B+ Trees:
An IO-Aware Index Structure

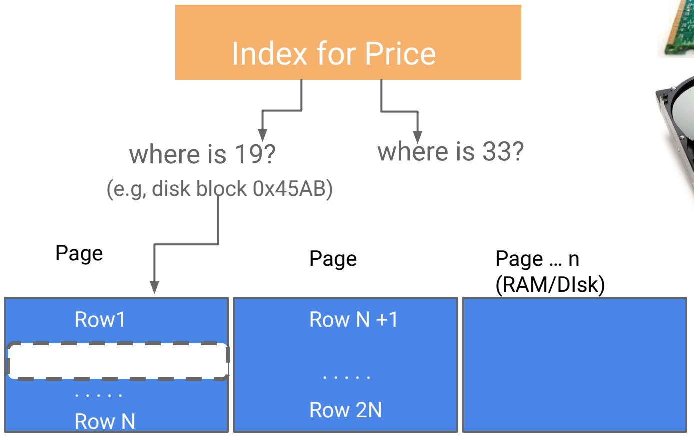
IO Aware

For larger-than-memory (big) files, we need efficient algorithms (and data structures) that work with non-memory IO systems

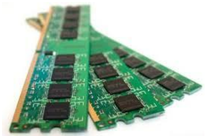
- An *IO aware* algorithm! (and data structures)

Index on row store [recall]

Query: Search for cname with specific price?



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



How do we store Index?
⇒ Idea: Index is just a table (rows/columns). Same ideas

- Store in pages
- Persist on disk
- Page into RAM buffer

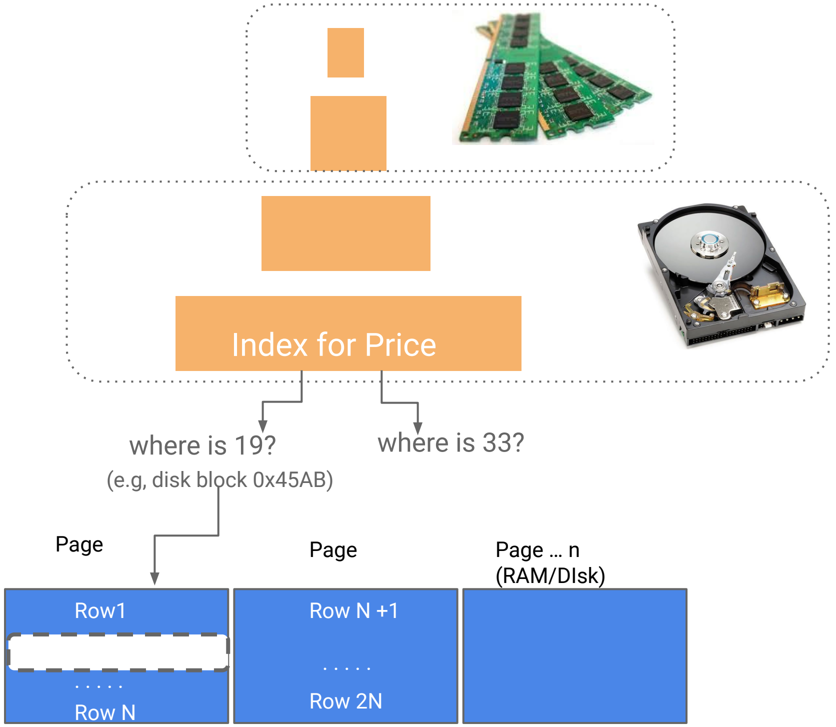
If Index fits in RAM?

- Lookups are fast

If Index does not fit in RAM?

- Could page at random
- Can we organize index pages better?

Hierarchical Indexes



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

How do we store Index?
⇒ Idea: Index is just a table (rows/columns)

Same ideas
- Store in pages
- Persist on disk
- Page into RAM buffer

+ Index the index :-)

Idea in B+ Trees

Search trees that are IO aware

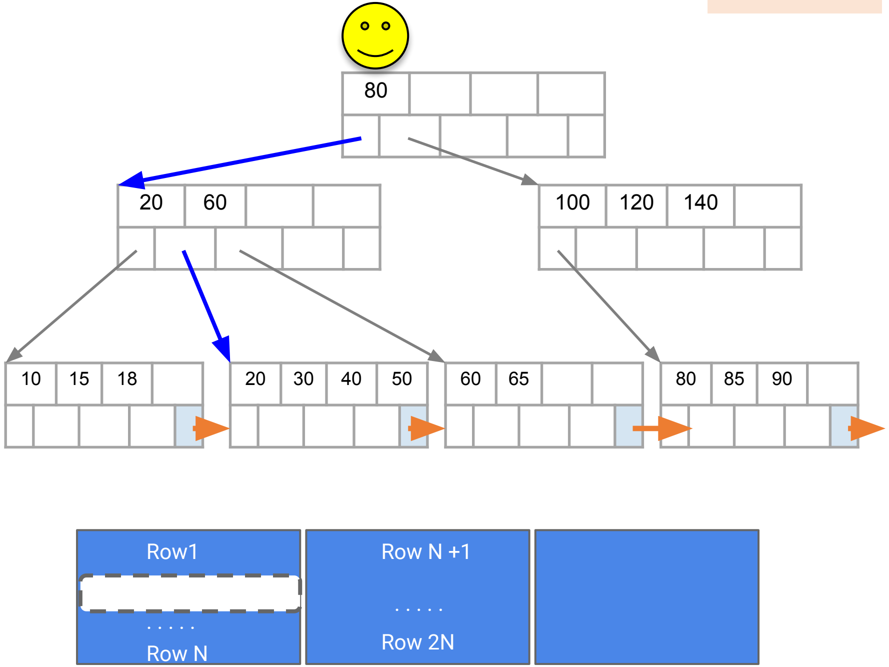
- make 1 node = 1 physical page
- Balanced, height adjusted tree
- Make leaves into a linked list (for range queries)

What you will learn about in this section

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

B+ Tree Exact Search

K = 30?



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

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

We truncate and only display search values for simplicity (as before)…

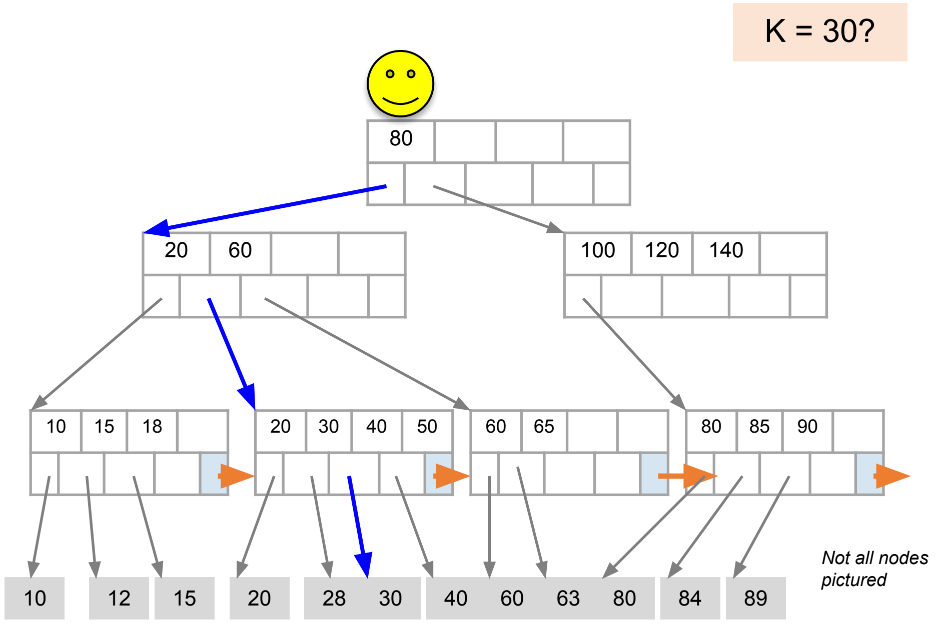
B+ Tree Exact Search

30 < 80

30 in [20,60)

30 in [30,40)

To the data!
[simplified]



Searching a B+ Tree

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

```
SELECT cname
FROM   Company
WHERE  price = 25
```

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

```
SELECT cname
FROM   Company
WHERE  20 <= price
  AND  price <= 30
```

B+ Tree Range Search

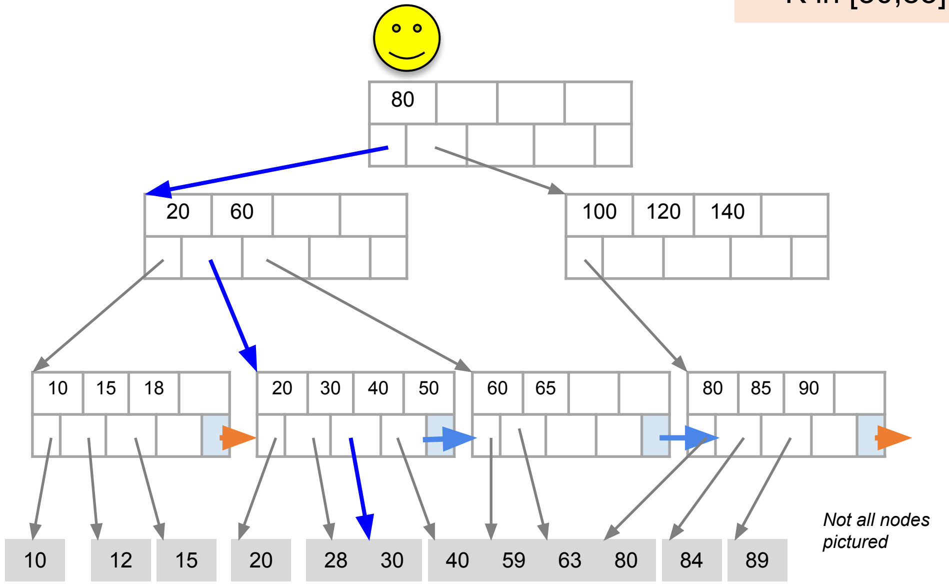
K in [30,85]?

30 < 80

30 in [20,60)

30 in [30,40)

To the data!



What you will learn about in this section

1. B+ Trees: Basics

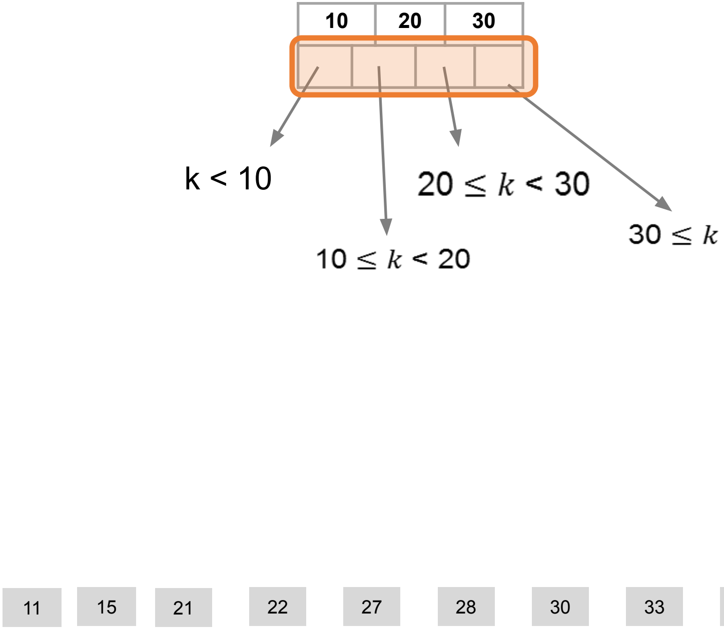
2. B+ Trees: Design & Cost
   - How many search values per page?
   - How many levels in tree?

3. Clustered Indexes

B+ Tree Basics -- Root, leaf and non-leaf nodes

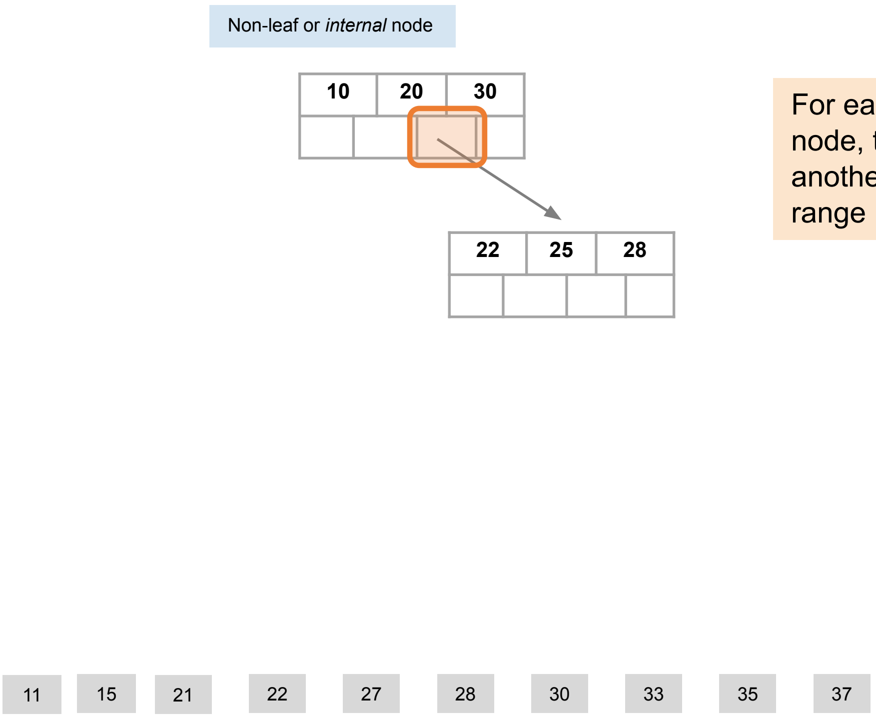
Parameter ***f*** = fanout

Each non-leaf node has *x* keys, x <= f keys



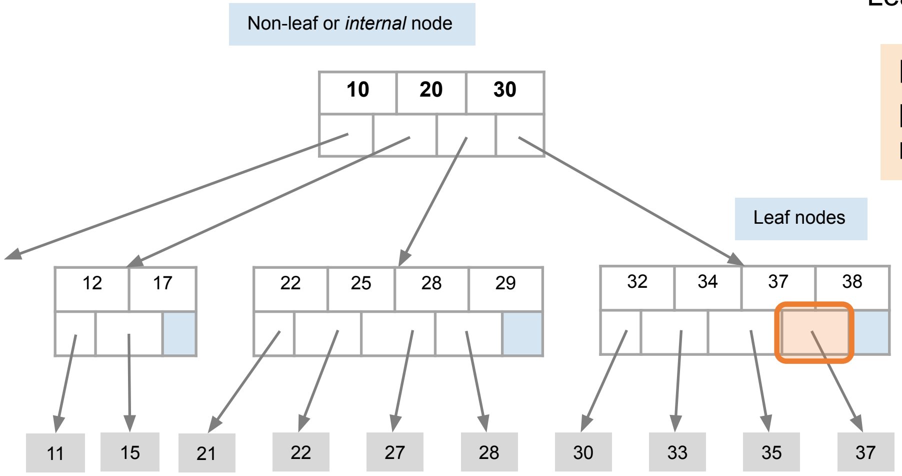
The *x* keys in a node define *x + 1* ranges

B+ Tree Basics -- Root, leaf and non-leaf nodes



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

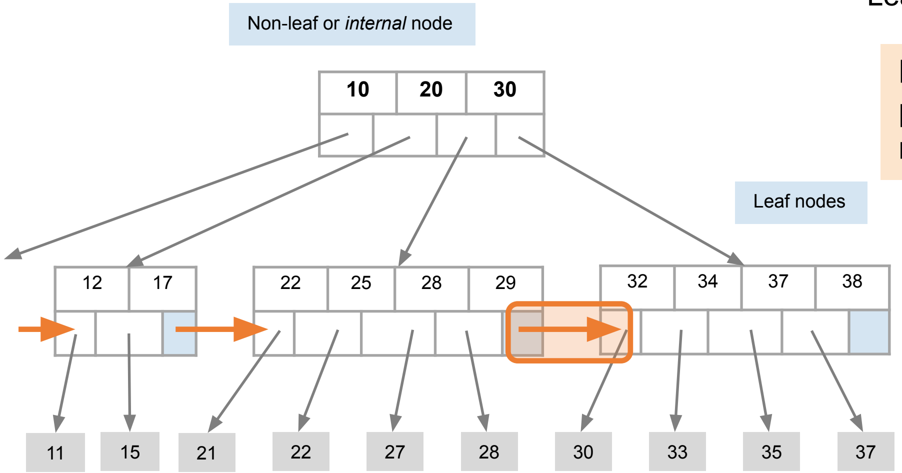
B+ Tree Basics -- Root, leaf and non-leaf nodes



Leaf nodes

Key slots contain pointers to data records

B+ Tree Basics -- Root, leaf and non-leaf nodes



Leaf nodes

Key slots contain pointers to data records

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

Costs of B+ trees

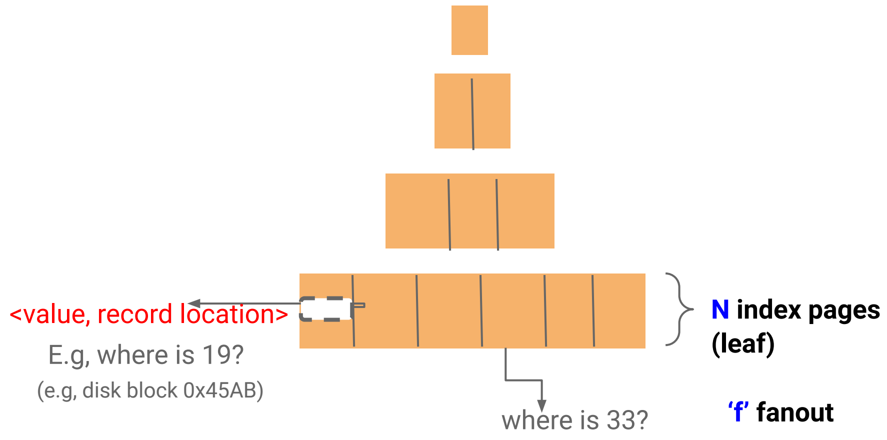
B+ Tree: High Fanout = Lower IO

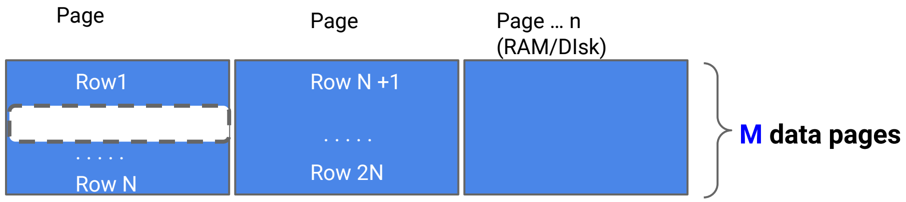
- As compared to e.g. binary search trees, B+ Trees have **high *fanout***
- Hence the **depth of the tree is small** → getting to any element requires very few IO operations!
    - Also can often store most/all of B+ Tree in RAM!

The <u>fanout</u> 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!

Cost Model for Indexes -- [Baseline simplest model]





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

Question: What's physical layout? What are costs?

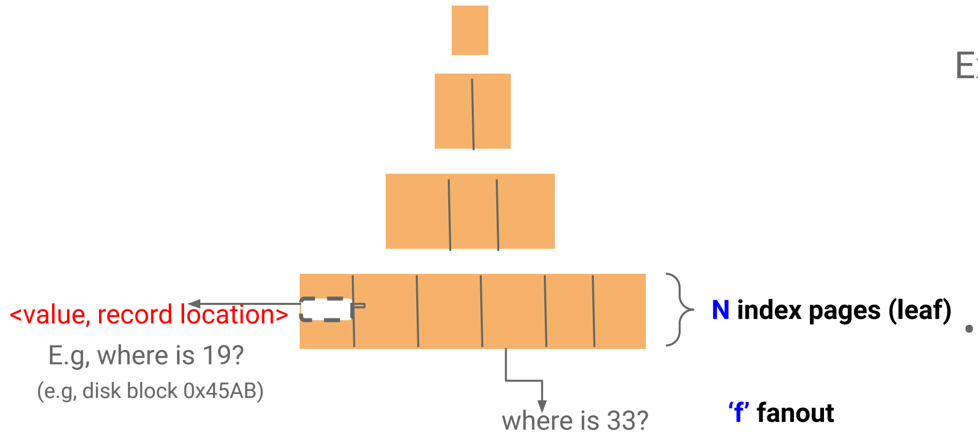
Let:

- f = fanout *(we'll assume it is constant for our cost model for simplicity…)*
- N = number of *pages* we need to index
- Height of tree = $\lceil \log_f N \rceil$

Key intuition

- 'M' depends on Table size
- 'N' depends on number of index values (e.g., <cname> or <cname, price, …> search keys)
- 'f' depends on key size and pointer size

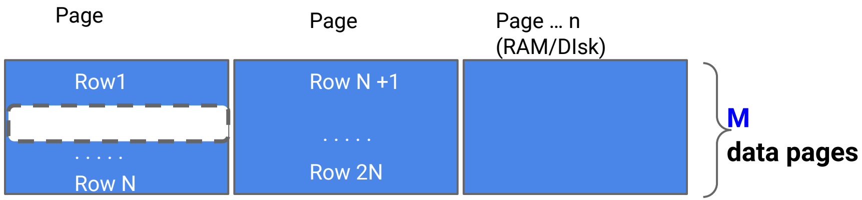
Cost Model for Indexes -- [Baseline simplest model]



Example 1:

- N = 2^40 index pages (~1 Trillion pages of 64KBs each)
- Value (or "search key") size = 4 bytes,
- "Location Pointer" size = 8 bytes

- We store one *node* per *page*
  f x 4 + (f+1) x 8 <= 64K → *f ~= 5460*



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

→ *h = 4 (i.e.,* $5460^h = 2^{40}$)

AMAZING, for big 'f'!! What about small 'f?'

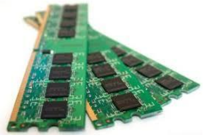
Example 2 -- What about small ‘f = 100’?

Level	Number of pages (Size)	Num of Index records
1	1 (64KB)	100
2	100 (6.4MB)	100^2
3	100^2 (0.64GB)	100^3
4	100^3 (64GB)	100^4 = 100 Million
5	...	...

Which levels will be in RAM, if you had
[a] 32 GB of RAM?
[b] 64 GB of RAM?

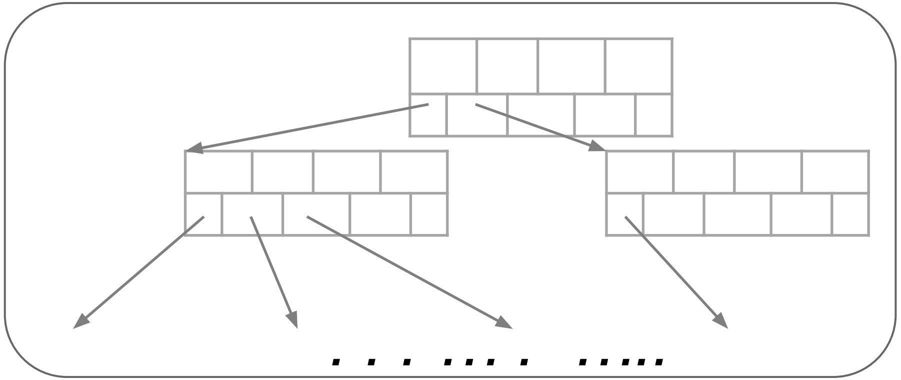
Other levels? Will (likely) cost a disk IO

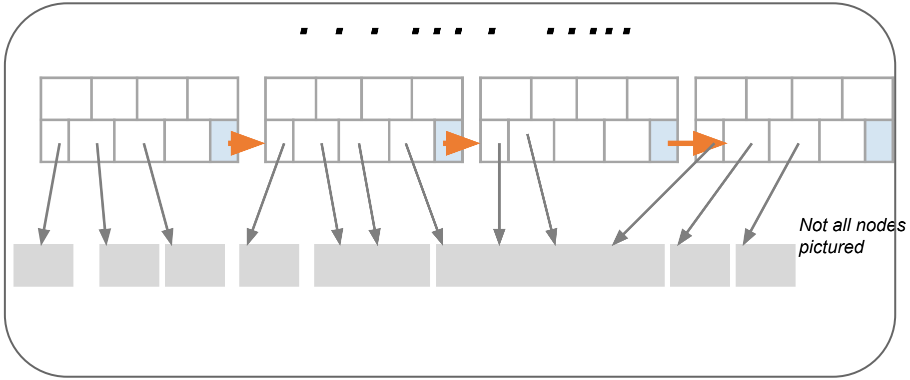
Search cost of B+ Tree (on RAM + Disk)



Read 1st levels
Into RAM buffer

Rest of index on
disk





$1 + f + f^2 + f^3 + \ldots <= \mathbf{B}$

Keep 1st L_B levels in RAM of size **B**

Algorithm: B+ Search
- Read 1 page per level
- Pages in RAM are free
- Read 1 page for record

IO Cost:
$\lceil \log_f N \rceil - L_B + 1$

Simple Cost Model for Range 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:

$\lceil \log_f N \rceil - L_B + \mathrm{Cost}(OUT)$

Fast Insertions & Self-Balancing

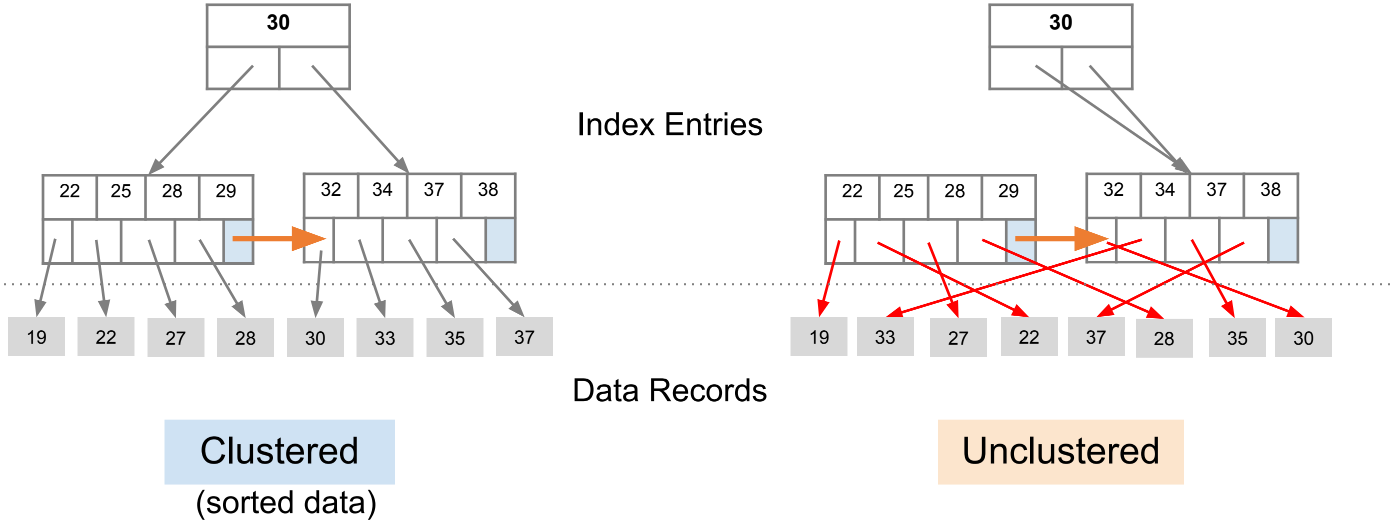
- We won’t go into specifics of B+ Tree insertion algorithm, but has several attractive qualities:
  - ~ Same cost as exact search
  - *Self-balancing:* B+ Tree remains balanced (with respect to height) even after insert

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

What you will learn about in this section

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

Clustered vs. Unclustered Index



An index is *clustered* if the underlying data is

ordered in the same way as the index's data entries.

Clustered vs. Unclustered Index

- Recall that sequential disk block IO is much faster than random IO

- For exact search, no difference between clustered / unclustered

- For range search over R values: difference between
  - [a] 1 random IO + R sequential IO and [b] 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