1

Big Scale

Roadmap

Hashing

Sorting

Hashing-Sorting solves “all” known data scale problems :=)

+ Boost with a few patterns -- Cache, Parallelize, Pre-fetch

THE BIG IDEA

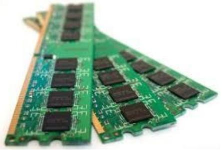
Note
Works for Relational, noSQL
(e.g. mySQL, postgres, BigQuery, BigTable, MapReduce, Spark)

2

Big Scale Lego Blocks

Roadmap

Primary data structures/algorithms



Hashing

HashTables
(hash$_i$(key) --> location)

HashFunctions
(hash$_i$(key) --> location)

HashFunctions
(hash$_i$(key) --> location)

Sorting

BucketSort, QuickSort
MergeSort

MergeSortedFiles

MergeSort

MergeSort

Why are Sort Algorithms Important?

Why not just use quicksort in main memory??

- How to Sort 10TB - 100 TB of data?
- E.g., with 1GB of RAM, i.e., 0.01-0.001% of data size…

A classic problem in computer science!

Example use cases

1. Query results in sorted order is extremely common
   - e.g., find students in increasing GPA order
2. Core building block for data compression, indexing, joins

External Merge Sort

So how do we sort big files?

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

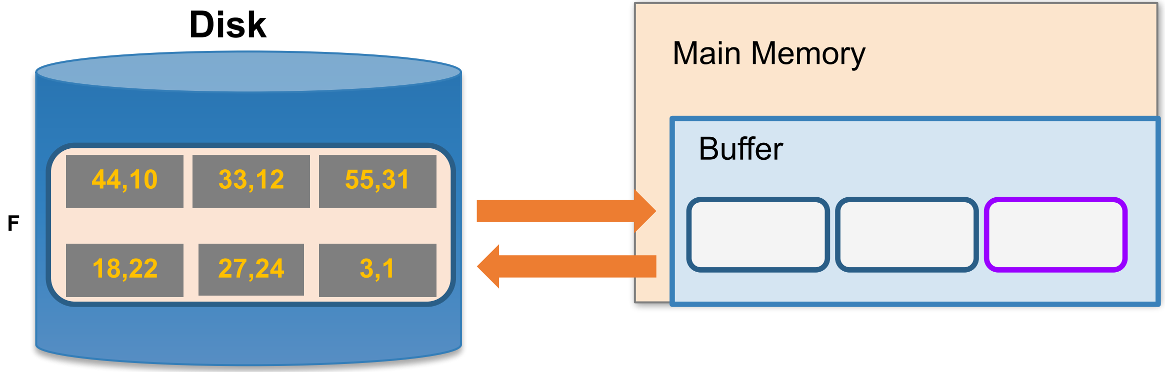
2. Merge groups of runs with *external merge algorithm*

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

External Merge Sort Algorithm

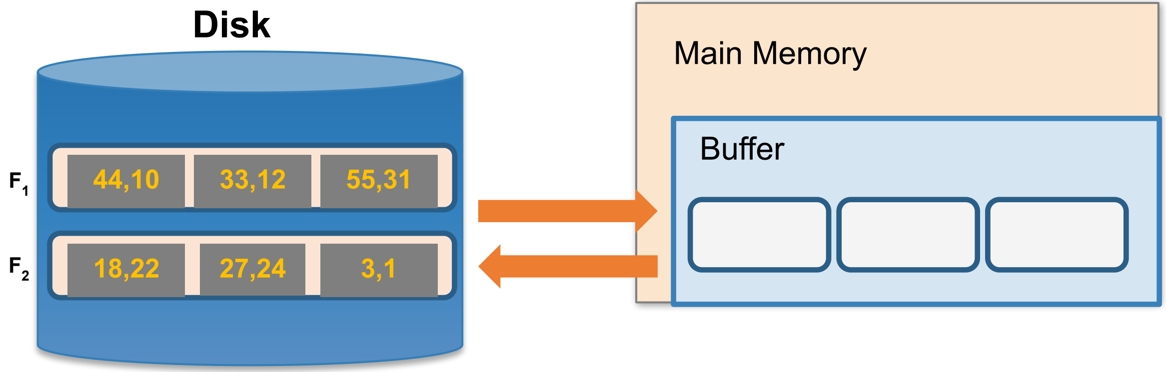
Example
- 3 Buffer pages
- 6-page file

Orange file = unsorted



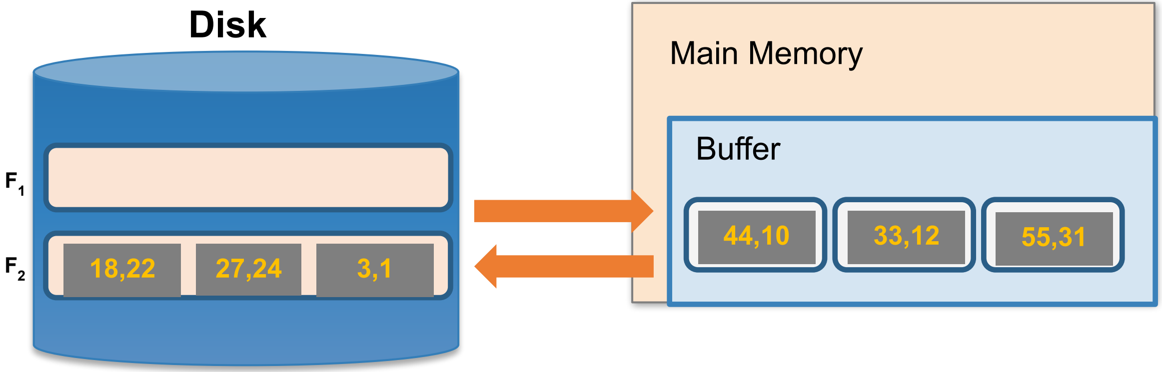
1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm



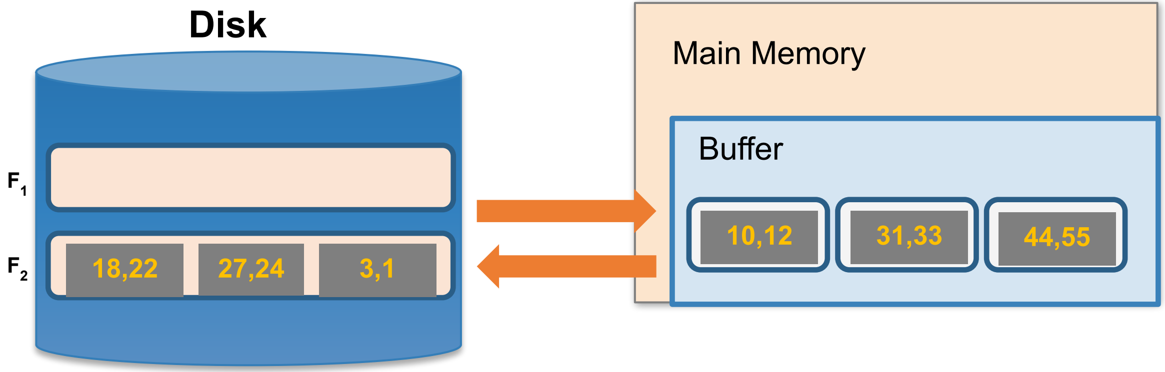
1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm



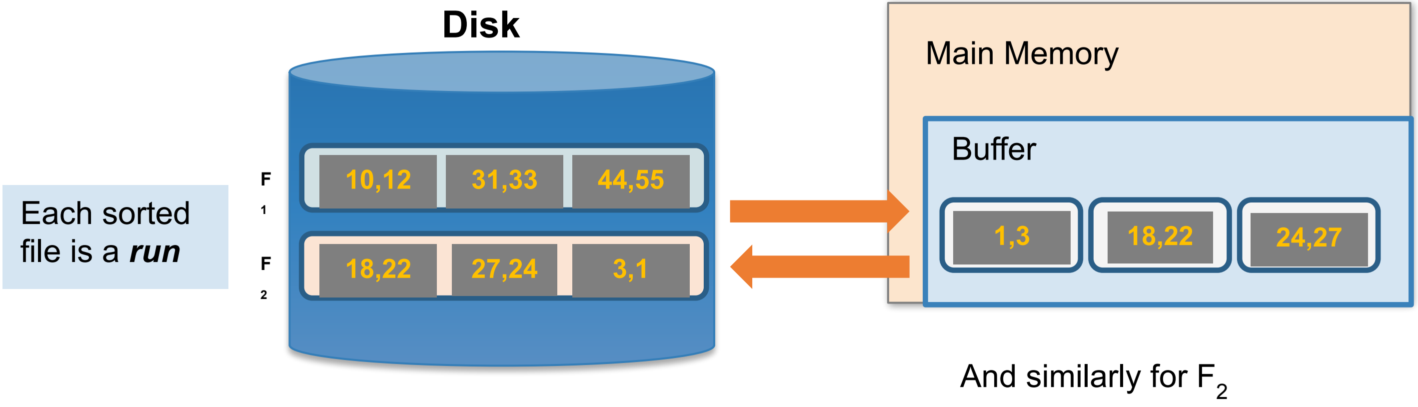
1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm



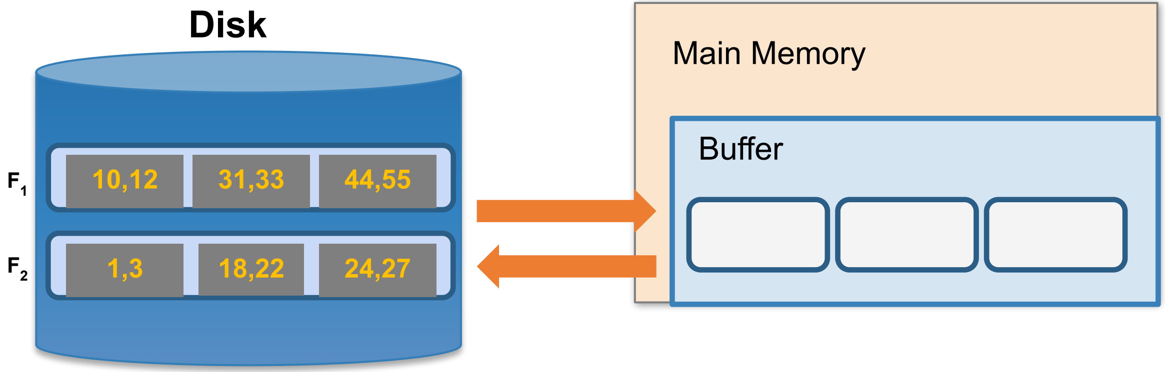
1. Split into chunks small enough to **sort in memory**

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

Assume: cost(Read) = cost(Write) = 1 IO
(Alternate examples in HMWK2)

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 = 6*(1R + 1W) = 12 IOs

2. Merge each pair of sorted chunks with *external merge algorithm*
   Recall: ExtMergeSort in [2(M+N), which is really [1R+1W]*(M+N)]*
   = [1R + 1W]*(3 + 3) = 2*6 = 12 IOs

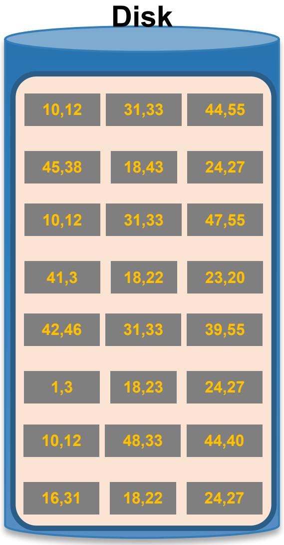
3. Total cost = 24 IO

<u>Note</u>: What's "IO" and how does it map to seek/scans?

- 24 IOs = 24 disk block read/writes
- Are disk blocks contiguous?
  - Cost = 1 seek + time to scan 24 blocks
  - Else, cost = 24 seeks + scan 24 blocks

⇒ For such problems, we'll use IO units for simplicity

Running External Merge Sort on Larger Files

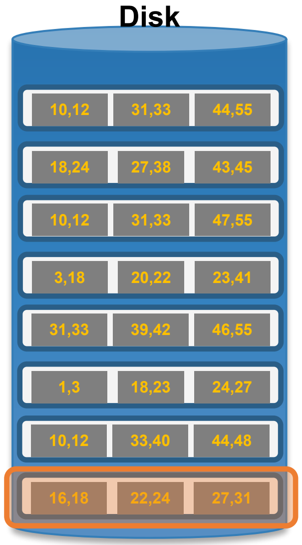


Assume we still only have *3* buffer pages *(Buffer not pictured)*

Running External Merge Sort on Larger Files

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

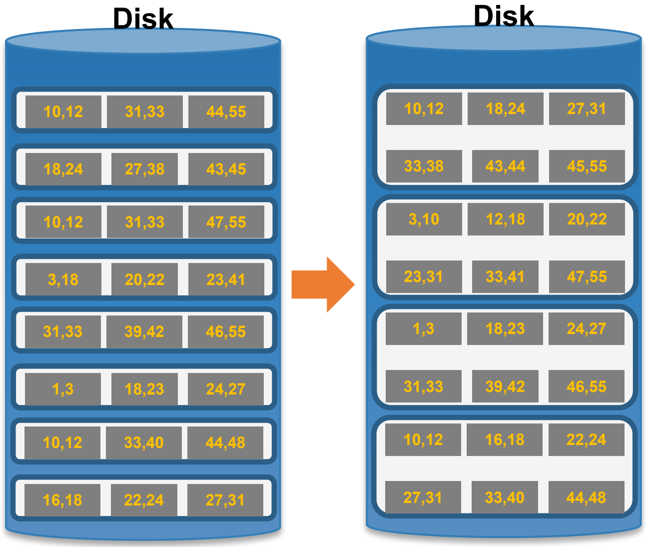
Running External Merge Sort on Larger Files



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

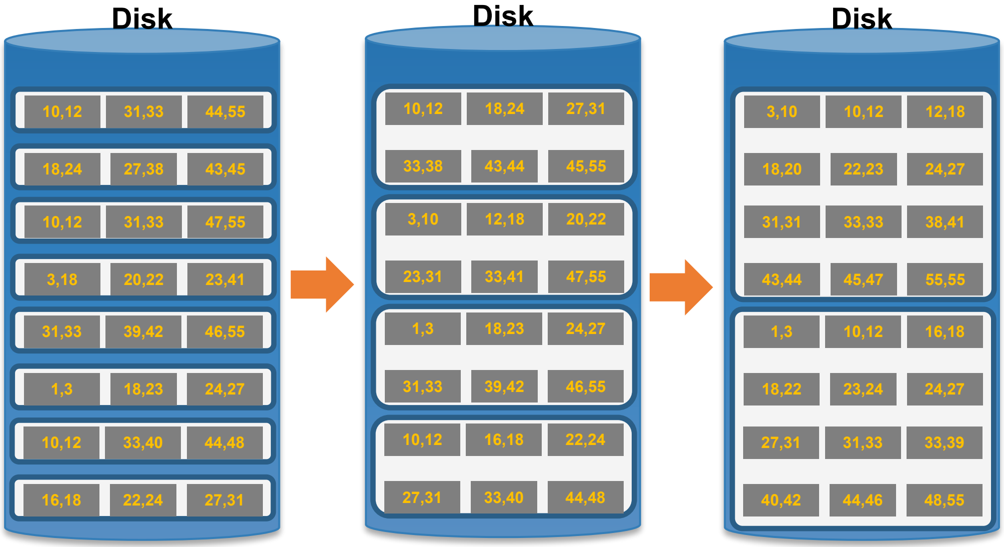
Each sorted file is a ***run***

Running External Merge Sort on Larger Files



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

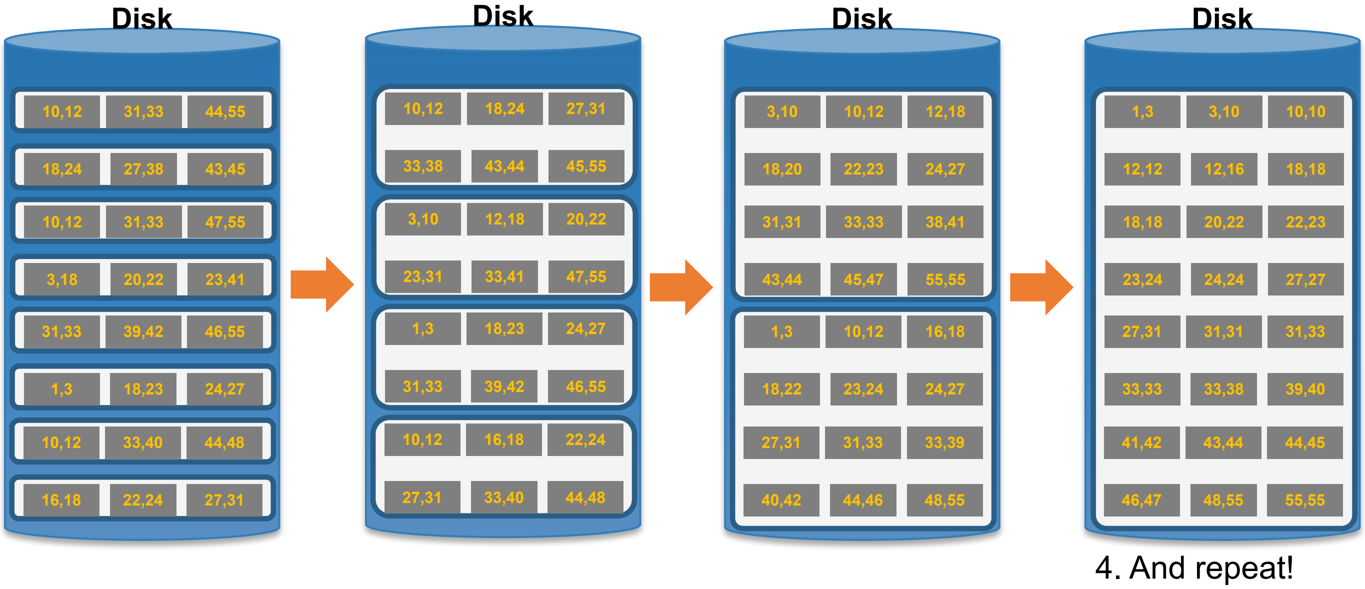
Running External Merge Sort on Larger Files



3. And repeat…

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

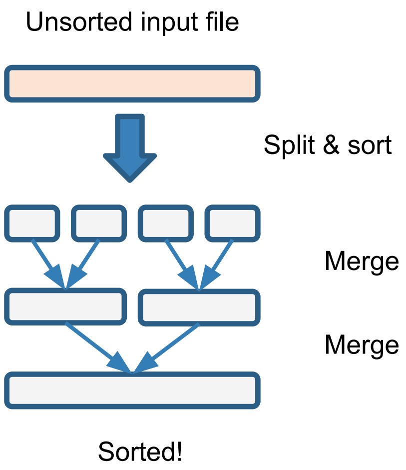
Running External Merge Sort on Larger Files



Simplified 3-page Buffer Version

For N page file, we do

- Sort step: Sort in 2N IOs
- Merge steps:
  - $\lceil \log_2 N \rceil$ passes
  - 2N IOs/pass
    (each page is read+write once)



→ **2N*($\lceil log_2 N \rceil$+1)** total IO cost!

External Merge Sort: Optimizations

Now assume we have **B+1** buffer pages (vs 3 pages in examples so far)

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!
Split the N pages into runs of length B+1 and sort these in memory

2. **Perform a B-way merge**.
On each pass, merge groups of *B* runs at a time (vs. merging pairs of runs)!

<u>IO Cost:</u>

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

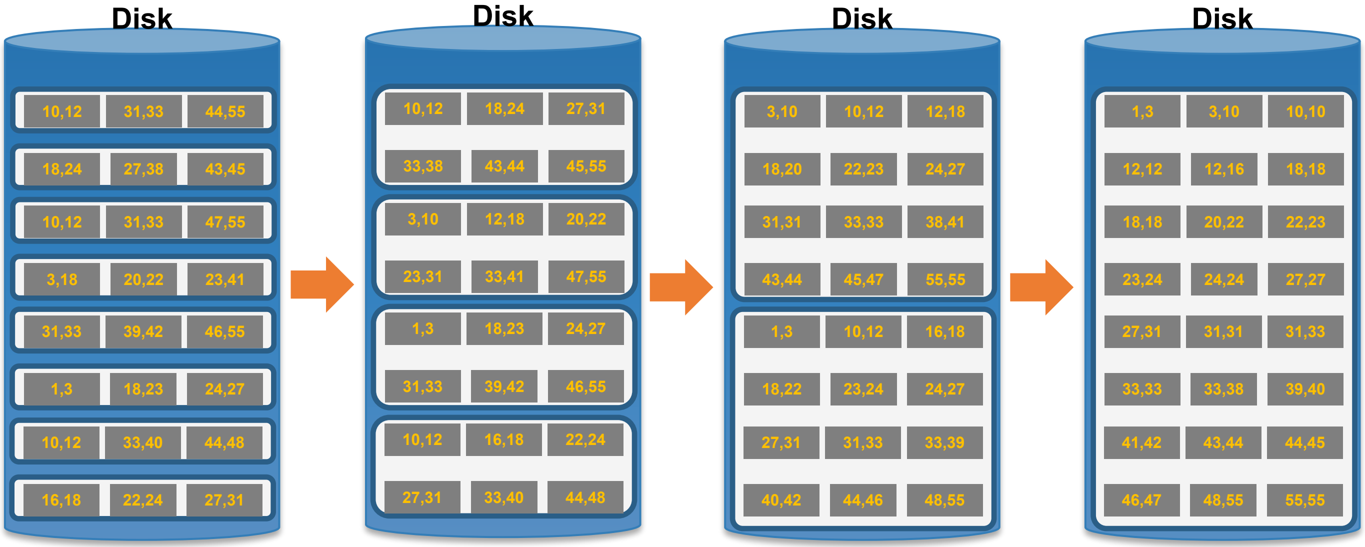
Starting with runs of length 1

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

Performing ***B*****-**way merges

Pretty fast IO aware sort !!

Repacking for longer runs (Optimization)

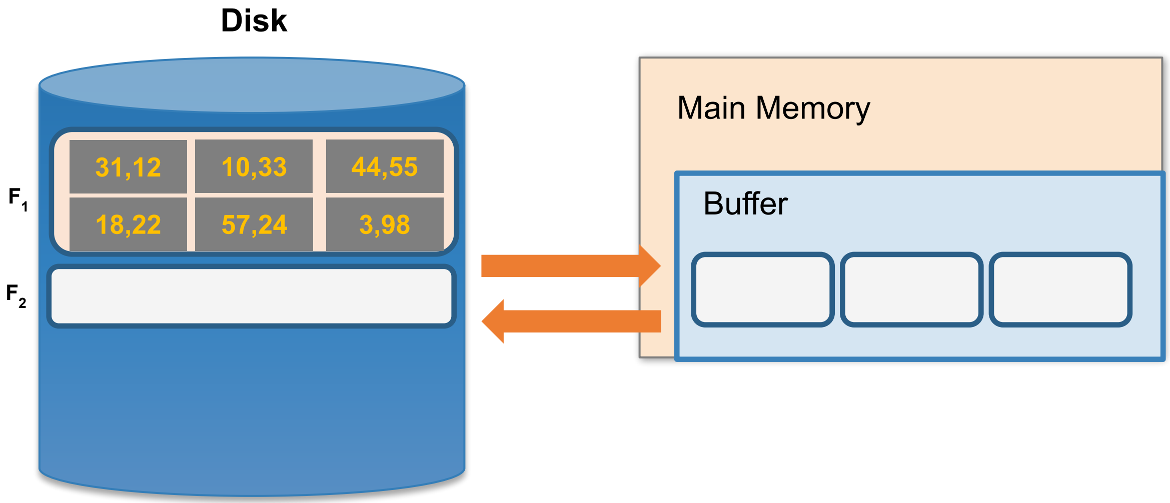


Idea: What if it's already 'nearly' or 'partly' sorted?

Can we be smarter with buffer? **Optimistic sorting**

Repacking Example: 3 page buffer

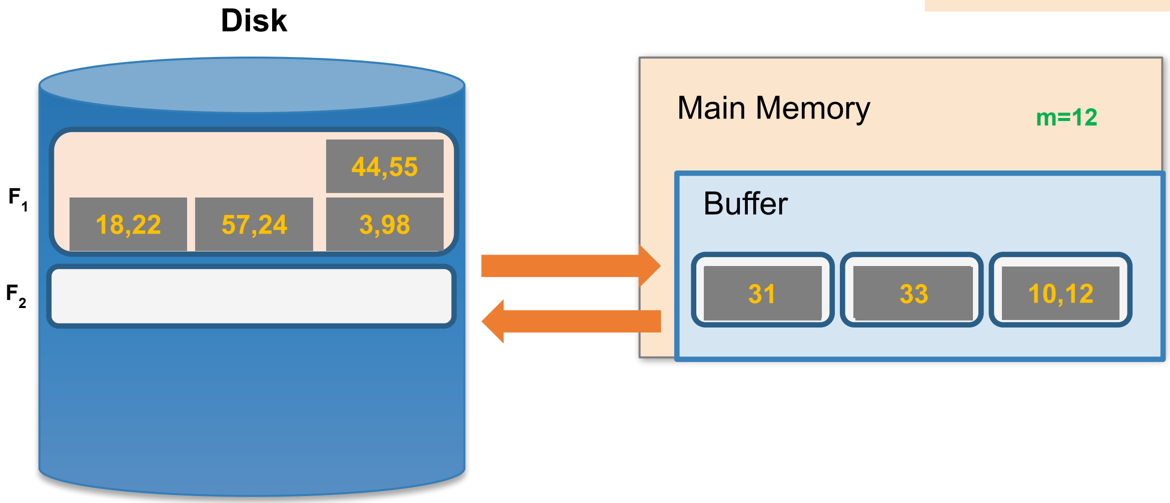
Start with unsorted single input file, and load 2 pages



Repacking Example: 3 page buffer

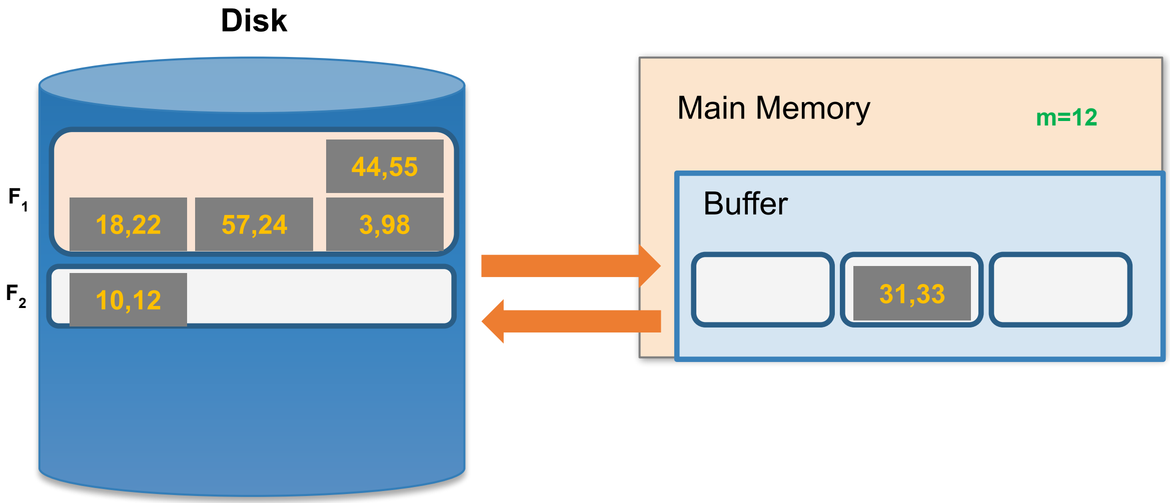
Take the minimum two values, and put in output page

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



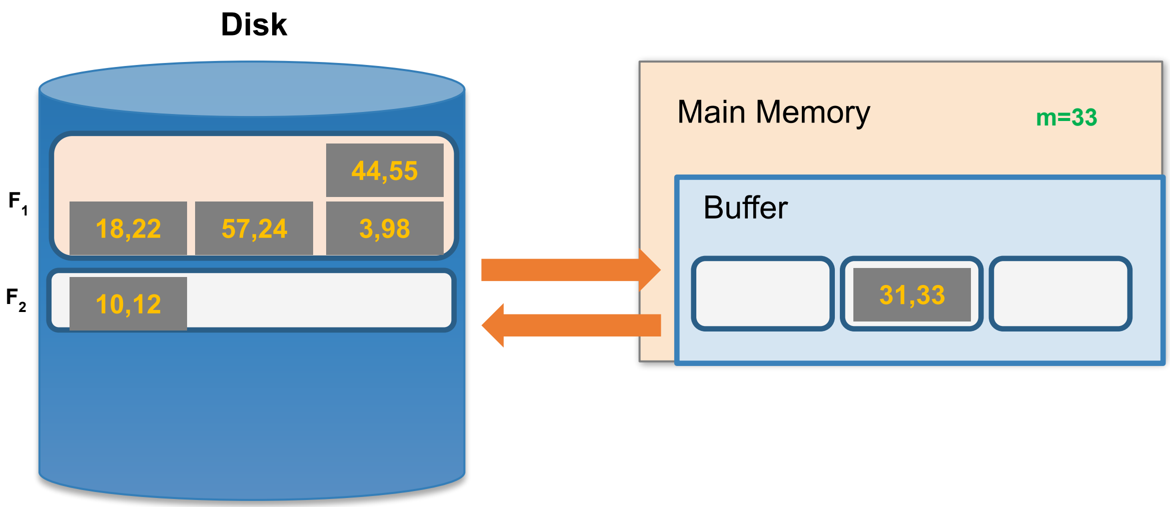
Repacking Example: 3 page buffer

- Next, ***repack***



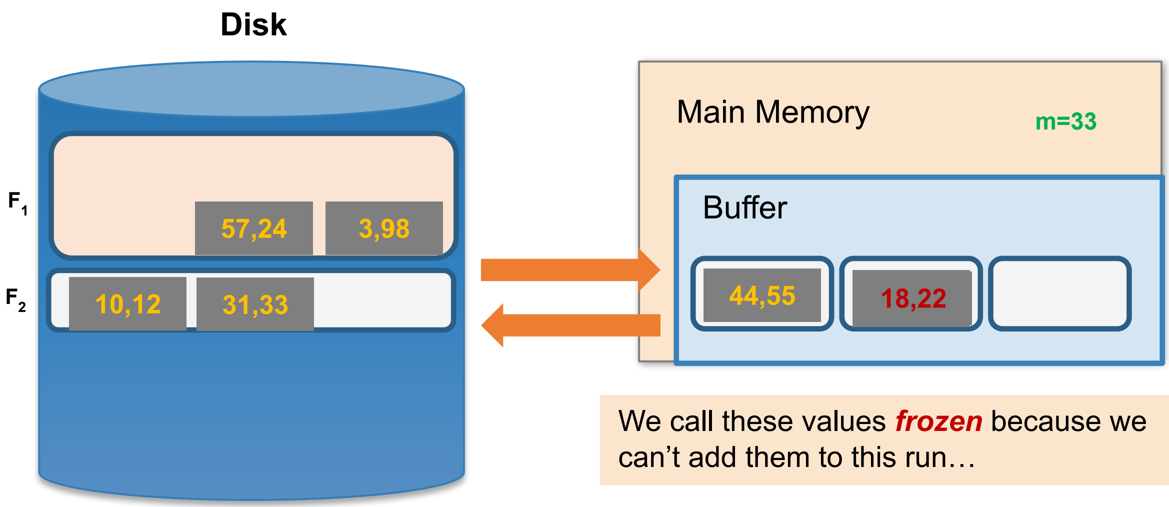
Repacking Example: 3 page buffer

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



Repacking Example: 3 page buffer

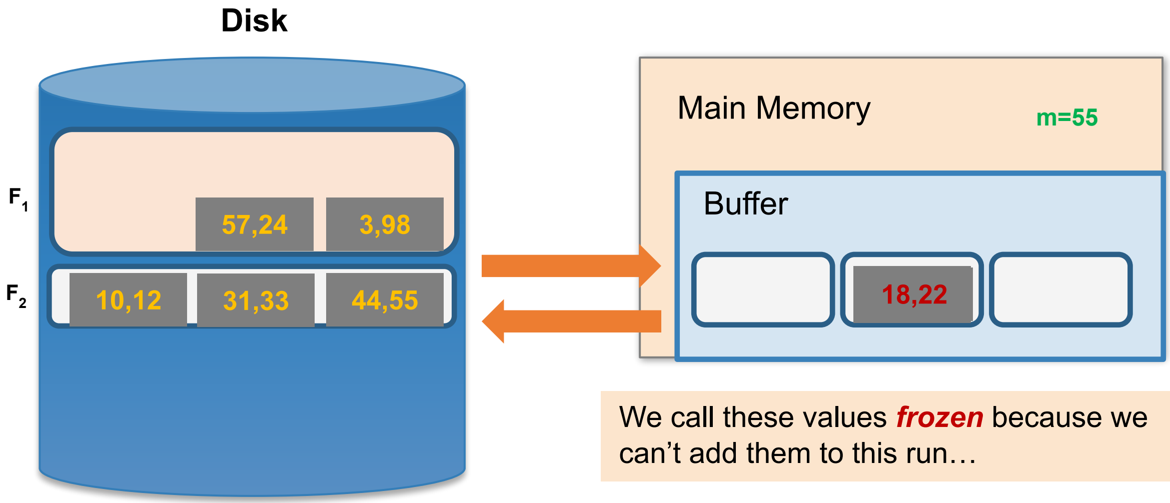
- Now, however, ***the smallest values are less than the largest (last) in the sorted run…***



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

Repacking Example: 3 page buffer

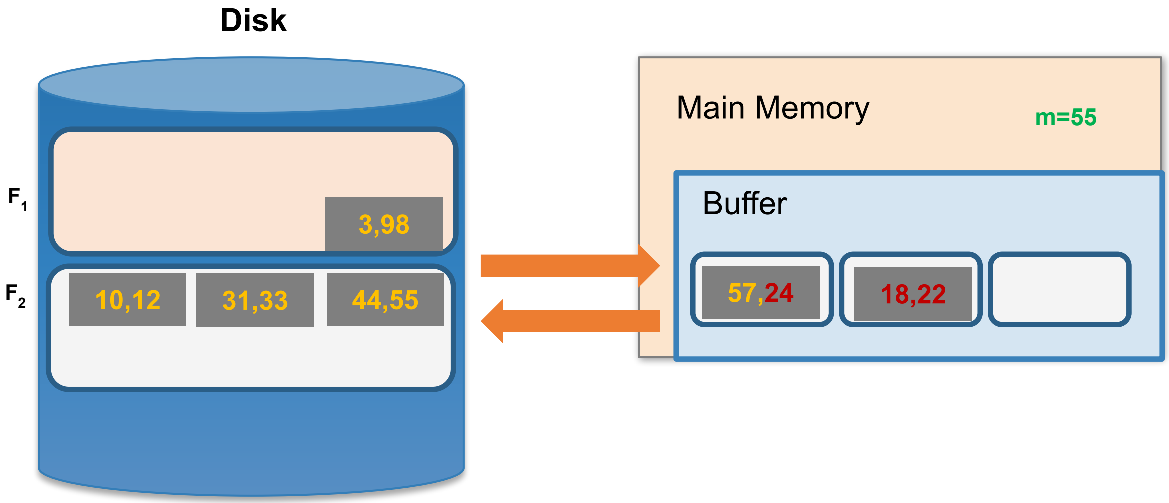
- Now, however, ***the smallest values are less than the largest (last) in the sorted run…***



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

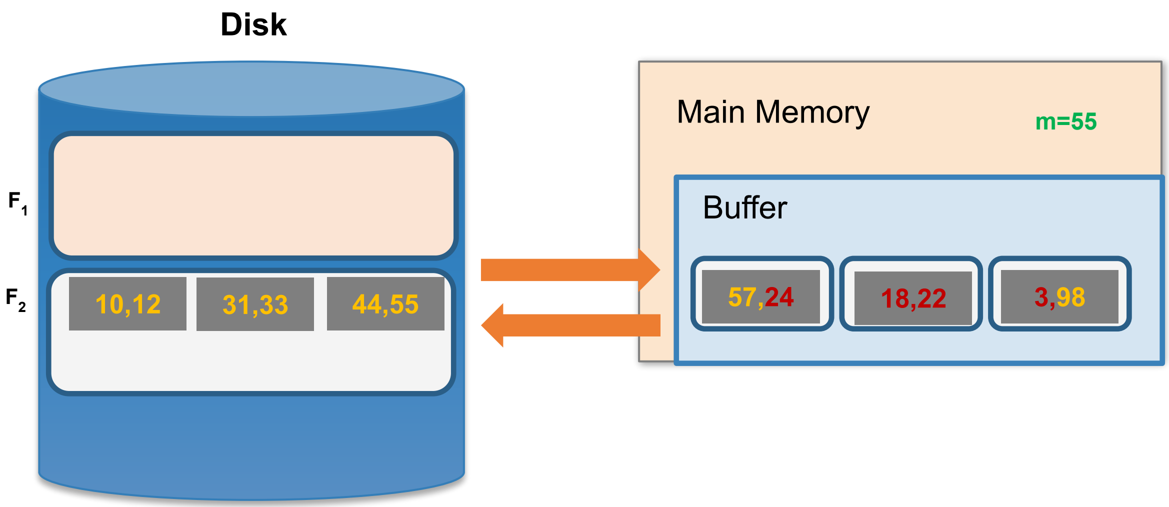
Repacking Example: 3 page buffer

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



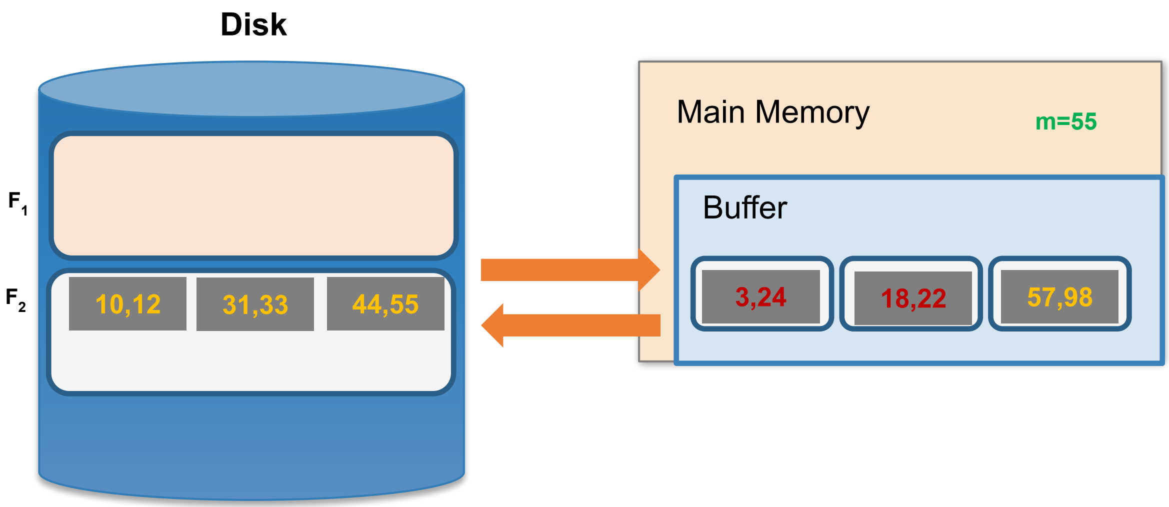
Repacking Example: 3 page buffer

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



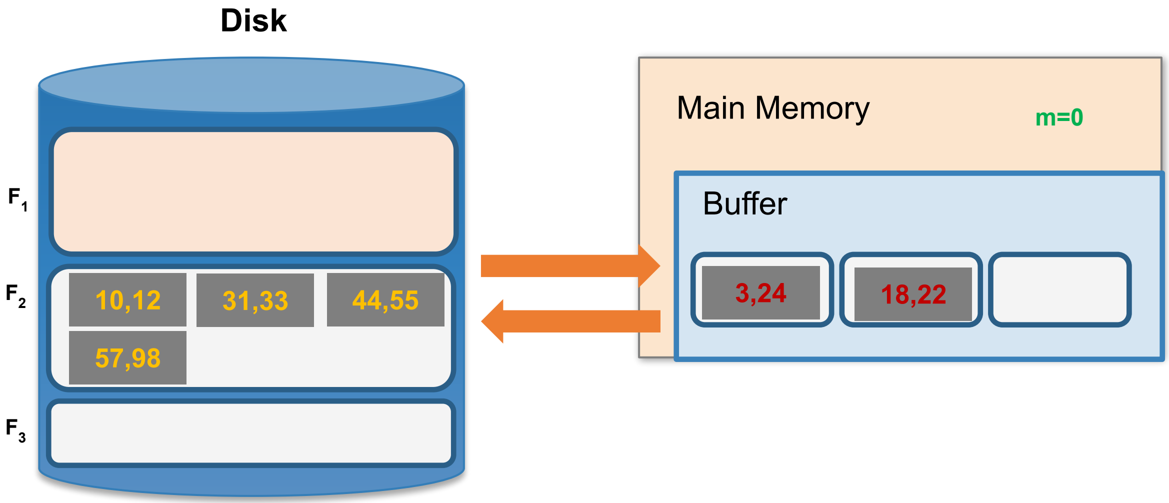
Repacking Example: 3 page buffer

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



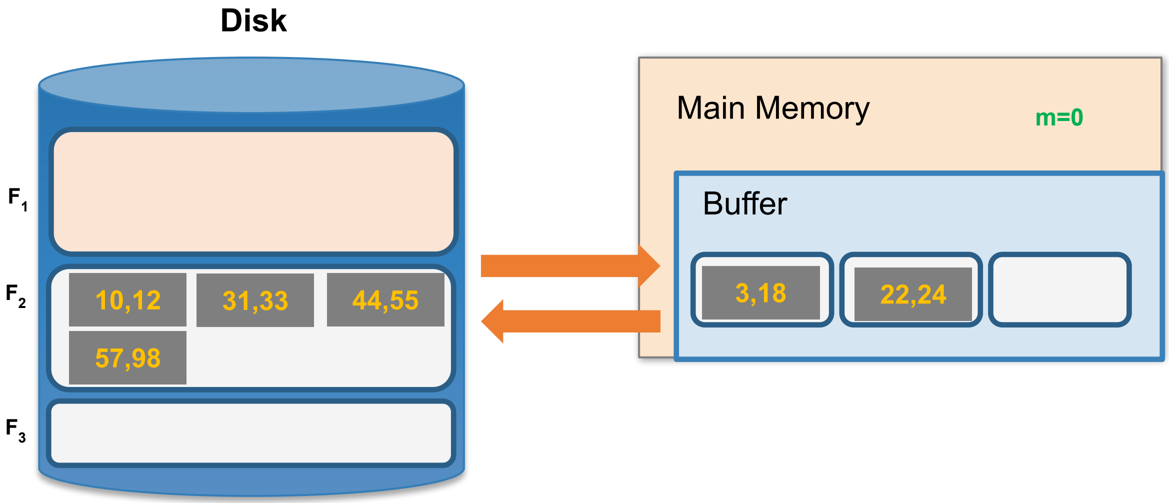
Repacking Example: 3 page buffer

- 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 <u>no worse</u> than without it!

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

10 TB Sorting Example

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

Sort 10 TB file with 1 GB of RAM

- I.e., File has 156.25 Million Disk Blocks, RAM: 15625 Pages
- I.e., N = 156.25 Million, B = 15624

⇒ Log_{15624} (N/[2(B+1)]) ~= Log_{15625} (5000) = 0.88

⇒ Sort cost = 2N (ceil[0.88]+1) = 4*N IOs

That's AMAZING!!!

Algorithm sorts BIG files (10,000x bigger than RAM) with a small constant factor (4x) on data size

IO analysis

Simplification

Sorting

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

Sort N pages with B+1 buffer size

(vs n log n, for n tuples in RAM. Negligible for large data, vs IO -- much, much slower)

~ 2 N

Sort N pages when N ~= B

(because (Log_B 0.5) < 0)

~ 4 N

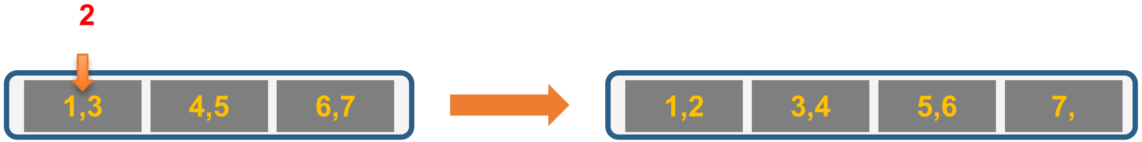
Sort N pages when N ~= 2*B^2

(because (Log_B B) = 1)

We assume cost = 1 IO for read and 1 IO for write.
Alternative IO model (e.g, SSDs in HW#2): 1 IO for read and 8 IOs for write?

Sorting, with insertions?

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

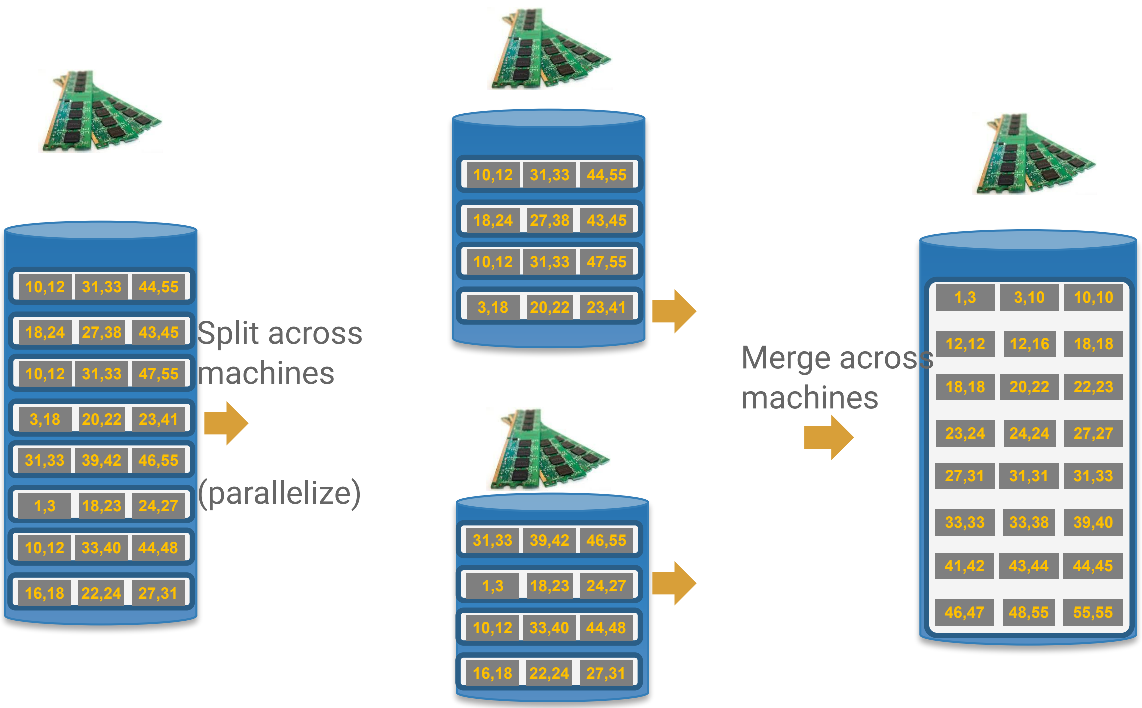


- We would have to potentially shift ***N*** records, requiring **~ 2*N/P** IO (worst case) operations (where P = # of records per page)!
  - We could leave some “slack” in the pages…

Could we get faster insertions?
(next section)

38

Scaling, Speeding Sort (in Cluster)



MergeSort locally in each machine (in parallel)

Notes

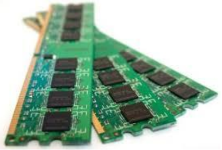
- Use N machines (N >= 2)
- Could reuse machines
- Speedup at cost of network bandwidth (especially with current data centers)

39

Big Scale Lego Blocks

Roadmap

Primary data structures/algorithms

	Hashing	Sorting
	HashTables ($hash_i$(key) --> location)	BucketSort, QuickSort MergeSort
	HashFunctions ($hash_i$(key) --> location)	MergeSortedFiles MergeSort
	HashFunctions ($hash_i$(key) --> location)	MergeSort

Let’s build Indexes

Example [Reminde]

CName_Index

Block #

CName	Block #
AAPL	
AAPL	
AAPL	
GOOG	
GOOG	
GOOG	
Alibaba	
Alibaba	

Block #

Company

CName	Date	Price	Country
AAPL	Oct1	101.23	USA
AAPL	Oct2	102.25	USA
AAPL	Oct3	101.6	USA
GOOG	Oct1	201.8	USA
GOOG	Oct2	201.61	USA
GOOG	Oct3	202.13	USA
Alibaba	Oct1	407.45	China
Alibaba	Oct2	400.23	China

PriceDate_Index

Date	Price	Block #
Oct1	101.23	
Oct2	102.25	
Oct3	101.6	
Oct1	201.8	
Oct2	201.61	
Oct3	202.13	
Oct1	407.45	
Oct2	400.23	

How?

1. Index contains search values + Block #: e.g., DB block number.
   - In general, "pointer" to where the record is stored (e.g., RAM page, DB block number or even machine + DB block)
   - Index is conceptually a table. In practice, implemented very efficiently (see how soon)

2. Can have multiple indexes to support multiple search keys

How to build?

1. How is data organized?
    - Is data in Row or Column store?
    - Is data sorted or not?

2. How do we organize search values?

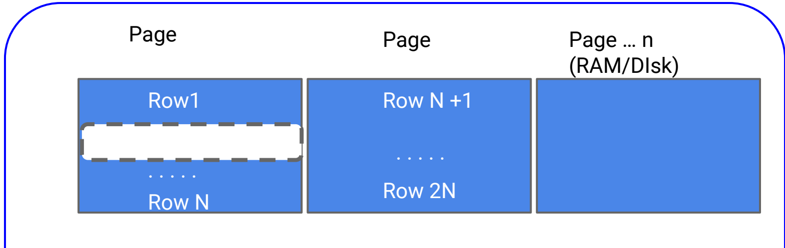
Recall Data Layout

Company(CName, StockPrice, Date, Country)

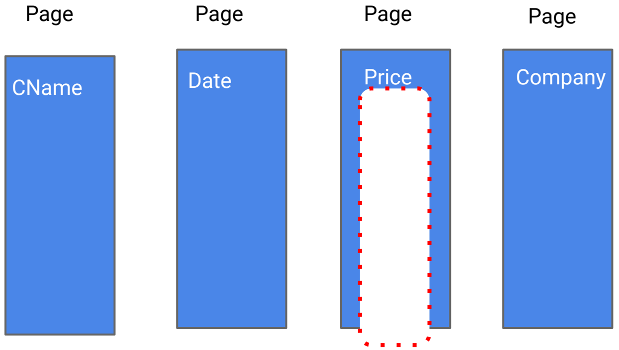
Logical Table

Col3

	Company			
	CName	Date	Price	Country
Row1	AAPL	Oct1	101.23	USA
	AAPL	Oct2	102.25	USA
Row3	AAPL	Oct3	101.6	USA
	GOOG	Oct1	201.8	USA
Row5	GOOG	Oct2	201.61	USA
	GOOG	Oct3	202.13	USA
	Alibaba	Oct1	407.45	China
Row8	Alibaba	Oct2	400.23	China



Row based storage (aka Row Store)

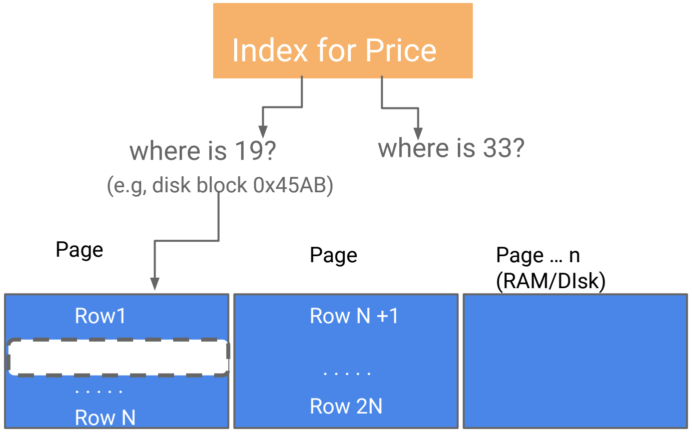


Column based storage (aka Column Store)

Index on row store

Query: Search for cname with specific price?

⇒ If ‘price’ is an indexed column, query will be fast.
⇒ ‘Price’ is search key. Values in price column are search values.



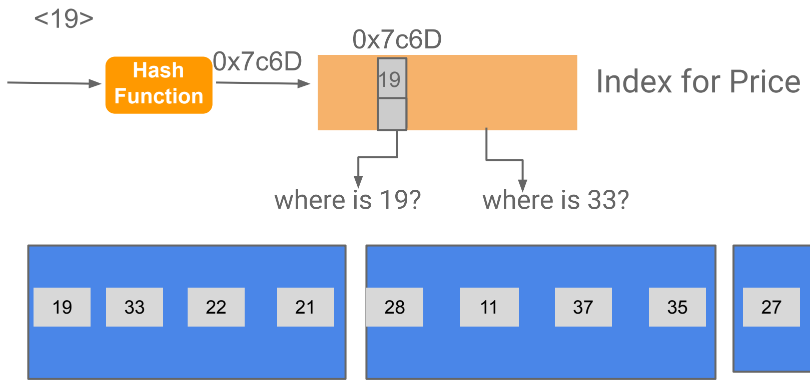
“Real” data layout, with full records
(including cname, prices, etc.)

Our 1st Hashing index

Goal of Index: where is location of each value?

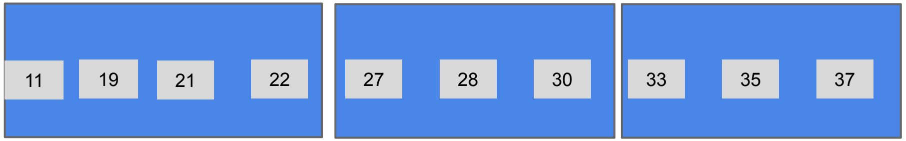
For simplicity, we'll only show the search values for Price index

Example row: <'goog', price=19, date=Oct 1, ...> as <price=19> or <19>



Maintain locations of N values

Sorted



If sorted, will need to maintain locations only of smallest value in each block.

How it works in practice?

1. Schema designer picks a column to keep data sorted by (e.g., price). Index for that column is cheap.

2. For other columns, index will be bigger (e.g., CName)

Index Types

- Hash Tables
  - IO-aware hashing (e.g., *linear* or *extendible hashing)*

These data structures are “IO aware”

- B-Trees *(covered next)*
  - Very good for range queries, sorted data
  - Some old databases only implemented B-Trees
  - *We will look at a variant called B+ Trees*

Real difference between structures:
costs of ops *determines which index you pick and why*