Lecture 13: Indexing, Sorting

Announcements

Nov 13
DJ Patil on data ethics,

ex-Chief Data Scientist in Obama's Whitehouse

Nov 15
Theo Vassilakis on big GPS data,

Group CTO at Grab (ridesharing behemoth in Asia, ~6.6B$ in funding)

ex-Google, Dremel/BigQuery

Note: Live audience form & points

Recap

Table vs File vs Pages

Buffer, Buffer Manager

Read, Flush, Discard Page

Table: PlayingCards(Number, Suit, …)

2	hearts	...
3	clubs	...
...		
...		

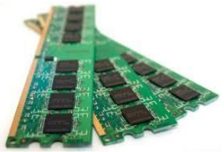
FIle: f = fopen(“cards.db”, “r”)

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")***

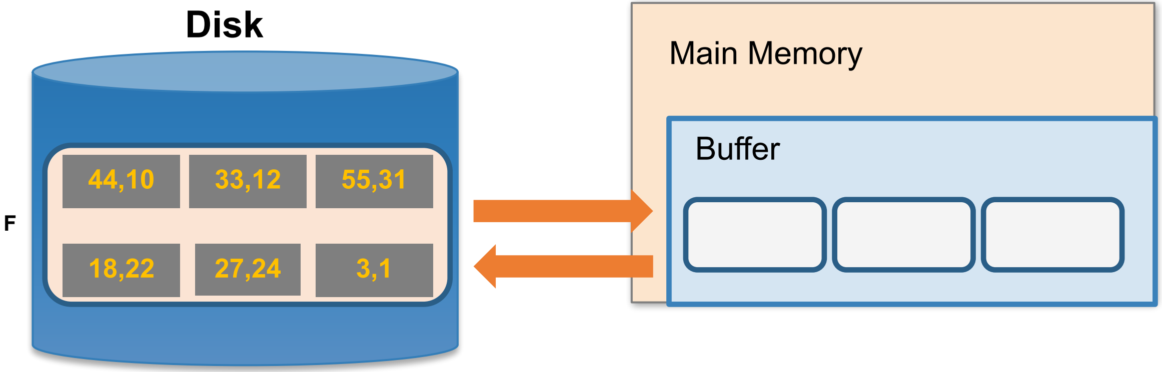
2. **Merge** pairs (or groups) of runs with ***external merge algorithm***

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

External Merge Sort Algorithm

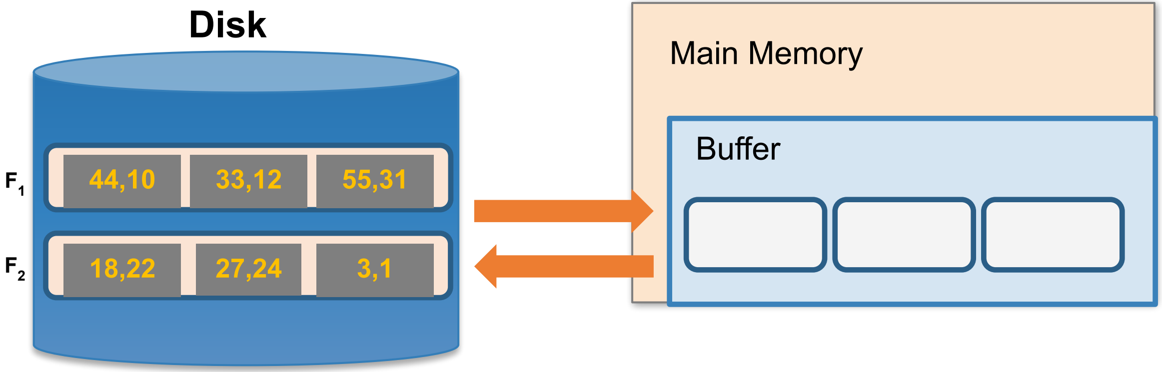
Example
- 3 Buffer pages
- 6-page file

Orange file
= unsorted



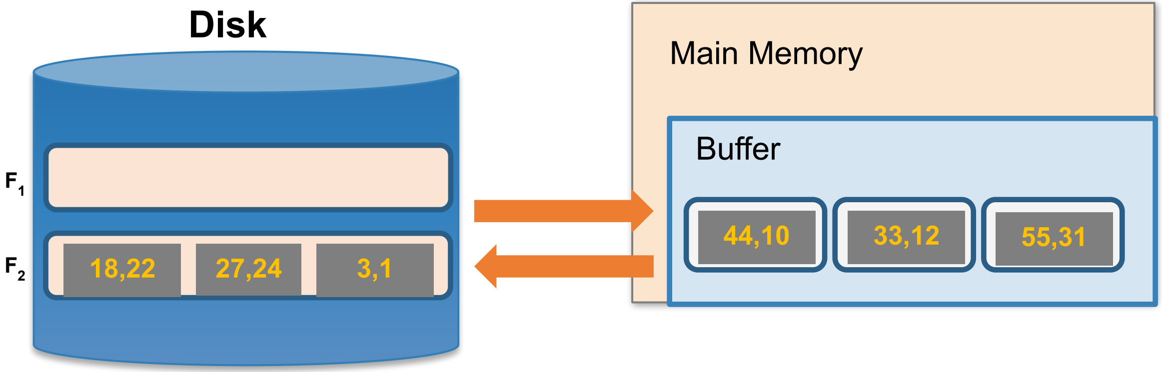
1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm



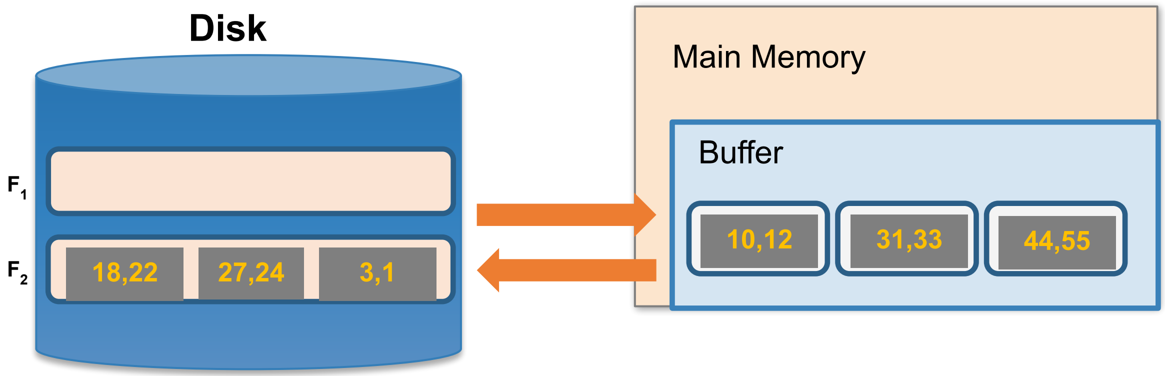
1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm



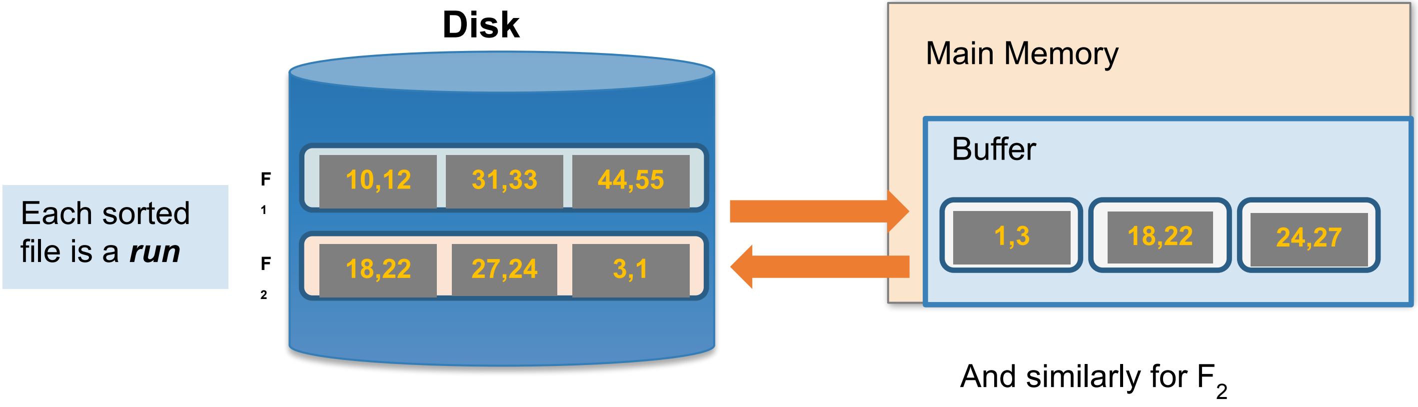
1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm



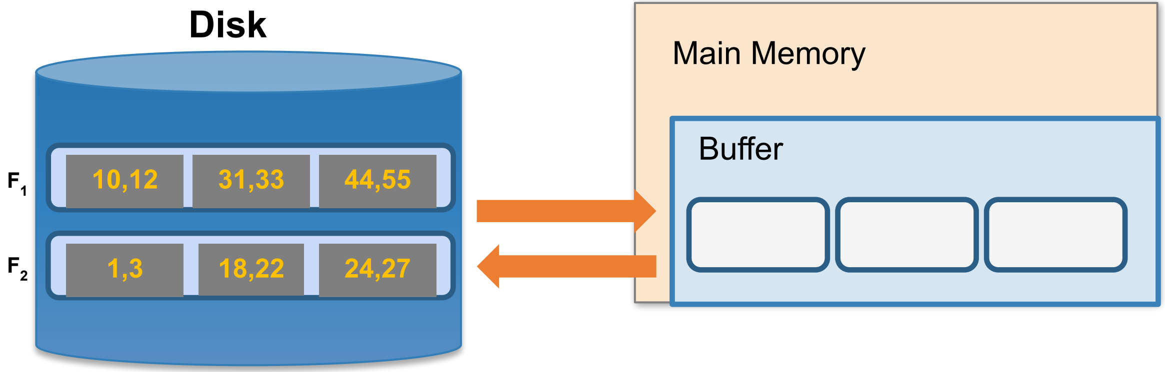
1. Split into chunks small enough to **sort in memory**

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

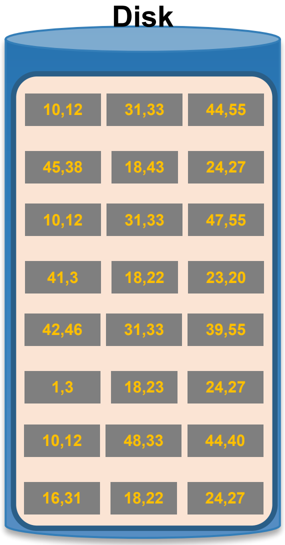
For 3 buffer pages, 6 page file:

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

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

3. Total cost = 24 IO

Running External Merge Sort on Larger Files

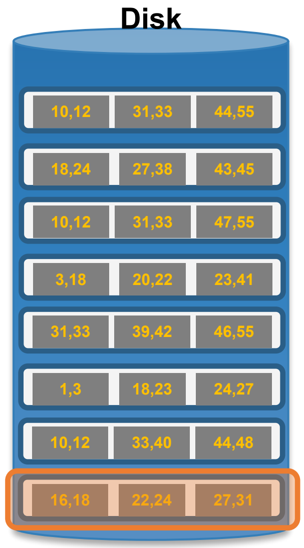


Assume we still only have *3* buffer pages *(Buffer not pictured)*

Running External Merge Sort on Larger Files

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

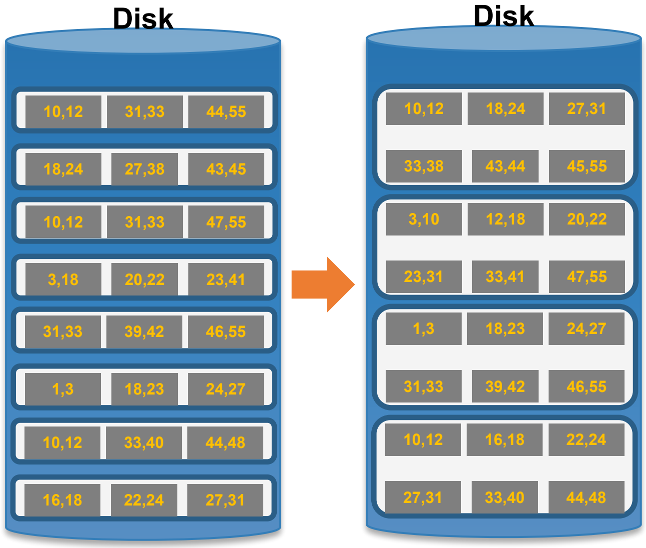
Running External Merge Sort on Larger Files



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

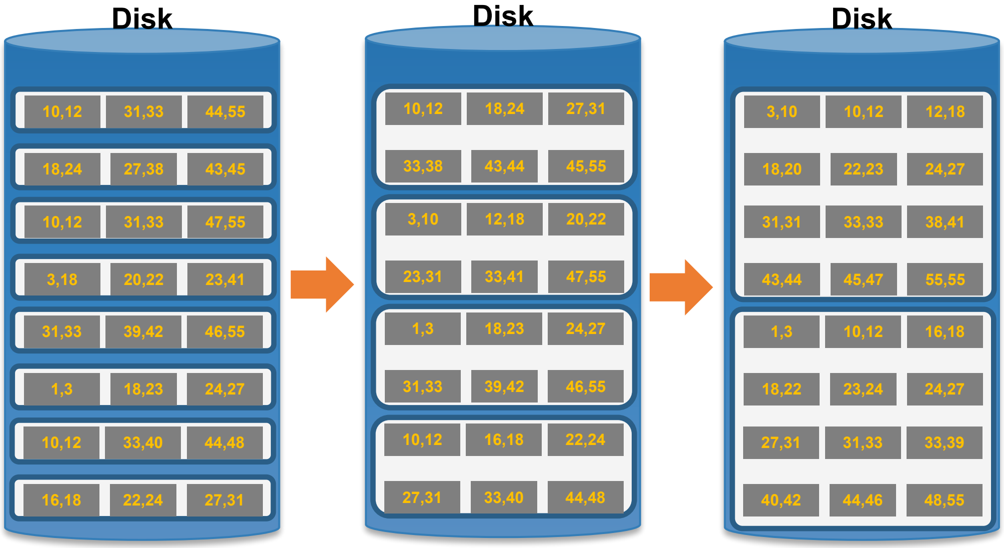
Each sorted file is a ***run***

Running External Merge Sort on Larger Files



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

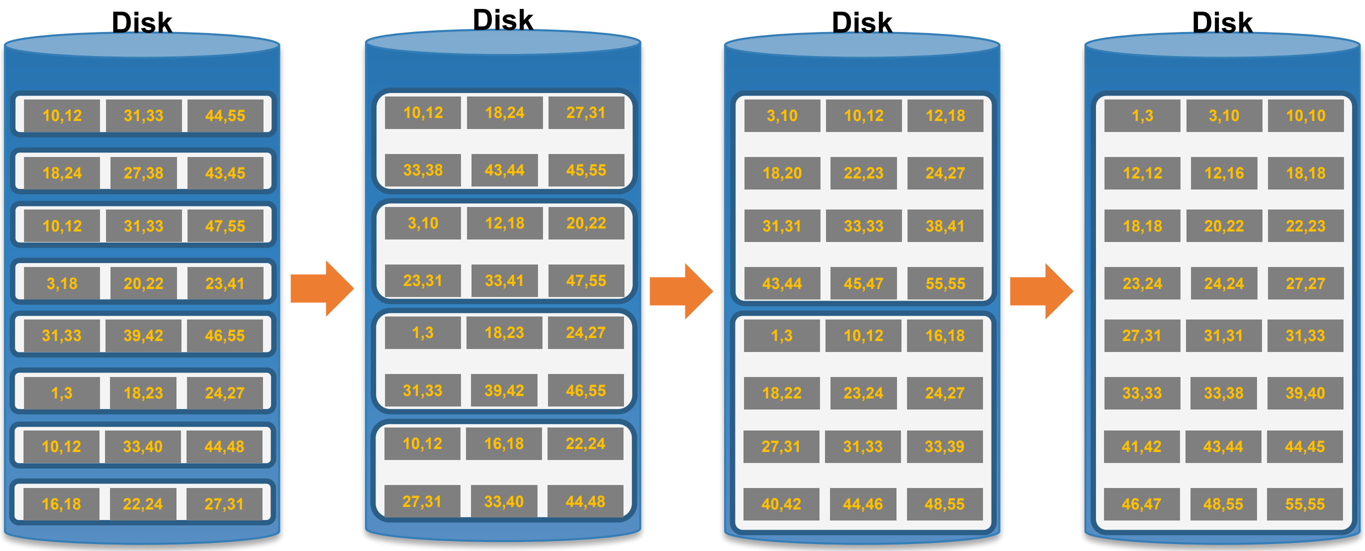
Running External Merge Sort on Larger Files



3. And repeat…

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

Running External Merge Sort on Larger Files

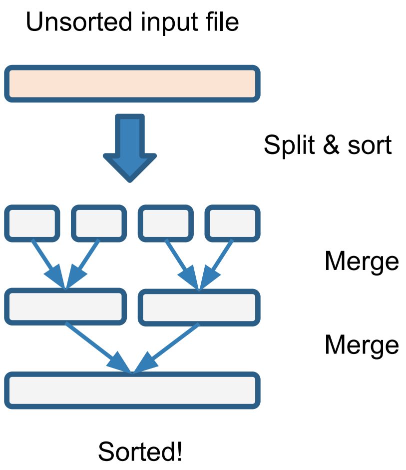


4. And repeat!

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>

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

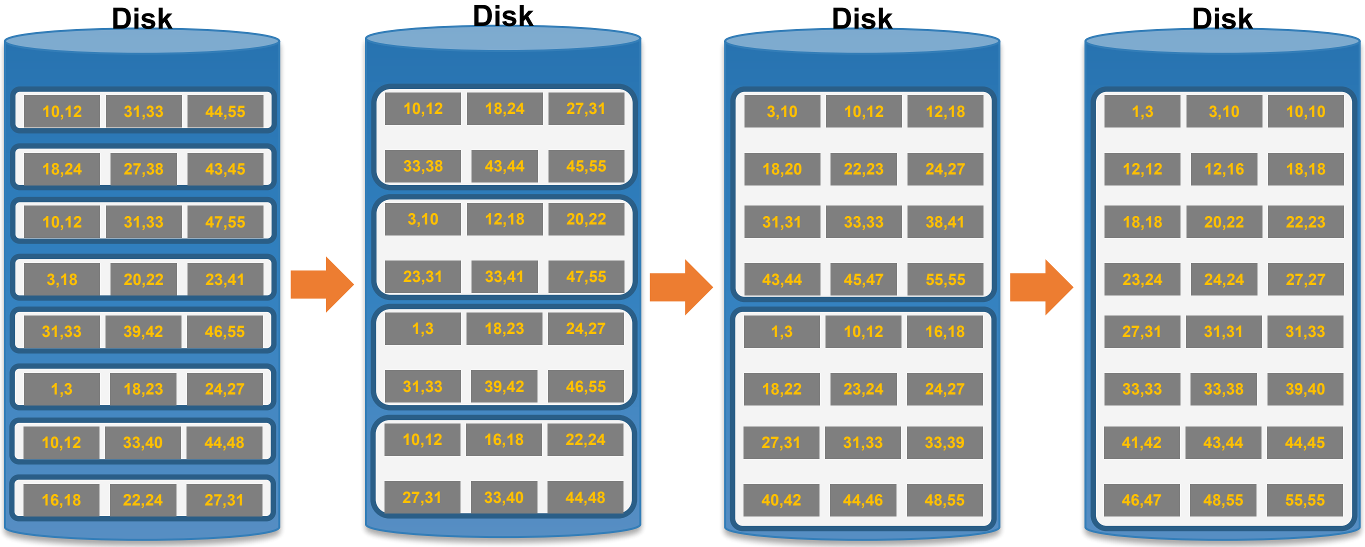
Starting with runs of length 1

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

Performing ***B***-way merges

Pretty fast IO aware sort !!

Repacking for longer runs (Optimization)

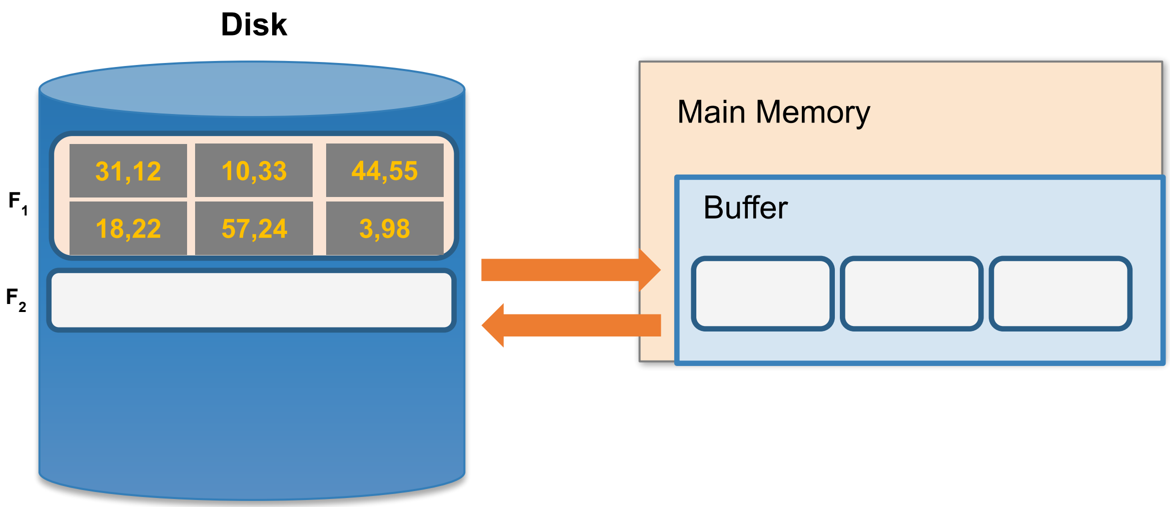


Idea: What if it's already 'partly' sorted?

Can we be smarter with buffer?

Repacking Example: 3 page buffer

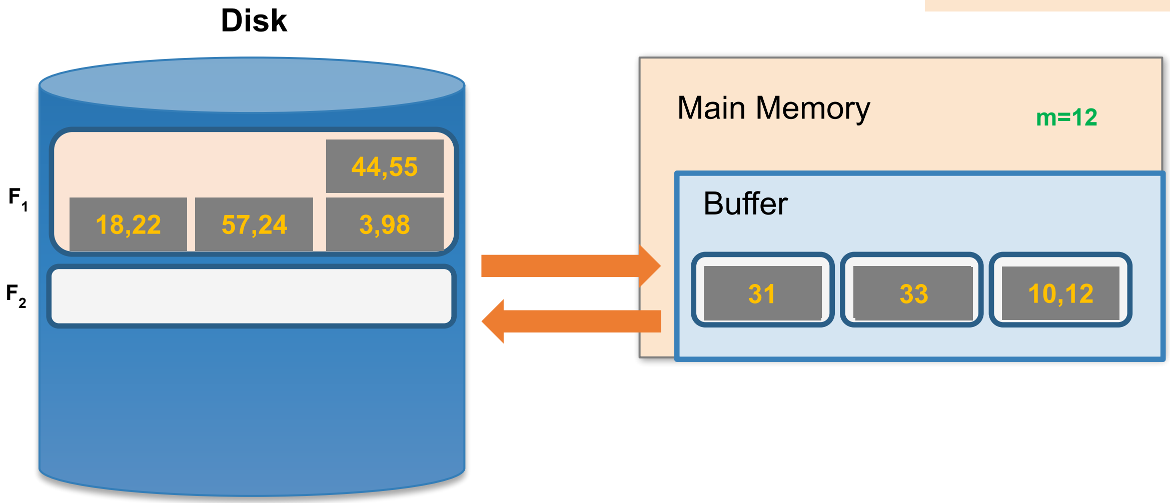
- Start with unsorted single input file, and load 2 pages



Repacking Example: 3 page buffer

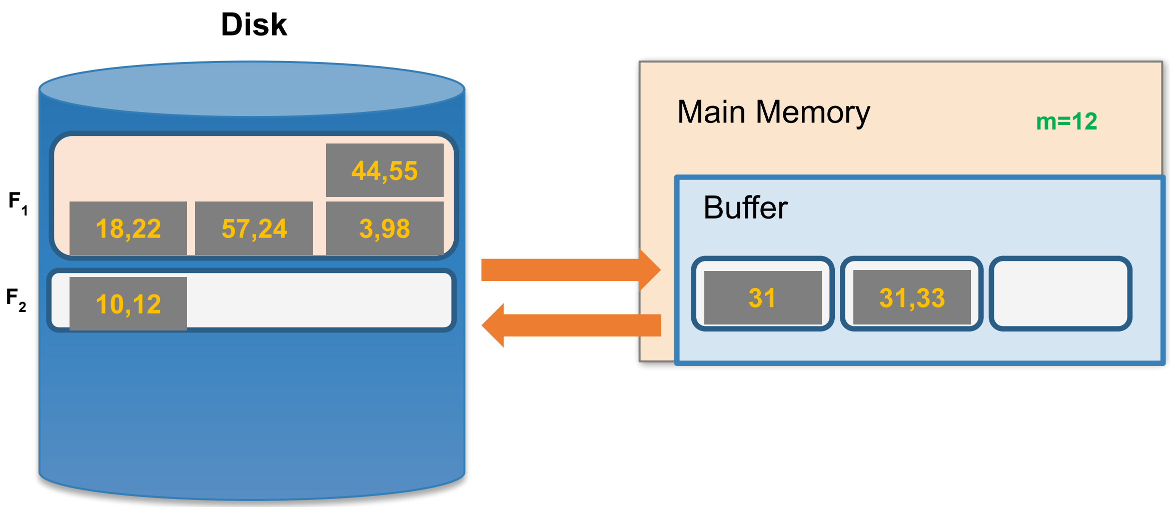
- Take the minimum two values, and put in output page

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



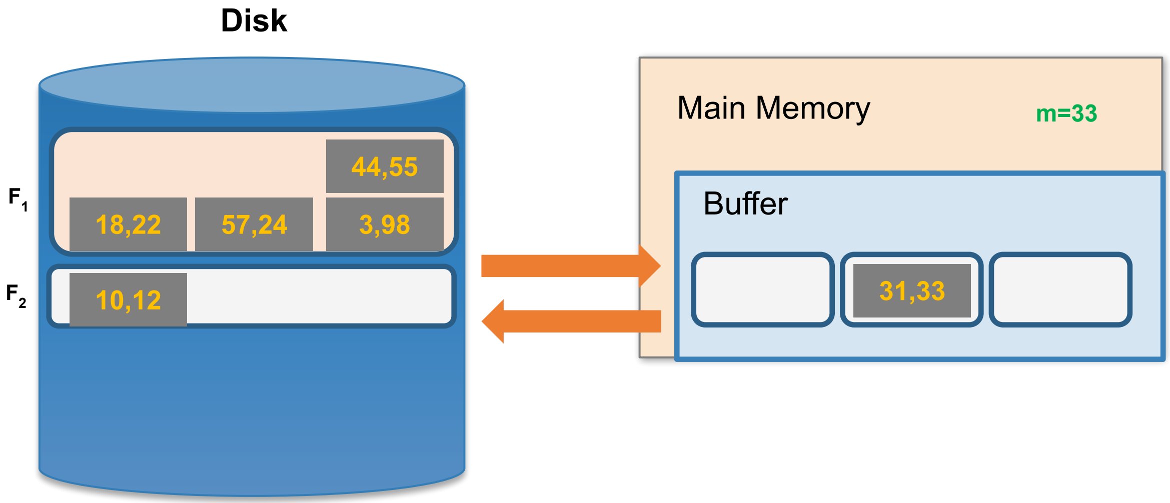
Repacking Example: 3 page buffer

- Next, ***repack***



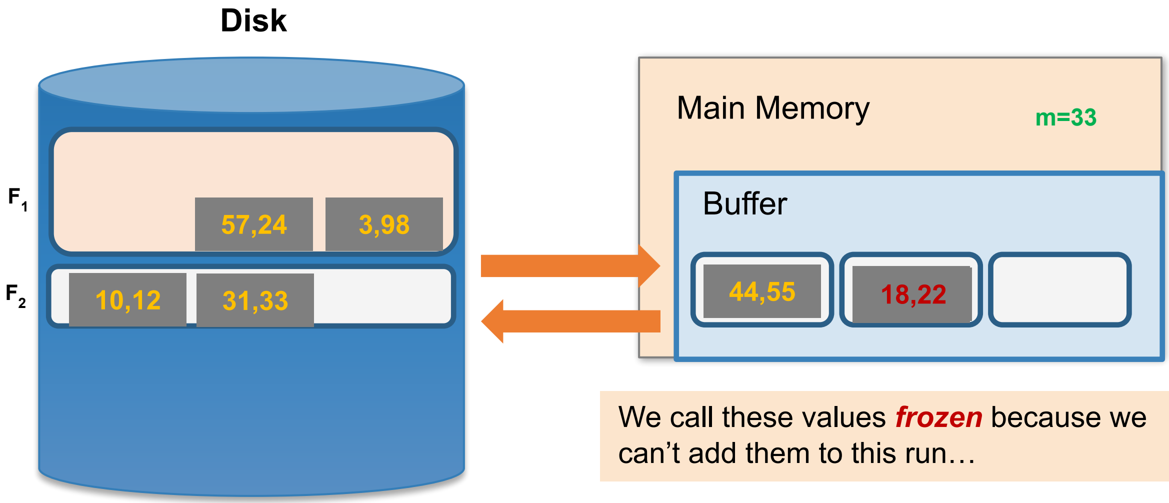
Repacking Example: 3 page buffer

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



Repacking Example: 3 page buffer

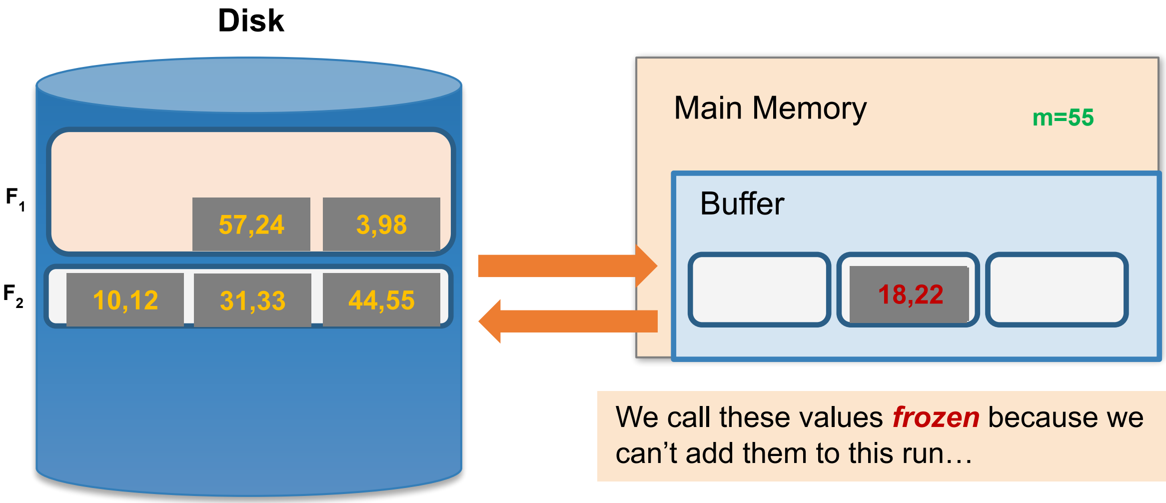
- Now, however, ***the smallest values are less than the largest (last) in the sorted run…***



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

Repacking Example: 3 page buffer

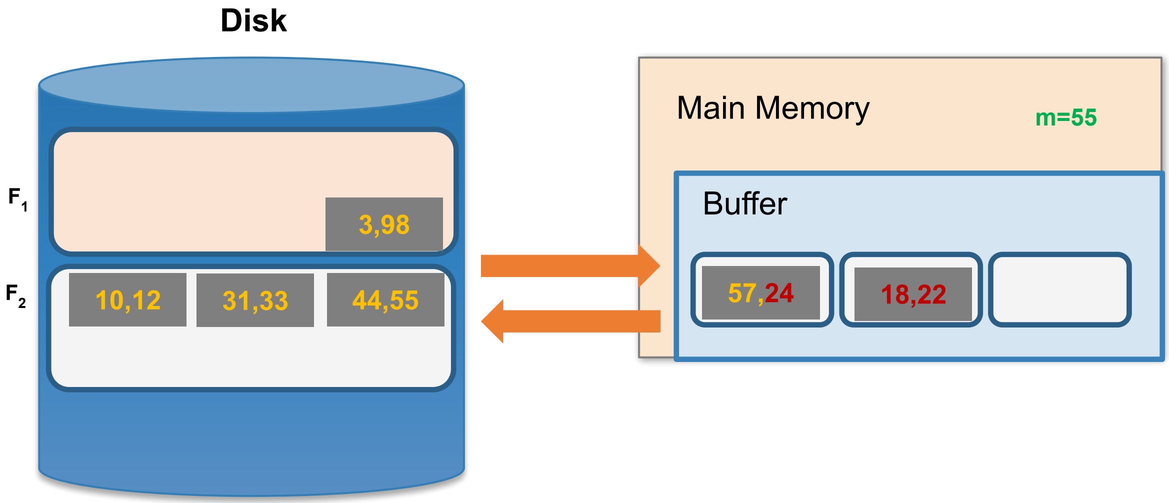
- Now, however, ***the smallest values are less than the largest (last) in the sorted run…***



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

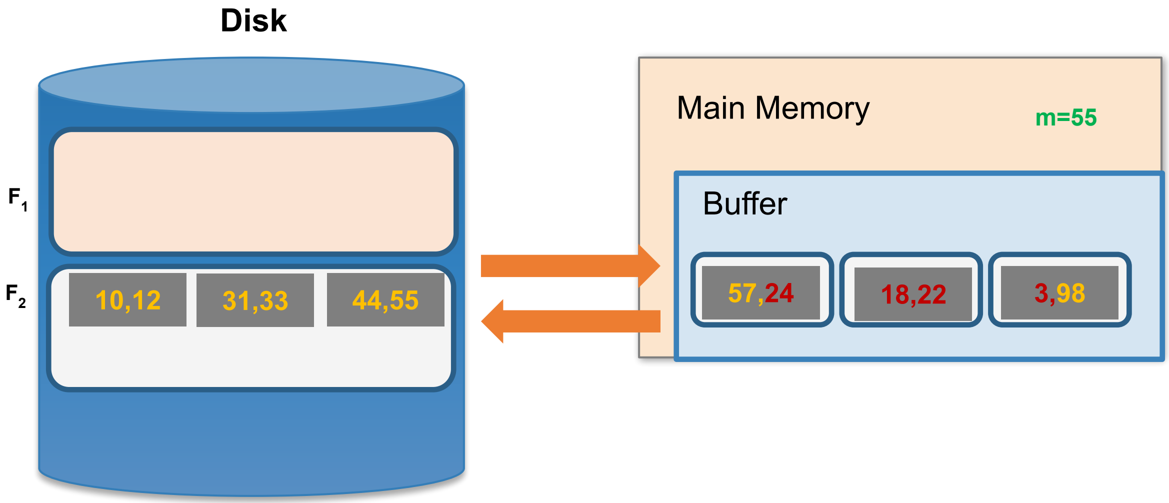
Repacking Example: 3 page buffer

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



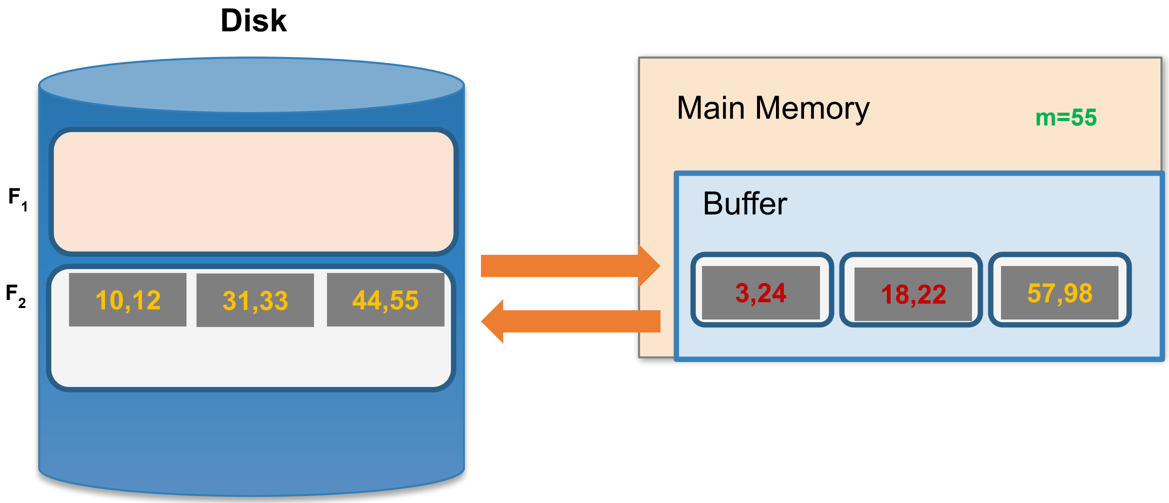
Repacking Example: 3 page buffer

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



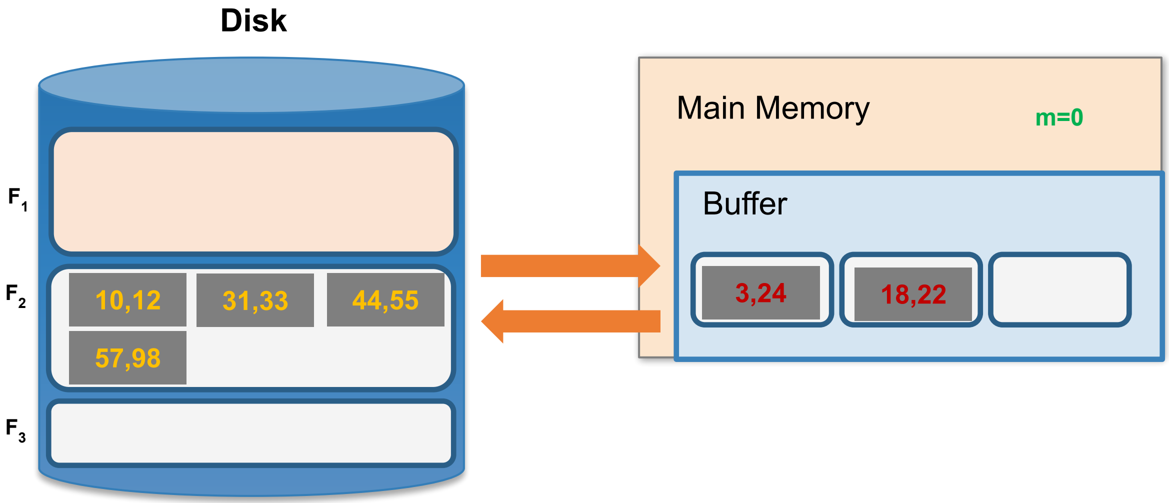
Repacking Example: 3 page buffer

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



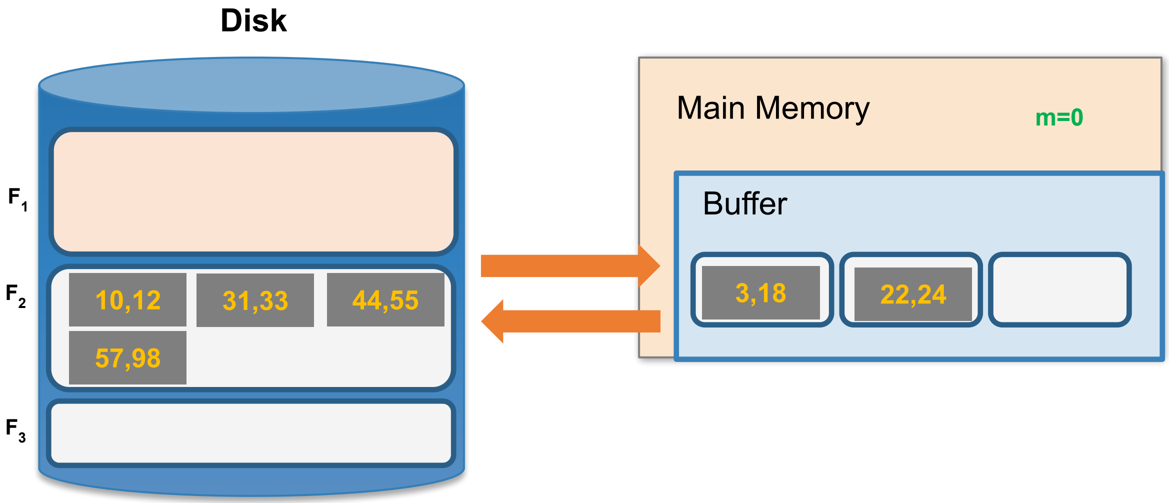
Repacking Example: 3 page buffer

- 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**

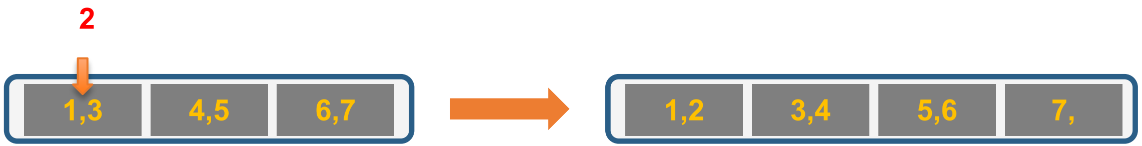
- In general, with repacking we do <u>**no worse**</u> than without it!

- 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?



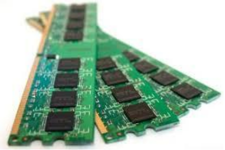
- We would have to potentially shift ***N*** records, requiring up to **~ 2*N/P** IO operations (where P = # of records per page)!
    - We could leave some “slack” in the pages…

Could we get faster insertions?
(next section)

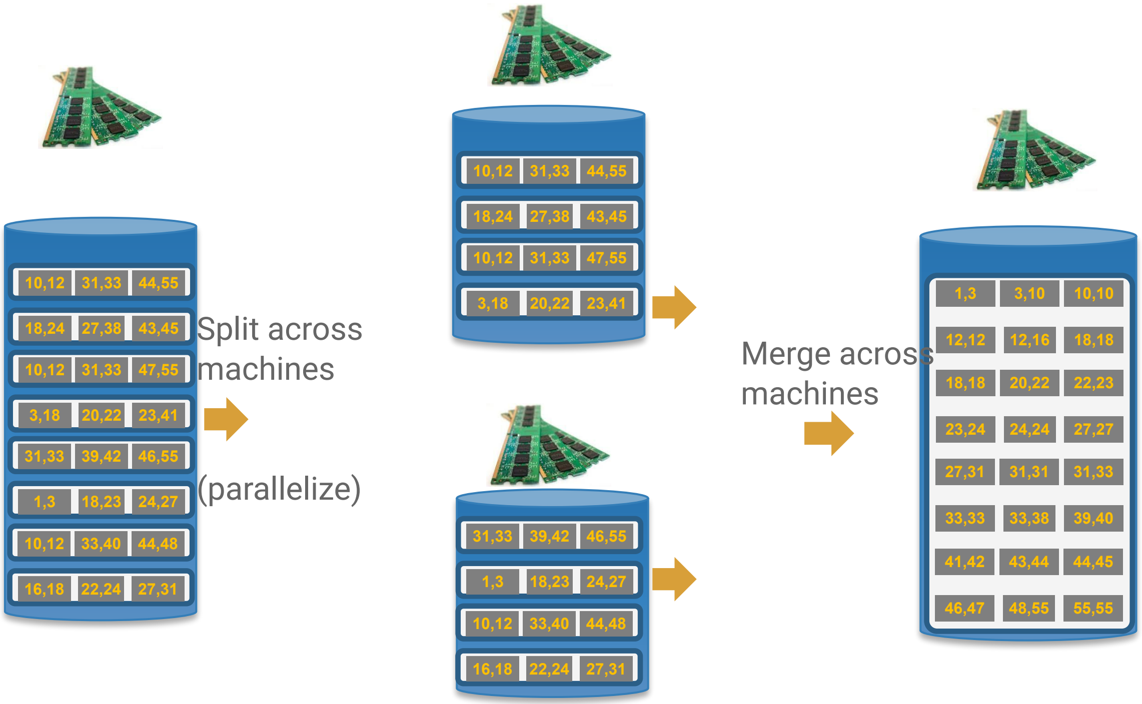
Big Scaling (with Indexes)

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

B+ Trees:
An IO-Aware Index Structure

What you will learn about in this section

1. B+ Trees: Basics
2. B+ Trees: Design & Cost
3. Clustered Indexes

Building our 1st index

Person(<u>name</u>, age)

Query: Search for people of specific age

<u>Design idea #1:</u>

- Sort records by age…(fast)
- How many IO operations to search over N sorted records?
  - Simple scan: O(N)
  - Binary search: $O(\log_2 N)$

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

Index Types

- 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***
- Hash Tables
  - There are variants of this basic structure to deal with IO
  - Called ***linear*** or ***extendible hashing-*** IO aware!

These data structures are “IO aware”

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

B+ Trees

- Search trees
  - B does not mean binary!
- Idea in B Trees:
  - make 1 node = 1 physical page
  - Balanced, height adjusted tree (not the B either)
- Idea in B+ Trees:
  - Make leaves into a linked list (for range queries)

B+ Tree Basics

Person(name, age)

Example:
Sorted data

Name: Joe Age: 11	Name: Jake Age: 15	Name: John Age: 21	Name: Bess Age: 22	Name: Bob Age: 27	Name: Sally Age: 28	Name: Sal Age: 30	Name: Sue Age: 33	Name: Jess Age: 35	Name: Alf Age: 37

For simplicity

11	15	21	22	27	28	30	33	35	37

B+ Tree Basics

10	20	30

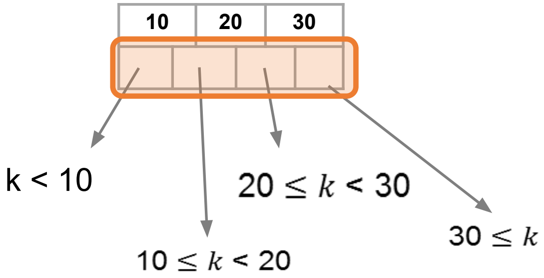
Parameter ***d*** = the degree

Each *non-leaf ("interior")* ***node*** has $\geq$ d and $\leq$ 2d ***keys****

except for root node, which can have between* ***1* *and 2d keys*

11 15 21 22 27 28 30 33 35 37

B+ Tree Basics

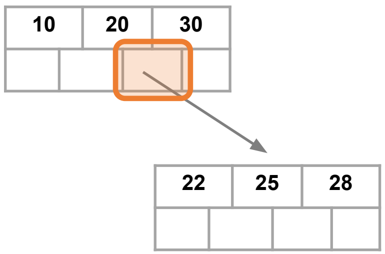


The *n* keys in a node define *n+1* ranges

11 15 21 22 27 28 30 33 35 37

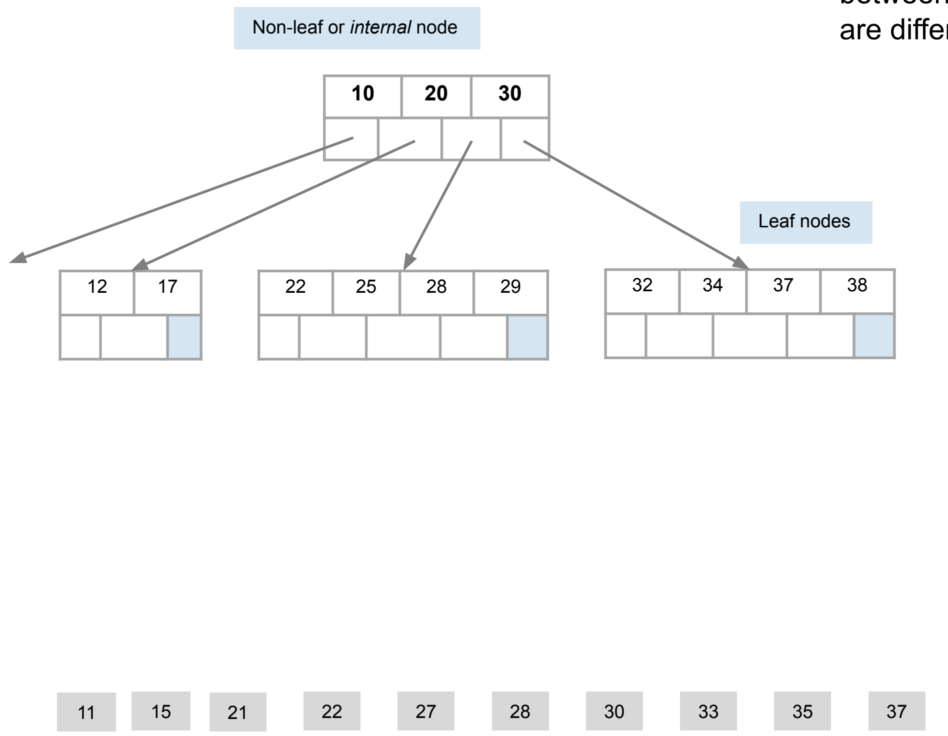
B+ Tree Basics

Non-leaf or *internal* node



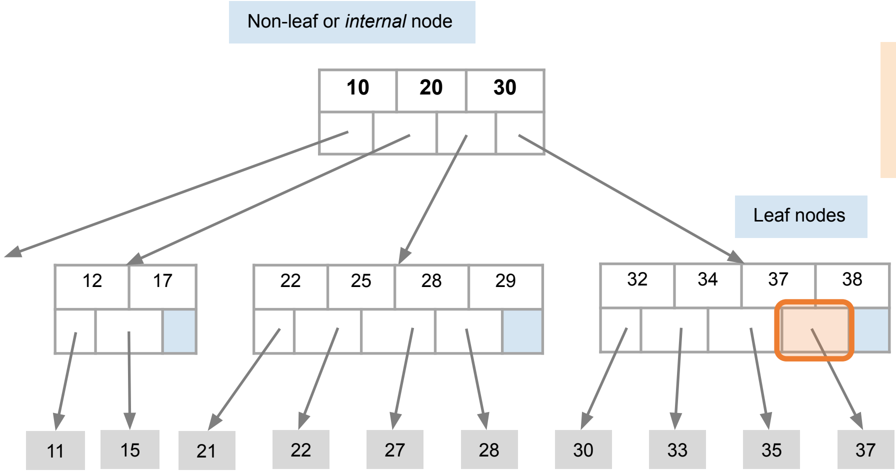
For each range, in a *non-leaf* node, there is a **pointer** to another node with keys in that range

B+ Tree Basics



Leaf nodes also have between *d* and *2d* keys, and are different in that:

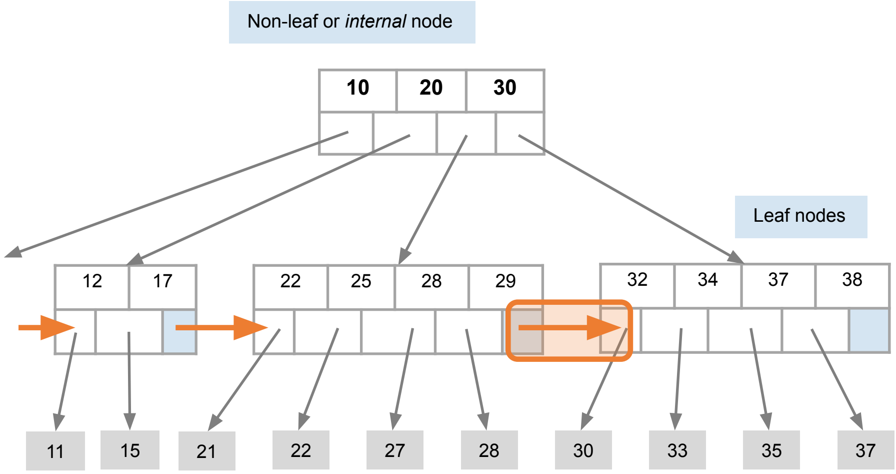
B+ Tree Basics



Leaf nodes also have between *d* and *2d* keys, and are different in that:

Their key slots contain pointers to data records

B+ Tree Basics

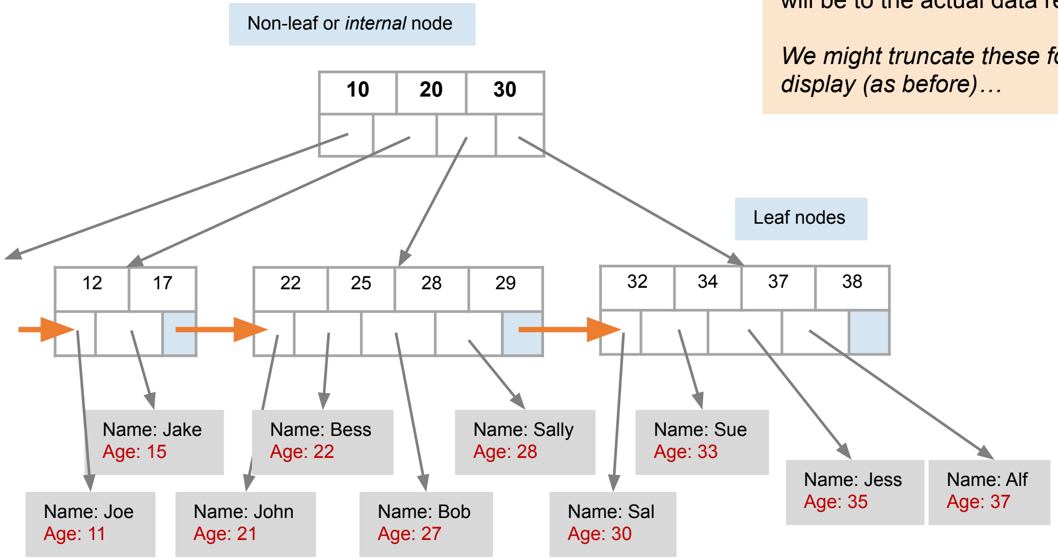


Leaf nodes also have between *d* and *2d* keys, and are different in that:

Their key slots contain pointers to data records

They contain a pointer to the next leaf node as well, ***for faster sequential traversal***

B+ Tree Basics



Note that the pointers at the leaf level will be to the actual data records (rows).

We might truncate these for simpler display (as before)…

Some finer points of B+ Trees

Searching a B+ Tree

- For exact key values:
  - Start at the root
  - Proceed down, to the leaf

```
SELECT name
FROM   people
WHERE  age = 25
```

- For range queries:
  - As above
  - *Then sequential traversal*

```
SELECT name
FROM   people
WHERE  20 <= age
  AND  age <= 30
```

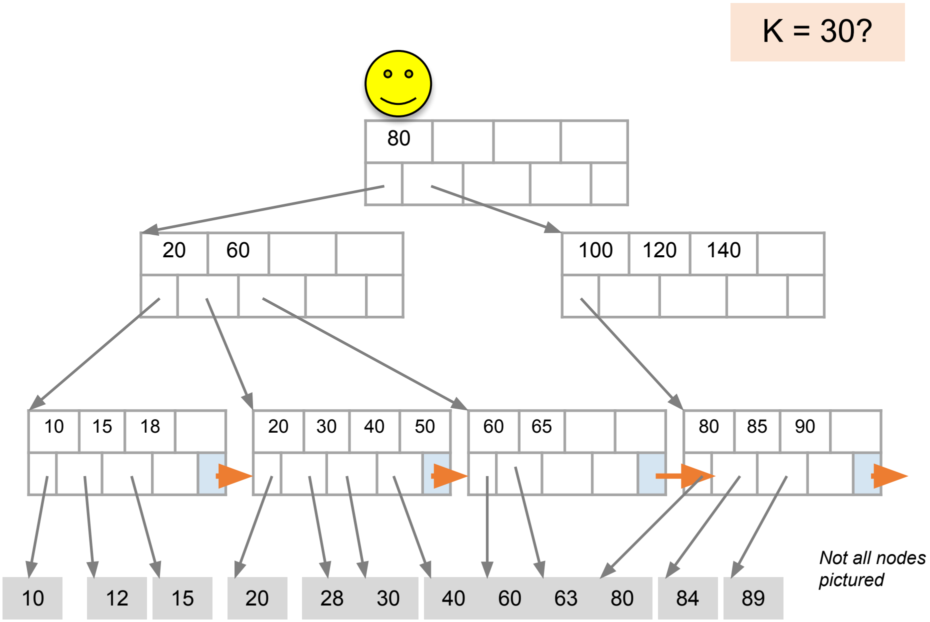
B+ Tree Exact Search Animation

30 < 80

30 in [20,60)

30 in [30,40)

To the data!



B+ Tree Range Search Animation

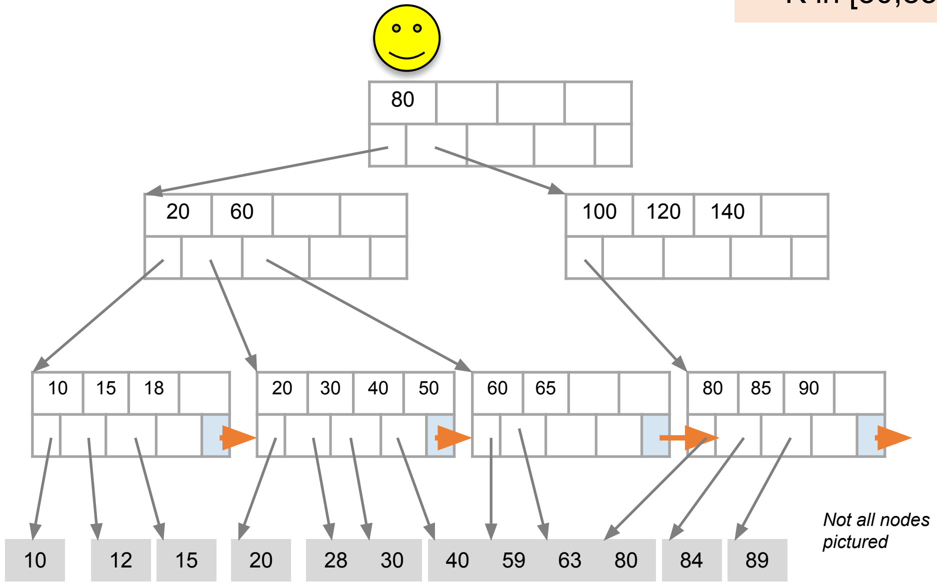
K in [30,85]?

30 < 80

30 in [20,60)

30 in [30,40)

To the data!



B+ Tree Design

- How large is ***d***?

- Example:
  - Key size = 4 bytes
  - Pointer size = 8 bytes
  - Block size = 64k bytes

- We want each *node* to fit on a single *block/page*

$$\underbrace{2d \times 4}_{\text{(keys)}} + \underbrace{(2d+1) \times 8}_{\text{(pointers)}} \le 64k \rightarrow \boldsymbol{d \sim= 2730}$$

B+ Tree: High Fanout = Smaller & Lower IO

- As compared to e.g. binary search trees, B+ Trees have **high *fanout*** (***between d+1 and 2d+1***)

- Hence the **depth of the tree is small** → getting to any element requires very few IO operations!
  - Also can often store most/all of B+ Tree in RAM!

- A TiB = 2^{40} Bytes. What is the height of a B+ Tree (with fill-factor = 1) that indexes it (with 64K pages)?
  - $(2*2730 + 1)^h = 2^{40}$ → ***h = 4***

The **fanout** is defined as the number of pointers to child nodes coming out of a node

Note that fanout is dynamic- we'll often assume it's constant just to come up with approximate eqns!

B+ Trees in Practice

- Typical order: d=100. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities:
  - Height 4: 133^4 = 312,900,700 records
  - Height 3: 133^3 = 2,352,637 records
- Top levels of tree sit *in the buffer pool*:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes

<u>Fill-factor</u> is the percent of available slots in the B+ Tree that are filled; is usually < 1 to leave slack for (quicker) insertions

Typically, only pay for one IO!

Simple Cost Model for Search

- Let:
  - ***f*** = fanout, which is **in [d+1, 2d+1]** ***(we'll assume it's constant for our cost model…)***
  - ***N*** = the total number of *pages* we need to index
  - ***F*** = fill-factor (usually ~= 2/3)
- Our B+ Tree needs to have room to index ***N / F*** pages!
  - We have the fill factor in order to leave some open slots for faster insertions
- What height (*h*) does our B+ Tree need to be?
  - h=1 → Just the root node- room to index f pages
  - h=2 → f leaf nodes- room to index f^2 pages
  - h=3 → f^2 leaf nodes- room to index f^3 pages
  - …
  - h → f^{h-1} leaf nodes- room to index f^h pages!

→ We need a B+ tree of height h = $\left\lceil \log_f \frac{N}{F} \right\rceil$

Simple Cost Model for Search

- Note that if we have ***B*** available buffer pages, by the same logic:
  - We can store $\boldsymbol{L_B}$ levels of the B+ Tree in memory
  - where $\boldsymbol{L_B}$ ***is the number of levels such that the sum of all the levels' nodes fit in the buffer:***
    - $B \geq 1 + f + \cdots + f^{L_B - 1} = \sum_{l=0}^{L_B - 1} f^l$

- In summary: to do exact search:
  - We read in one page per level of the tree
  - However, levels that we can fit in buffer are free!
  - Finally we read in the actual record

IO Cost: $\left\lceil \log_f \frac{N}{F} \right\rceil - L_B + 1$

where $B \geq \sum_{l=0}^{L_B - 1} f^l$

Simple Cost Model for Search

- To do range search, we just follow the horizontal pointers
- The IO cost is that of loading additional leaf nodes we need to access + the IO cost of loading each ***page*** of the results- we phrase this as “Cost(OUT)”

IO Cost: $\left\lceil \log_f \frac{N}{F} \right\rceil - L_B + Cost(OUT)$

where $B \geq \sum_{l=0}^{L_B - 1} f^l$

Fast Insertions & Self-Balancing

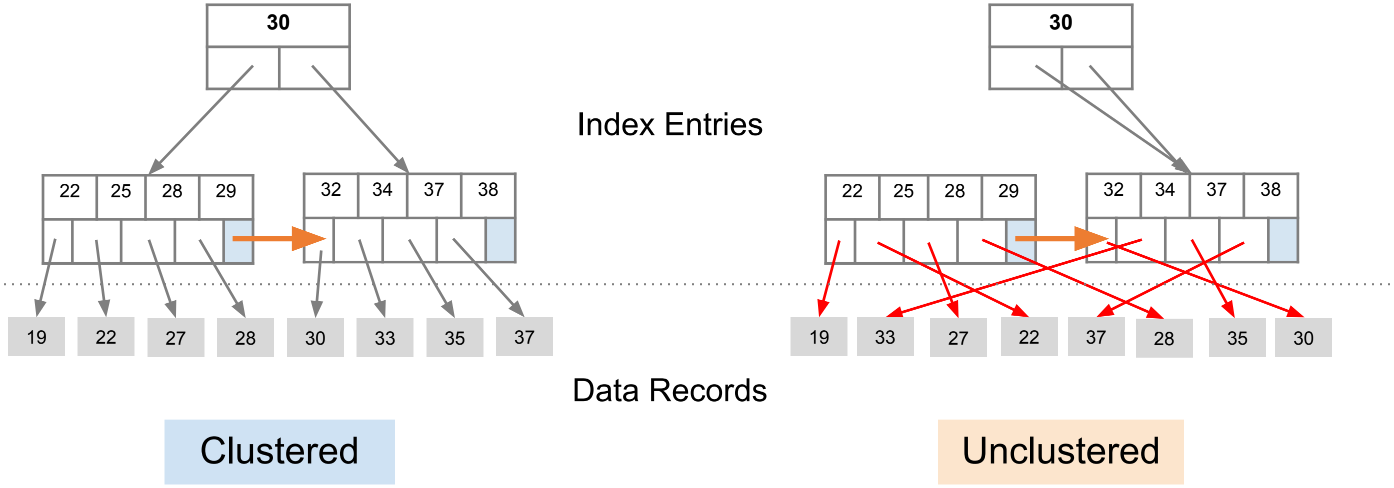
- We won’t go into specifics of B+ Tree insertion algorithm, but has several attractive qualities:
  - **~ Same cost as exact search**
  - ***Self-balancing:*** B+ Tree remains **balanced** (with respect to height) even after insert

B+ Trees also (relatively) fast for single insertions!
However, can become bottleneck if many insertions (if fill-factor slack is used up…)

Clustered Indexes

An index is ***clustered*** if the underlying data is ordered in the same way as the index's data entries.

Clustered vs. Unclustered Index



Clustered vs. Unclustered Index

- Recall that for a disk with block access, **sequential IO is much faster than random IO**
- For exact search, no difference between clustered / unclustered
- For range search over R values: difference between **1 random IO + R sequential IO**, and **R random IO**:
  - A random IO costs ~ 10ms (sequential much much faster)
  - For R = 100,000 records- **difference between ~10ms and ~17min!**

Summary

- We covered an algorithm + some optimizations for sorting larger-than-memory files efficiently
  - An ***IO aware*** algorithm!

- We create **indexes** over tables in order to support ***fast (exact and range) search*** and ***insertion*** over ***multiple search keys***

- **B+ Trees** are one index data structure which support very fast exact and range search & insertion via ***high fanout***
  - ***Clustered vs. unclustered*** makes a big difference for range queries too

THANK YOU!