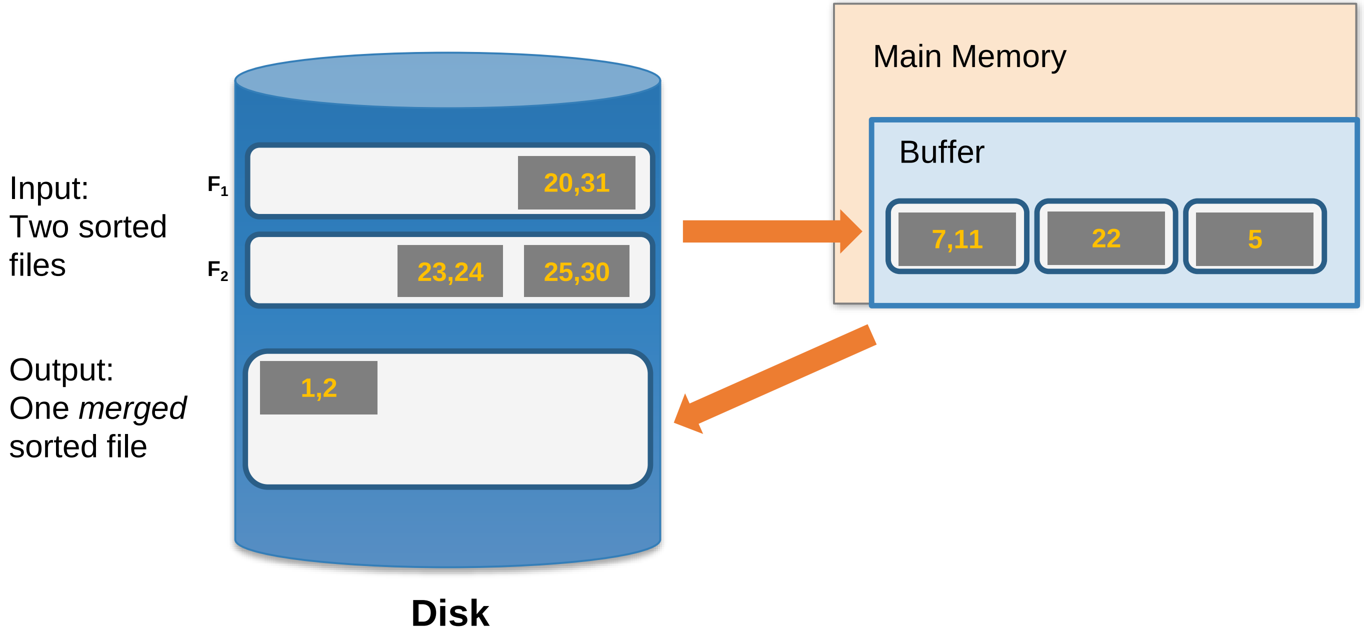
Input:
Two sorted files

Output:
One *merged* sorted file

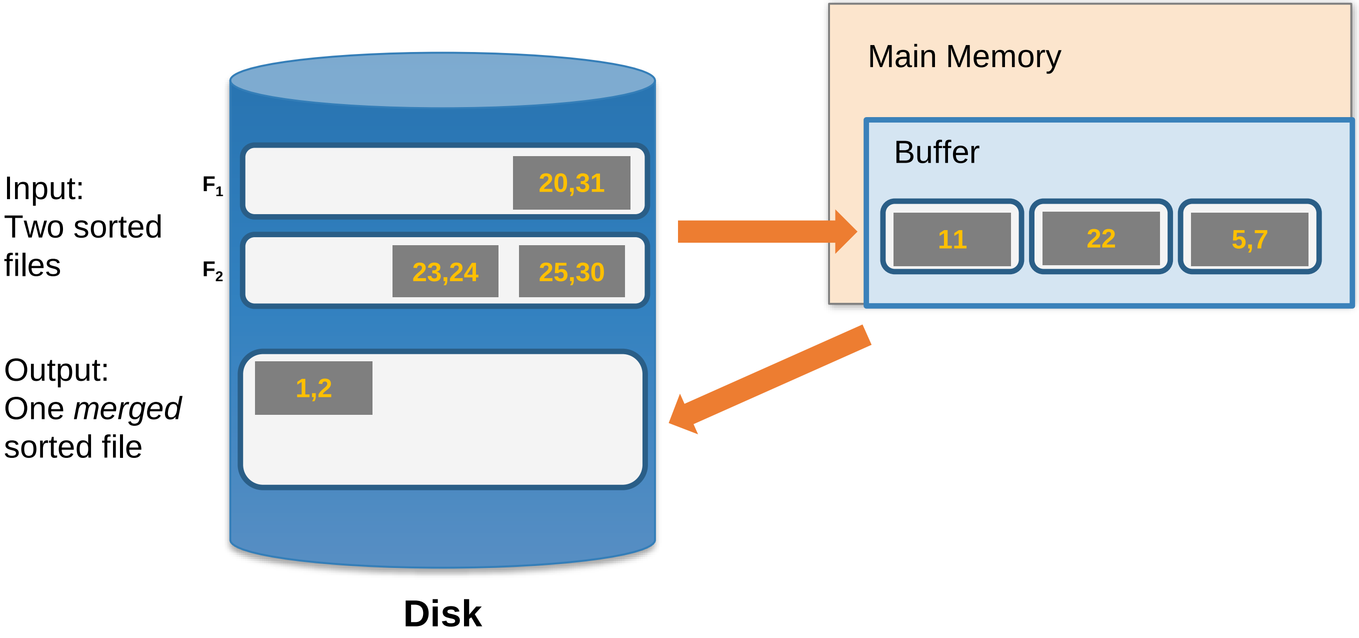


We know that F_2 only contains values $\geq$ 22… so we should load from F_1!

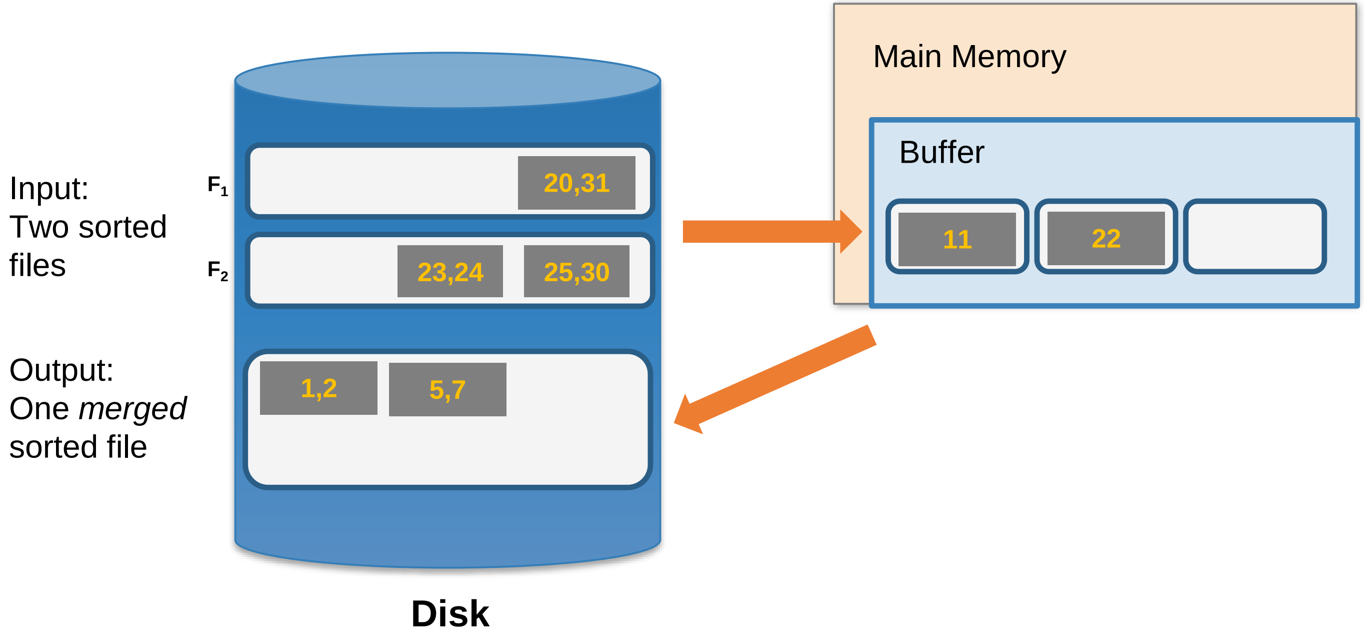
External Merge Algorithm



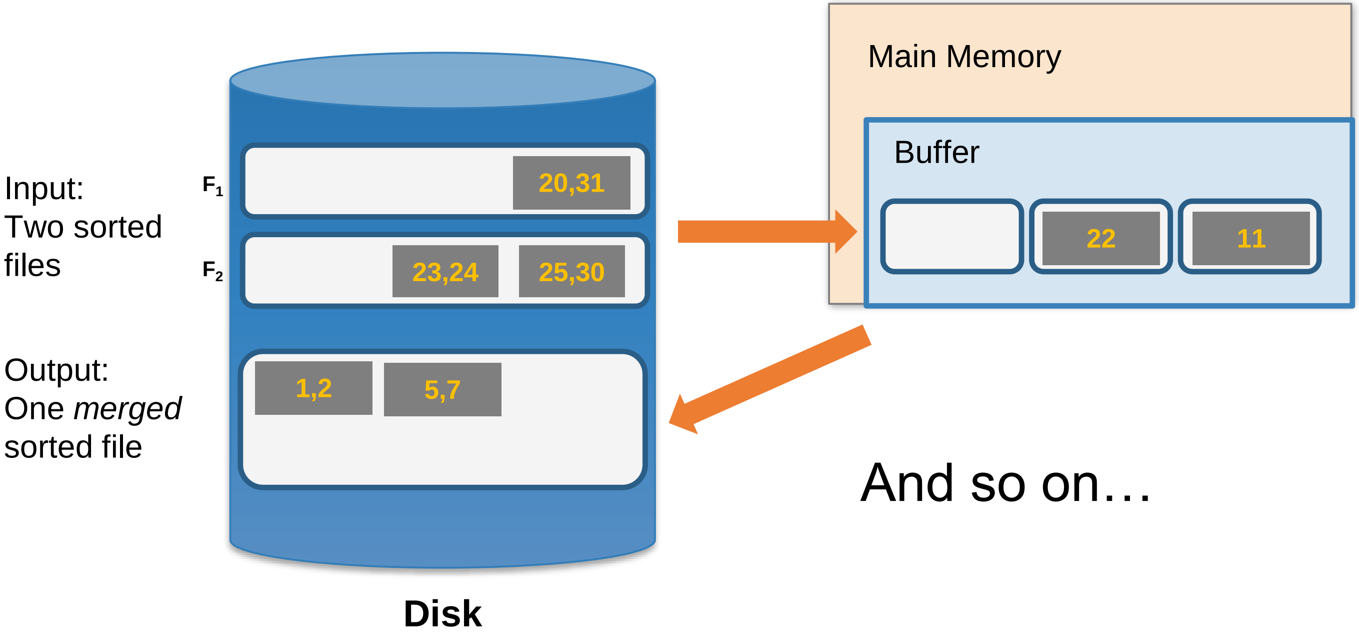
External Merge Algorithm



External Merge Algorithm



External Merge Algorithm



And so on…

We can merge lists of arbitrary length with only 3 buffer pages.

If lists of size M and N, then
Cost: 2(M+N) IOs
Each page is read once, written once

With B+1 buffer pages, can merge B lists. How?

Recap: External Merge Algorithm

- Suppose we want to merge two **sorted** files both much larger than main memory (i.e. the buffer)

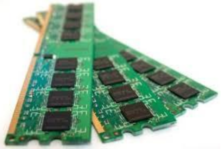
- We can use the **external merge algorithm** to merge files of ***arbitrary length*** in **2*(N+M) IO** operations with only **3 buffer pages**!

Our first example of an “IO aware” algorithm / cost model

Big Scaling (with Indexes)

Roadmap

Primary data structures/algorithms

	Hashing	Sorting	Counting
	HashTables ($hash_i$(key) --> value)	BucketSort, QuickSort MergeSort	HashTable + Counter ($hash_i$(key) --> <count>)
		MergeSortedFiles	
		?????	

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??**
  - How to Sort 10TB of data with 1GB of RAM…

A classic problem in computer science!

So how do we sort big files?

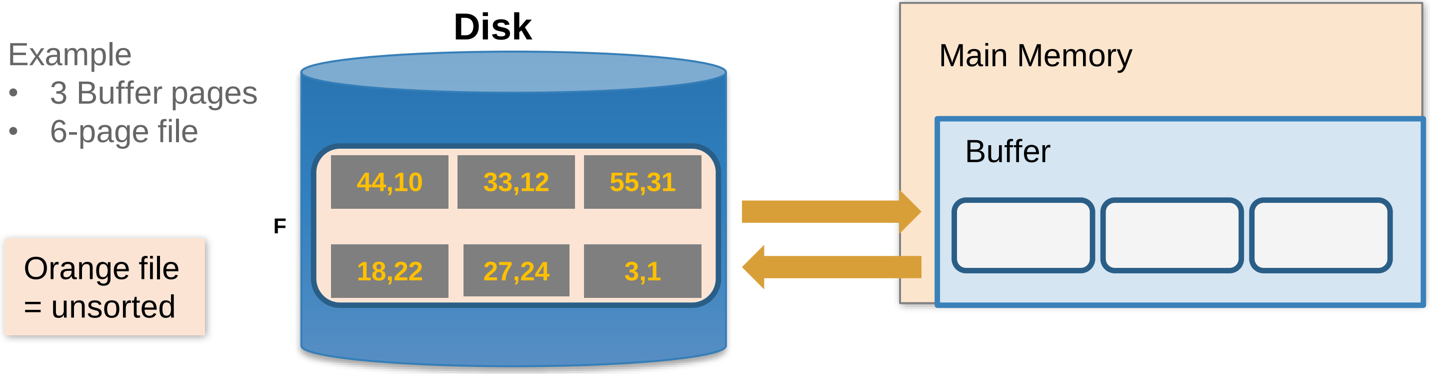
1. Split into chunks small enough to **sort in memory** ***("runs")***

1. **Merge** pairs (or groups) of runs with ***external merge algorithm***

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

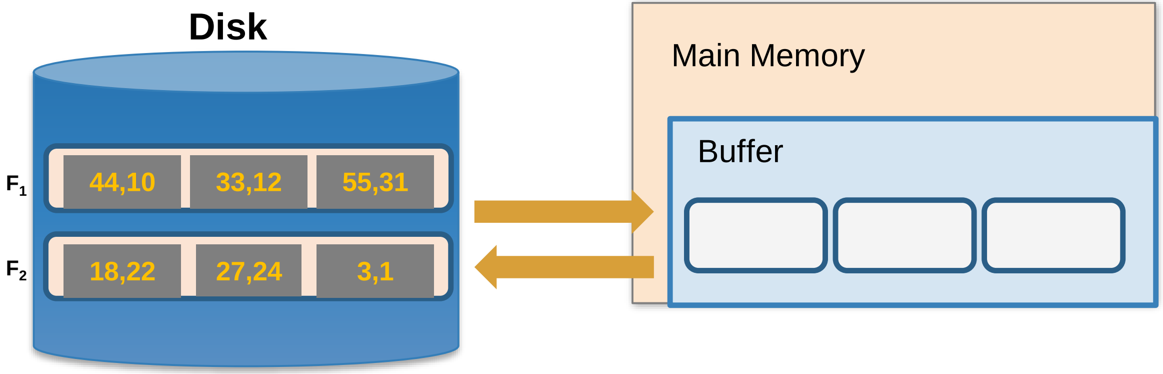
External Merge Sort Algorithm

Example
- 3 Buffer pages
- 6-page file



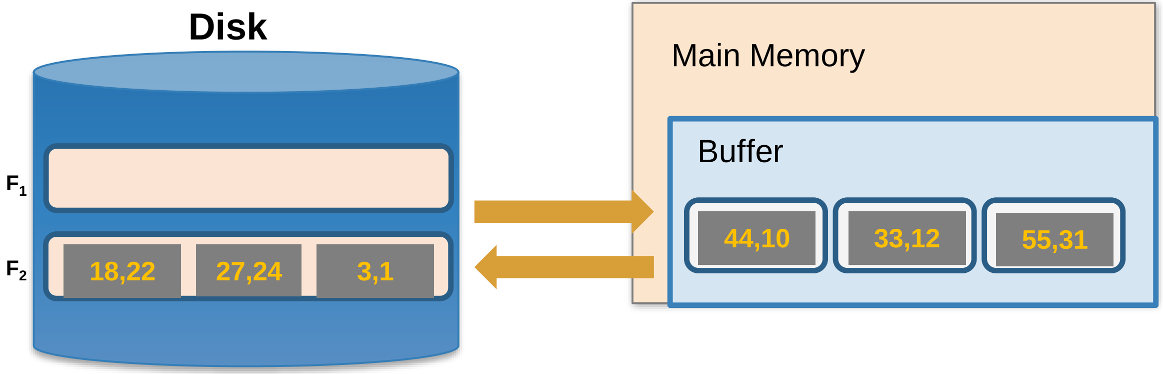
1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm



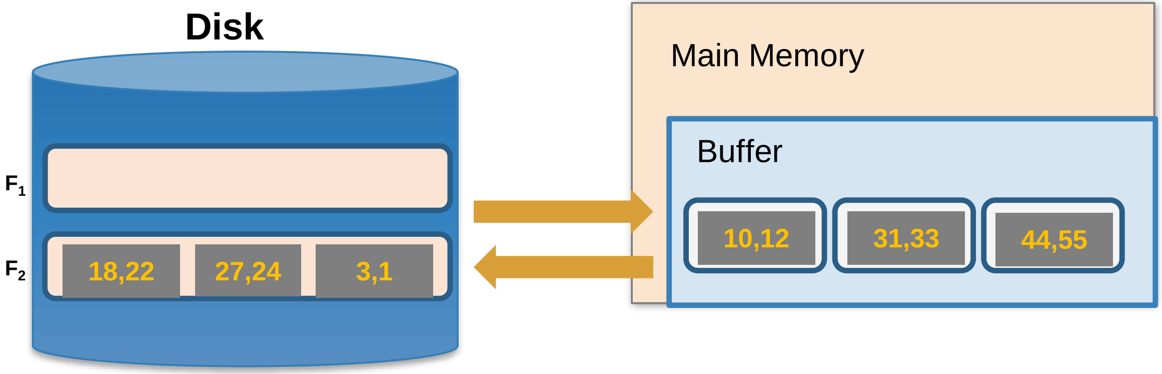
1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm



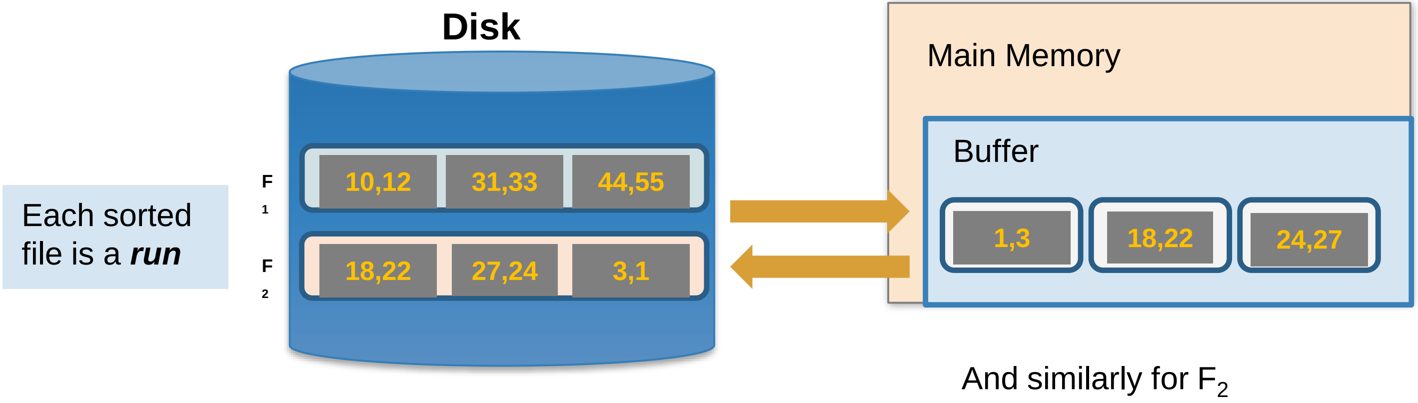
1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm



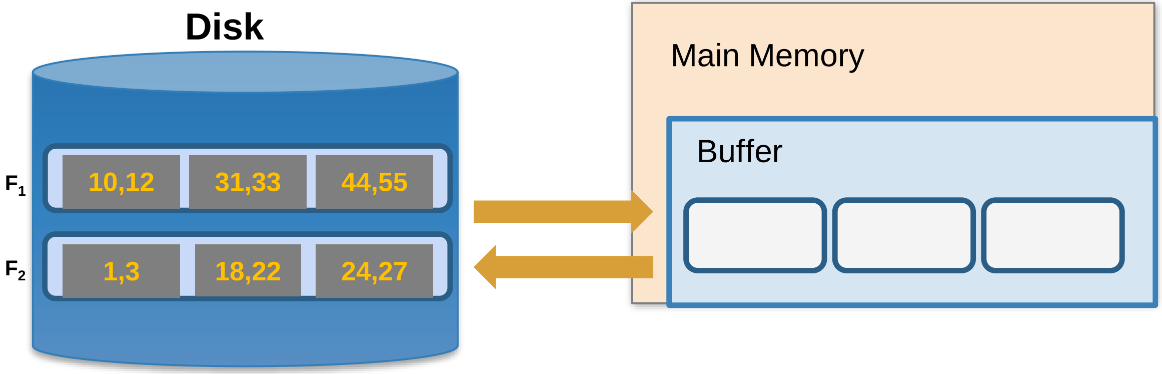
1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm



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

External Merge Sort Algorithm



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

Calculating IO Cost

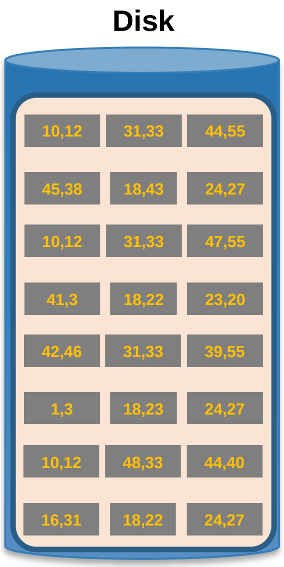
For 3 buffer pages, 6 page file:

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

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

1. Total cost = 24 IO

Running External Merge Sort on Larger Files

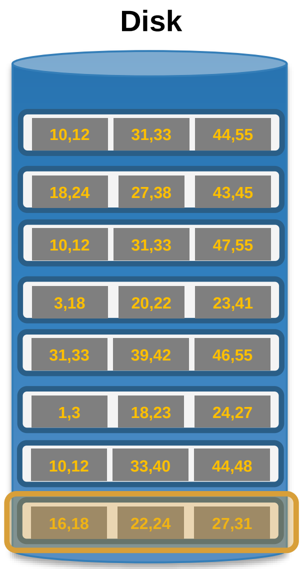


Assume we still only have *3* buffer pages *(Buffer not pictured)*

Running External Merge Sort on Larger Files

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

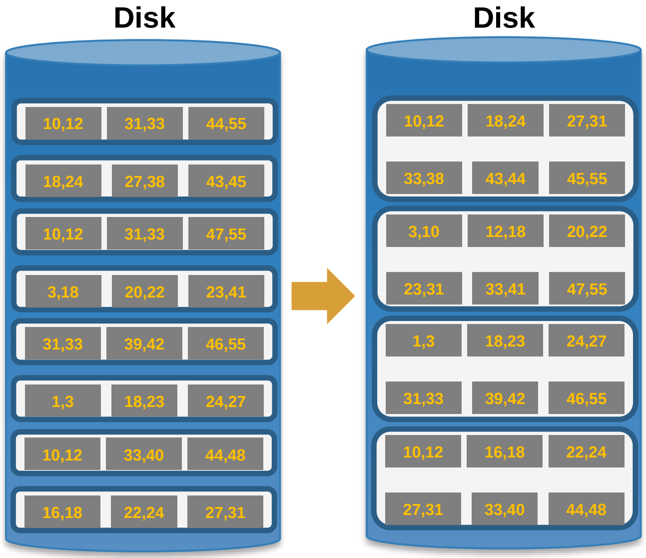
Running External Merge Sort on Larger Files



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

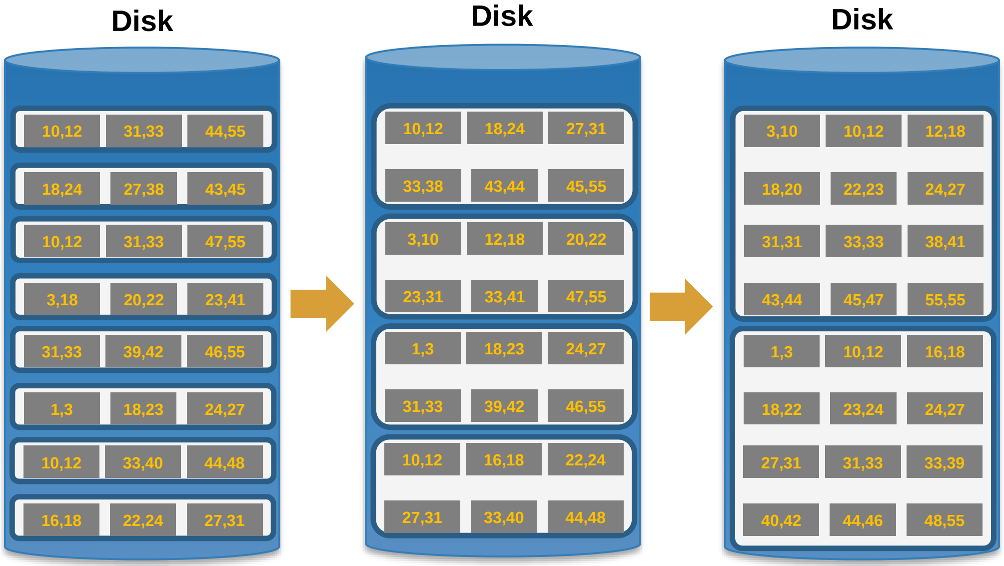
Each sorted file is a ***run***

Running External Merge Sort on Larger Files



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

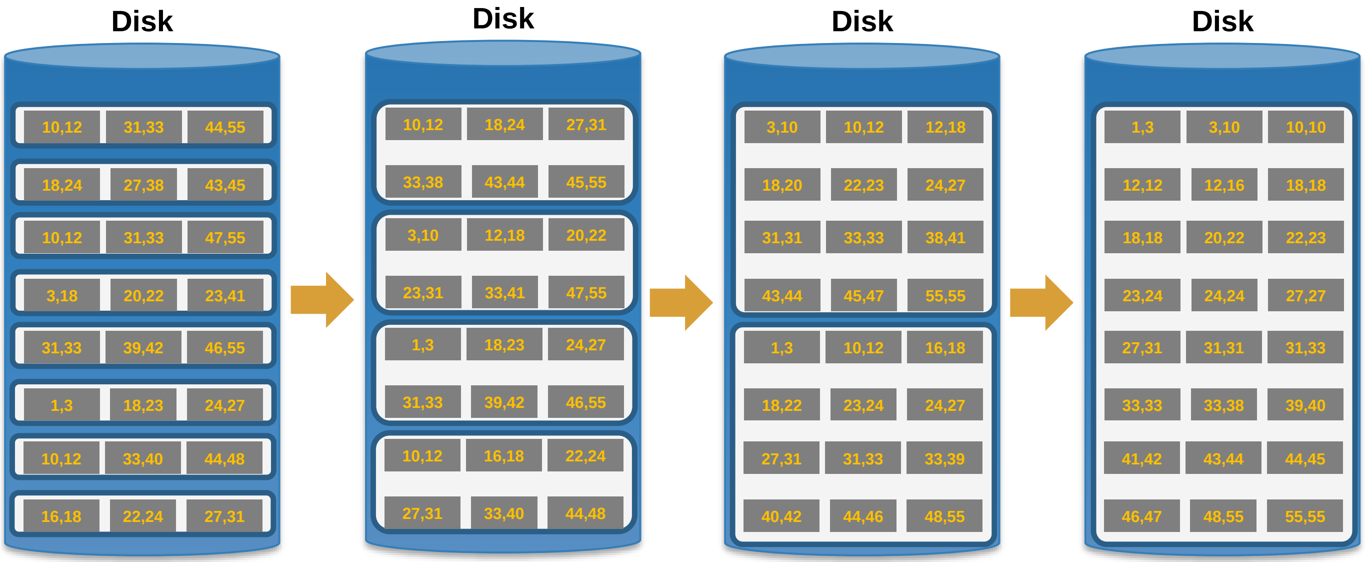
Running External Merge Sort on Larger Files



3. And repeat…

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

Running External Merge Sort on Larger Files

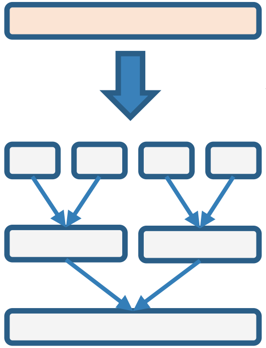


4. And repeat!

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

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

<u>IO Cost:</u>

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

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>

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

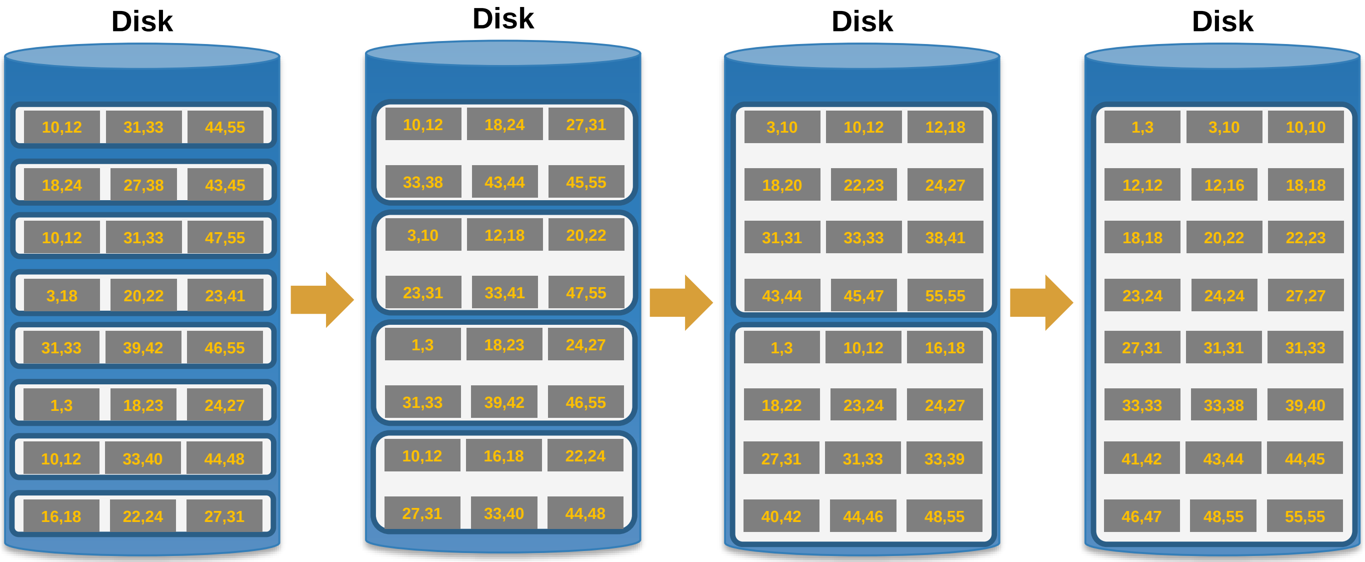
Starting with runs of length 1

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

Performing ***B***-way merges

Pretty fast IO aware sort !!

Repacking for longer runs (Optimization)

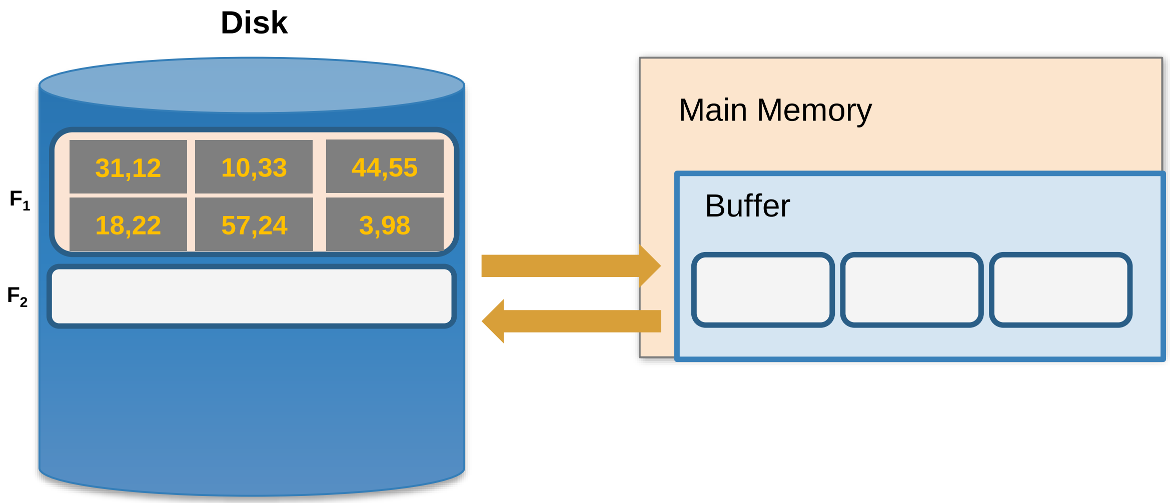


Idea: What if it’s already ‘partly’ sorted?

Can we be smarter with buffer?

Repacking Example: 3 page buffer

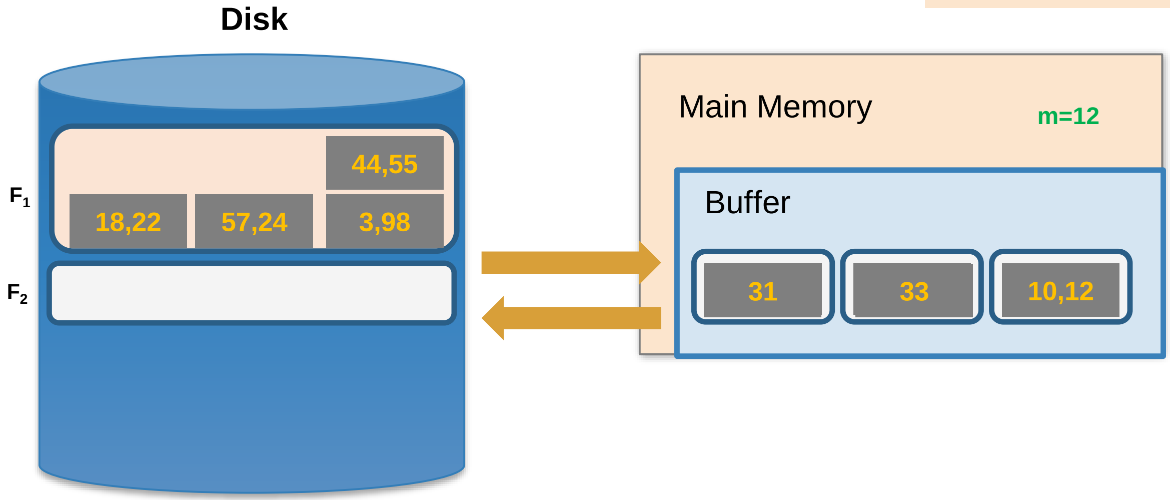
- Start with unsorted single input file, and load 2 pages



Repacking Example: 3 page buffer

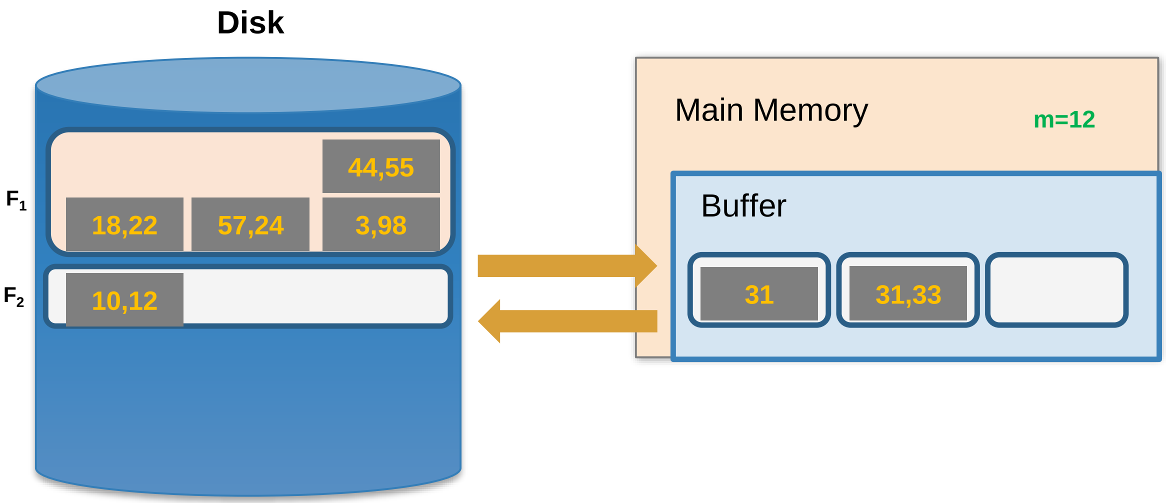
- Take the minimum two values, and put in output page

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



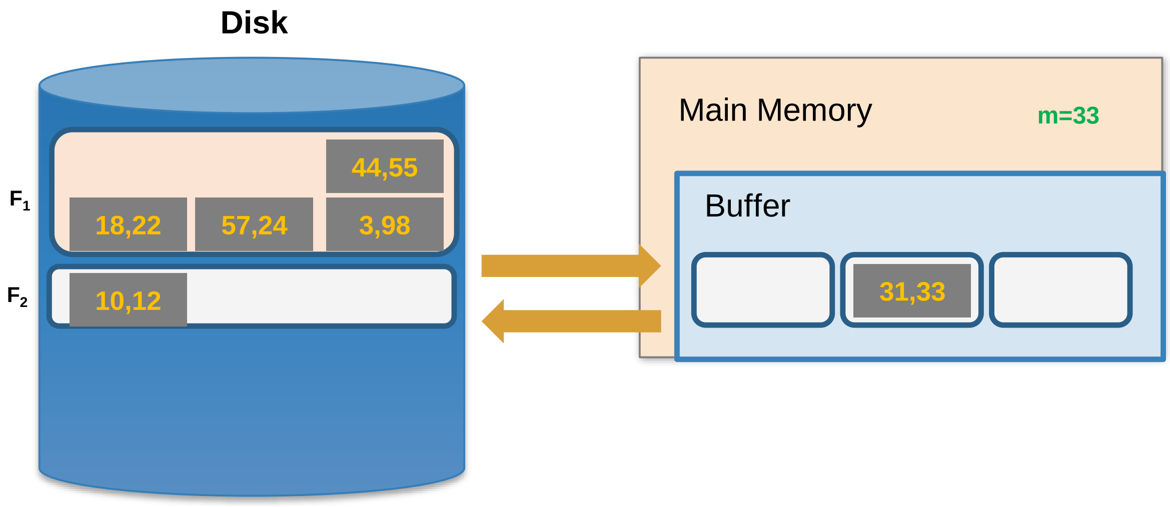
Repacking Example: 3 page buffer

- Next, ***repack***



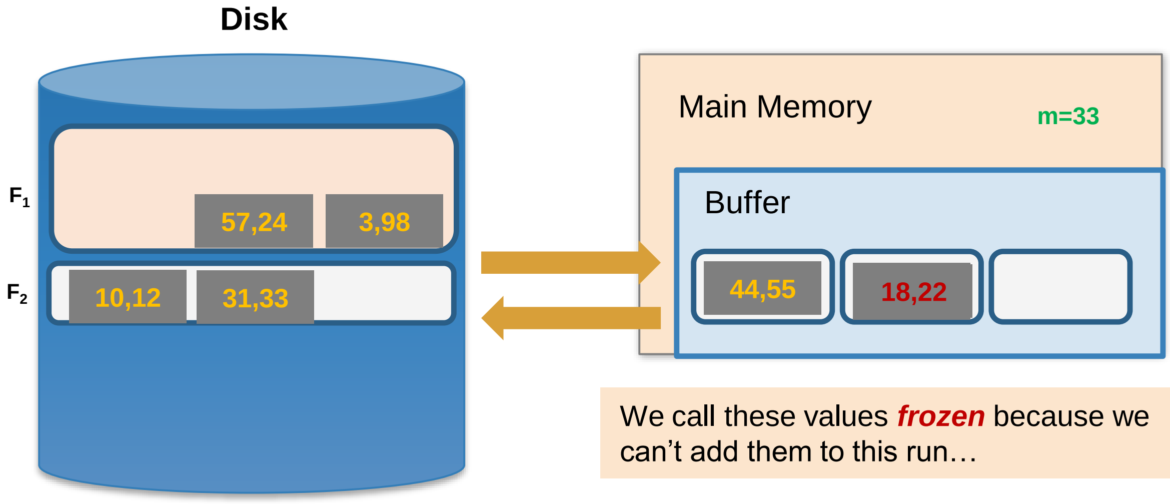
Repacking Example: 3 page buffer

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



Repacking Example: 3 page buffer

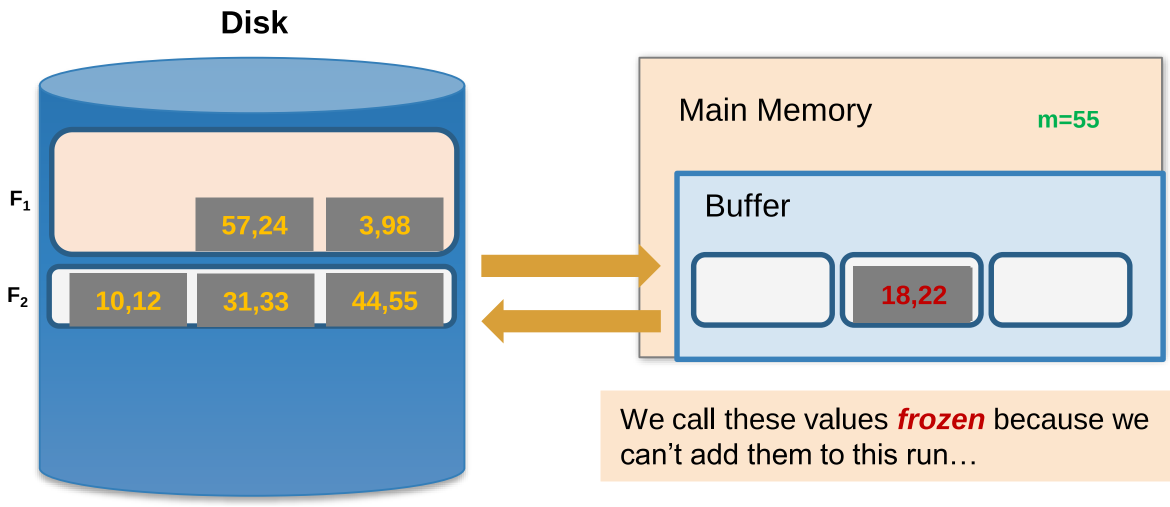
- Now, however, ***the smallest values are less than the largest (last) in the sorted run…***



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

Repacking Example: 3 page buffer

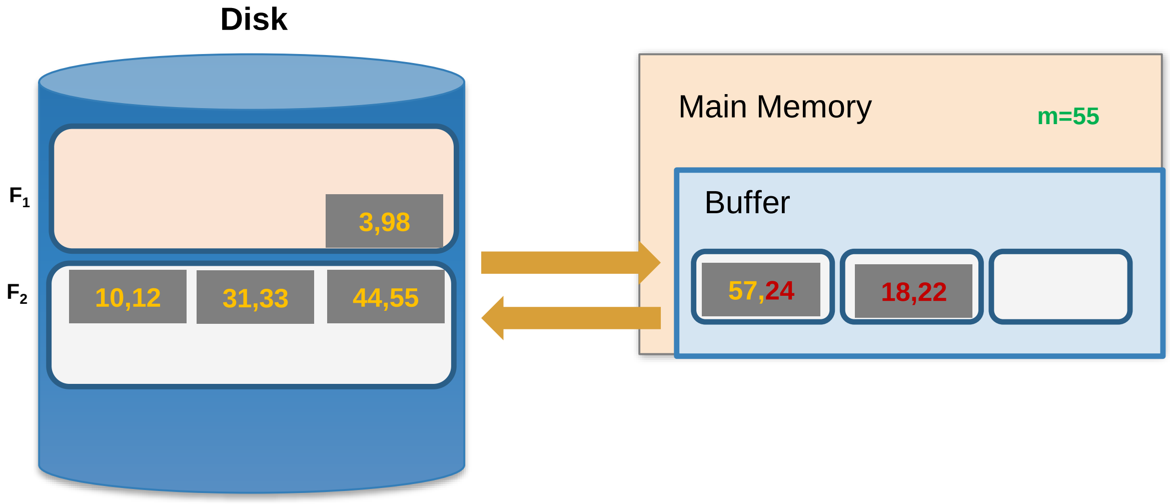
- Now, however, ***the smallest values are less than the largest (last) in the sorted run…***



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

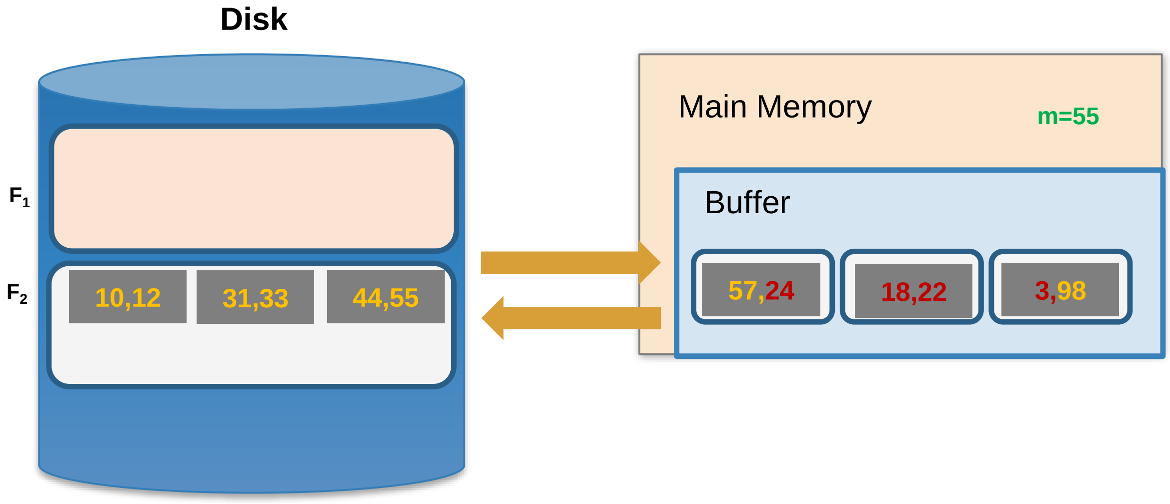
Repacking Example: 3 page buffer

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



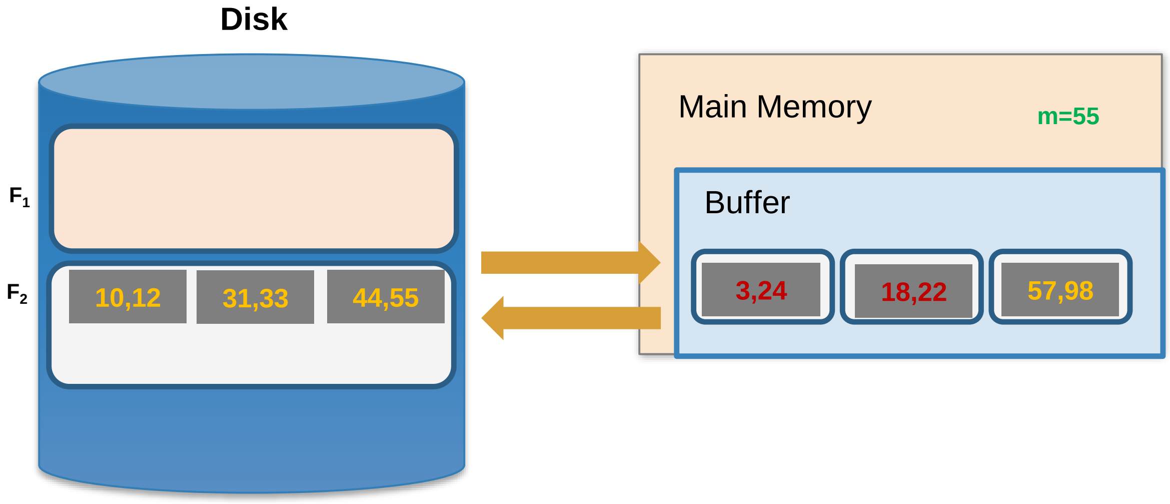
Repacking Example: 3 page buffer

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



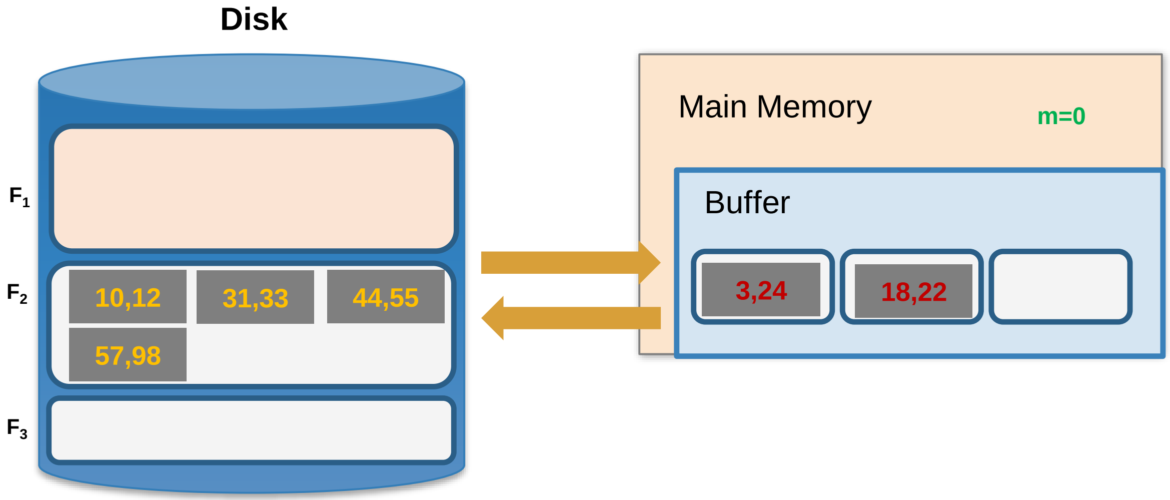
Repacking Example: 3 page buffer

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



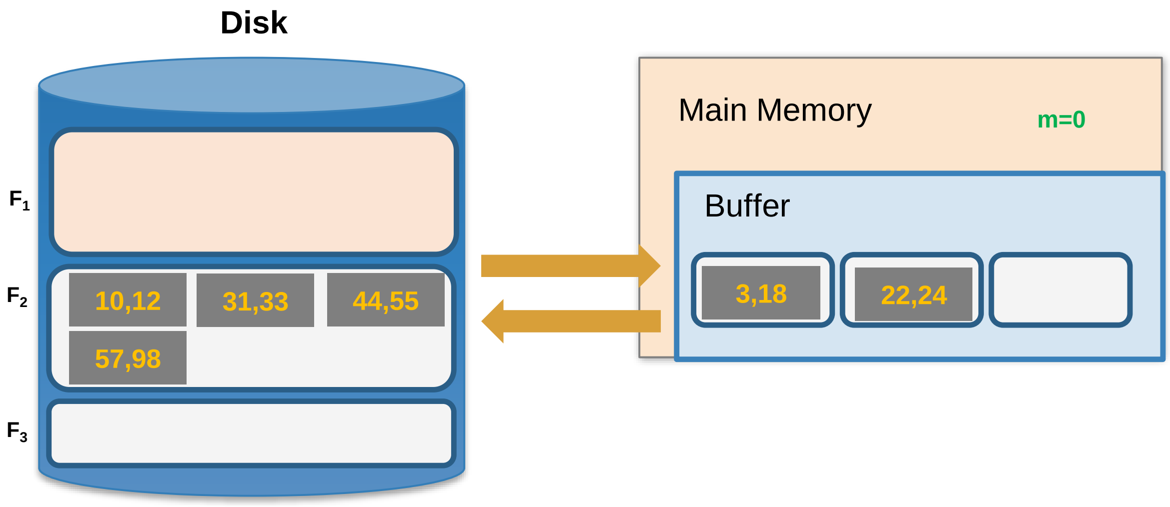
Repacking Example: 3 page buffer

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



Repacking Example: 3 page buffer

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



Repacking

- Note that, for buffer with B+1 pages:
  - **Best case:** If input file is sorted → nothing is frozen → we get **a single** run!
  - **Worst case:** If input file is reverse sorted → everything is frozen → we get runs of length **B+1**

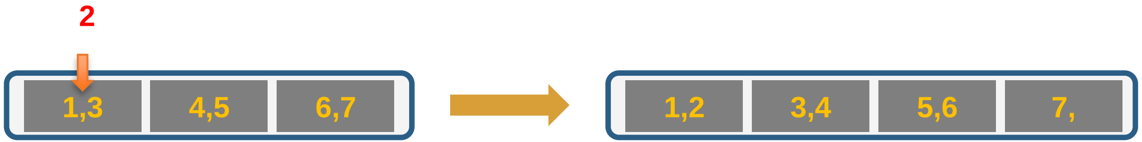
- In general, with repacking we do **no worse** than without it!

- Engineer’s approximation: runs will have **~2(B+1)** length

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

Sorting, with insertions?

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



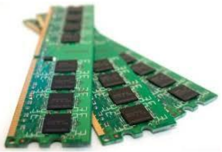
- We would have to potentially shift ***N*** records, requiring up to **~ 2*N/P** IO operations (where P = # of records per page)!
  - We could leave some “slack” in the pages…

Could we get faster insertions?
(next section)

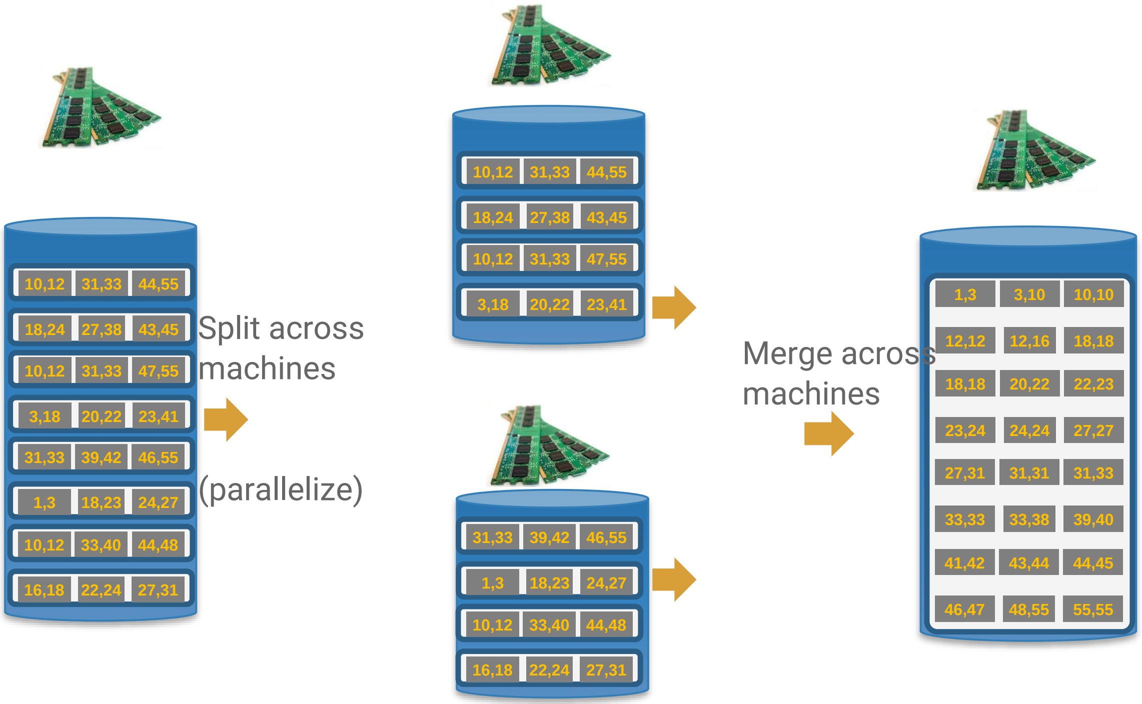
Big Scaling (with Indexes)

Roadmap

Primary data structures/algorithms

	Hashing	Sorting	Counting
	HashTables ($hash_i$(key) --> value)	BucketSort, QuickSort MergeSort	HashTable + Counter ($hash_i$(key) --> <count>)
		MergeSortedFiles SortFiles	
		?????	

Scaling, Speeding Sort (in Cluster)



MergeSort locally in each machine (in parallel)

Notes
- Use N machines (N >= 2)
- Could reuse machines
- Speedup at cost of network bandwidth (especially with current data centers)

Example: AWS/GCP offer machine instances
(e.g, ec2.r5 offers 1-3GBps network bandwidth,
2CPU/16GB RAM to 96 CPU/768GB RAM for $-$$$$ in Nov’18)

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