Lecture 11: B+ Trees: An IO-Aware Index Structure

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

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

Today's Lecture

1. Indexes: Motivations & Basics

2. B+ Trees

1. Indexes: Motivations & Basics

What you will learn about in this section

1. Indexes: Motivation
2. Indexes: Basics
3. ACTIVITY: Creating indexes

Index Motivation

`Person(name, age)`

- Suppose we want to search for people of a specific age

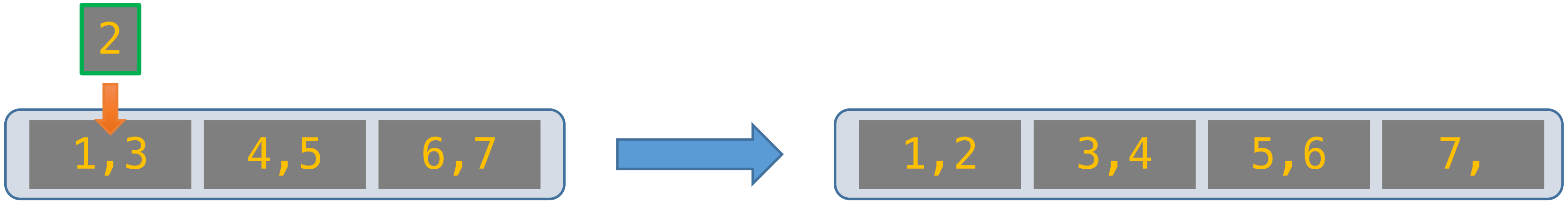
- ***First idea:*** Sort the records by age… we know how to do this fast!

- 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 Motivation

- What about if we want to **insert** a new person, but keep the 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?

Index Motivation

- What about if we want to be able to search quickly along multiple attributes (e.g. not just age)?
  - We could keep multiple copies of the records, each sorted by one attribute set… this would take a lot of space

Can we get fast search over multiple attribute (sets) without taking too much space?

We'll create separate data structures called *indexes* to address all these points

Further Motivation for Indexes: NoSQL!

- NoSQL engines are (basically) ***just indexes!***
  - A lot more is left to the user in NoSQL… one of the primary remaining functions of the DBMS is still to provide index over the data records, for the reasons we just saw!
  - Sometimes use B+ Trees (covered next), sometimes hash indexes (not covered here)

Indexes are critical across all DBMS types

Indexes: High-level

- An ***index*** on a file speeds up selections on the ***search key fields*** for the index.
  - Search key properties
    - Any subset of fields
    - is **not** the same as *key of a relation*

- *Example:*

`Product(name, maker, price)`

On which attributes would you build indexes?

More precisely

- An *index* is a **data structure** mapping search keys to sets of rows in a database table
  - Provides efficient lookup & retrieval by search key value- usually much faster than searching through all the rows of the database table
- An index can store the full rows it points to (*primary index*) or pointers to those rows (*secondary index*)
  - We'll mainly consider secondary indexes

Operations on an Index

- <u>Search</u>: Quickly find all records which meet some *condition on the search key attributes*
  - More sophisticated variants as well. Why?

- <u>Insert / Remove</u> entries
  - Bulk Load / Delete. Why?

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

Conceptual Example

What if we want to return all books published after 1867? The above table might be very expensive to search over row-by-row…

Russian_Novels

BID	Title	Author	Published	Full_text
001	*War and Peace*	Tolstoy	1869	…
002	*Crime and Punishment*	Dostoyevsky	1866	…
003	*Anna Karenina*	Tolstoy	1877	…

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

Conceptual Example

By_Yr_Index

Published	BID
1866	002
1869	***001***
1877	***003***

Russian_Novels

BID	Title	Author	Published	Full_text
001	*War and Peace*	Tolstoy	1869	…
002	*Crime and Punishment*	Dostoyevsky	1866	…
003	*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?

Conceptual Example

By_Yr_Index

Published	BID
1866	002
1869	001
1877	003

Russian_Novels

BID	Title	Author	Published	Full_text
001	*War and Peace*	Tolstoy	1869	…
002	*Crime and Punishment*	Dostoyevsky	1866	…
003	*Anna Karenina*	Tolstoy	1877	…

By_Author_Title_Index

Author	Title	BID
Dostoyevsky	Crime and Punishment	002
Tolstoy	Anna Karenina	003
Tolstoy	War and Peace	001

Can have multiple indexes to support multiple search keys

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

Composite Keys

Name	Age	Sal
Bob	12	10
Cal	11	80
Luda	12	20
Tara	13	75

11	80
12	10
12	20
13	75

<Age, Sal>

<age,sal> not equal to <sal,age>

80	11
10	12
20	12
75	13

<Sal, Age>

11
12
12
13

<Age>

80
10
20
75

<Sal>

<ins>Equality Query:</ins>
Age = 12 and Sal = 90?

<ins>Range Query:</ins>
Age = 5 and Sal > 5?

Composite keys in *<ins>Dictionary Order.</ins>*

On which attributes can we do range queries?

Composite Keys

- Pro:
  - When they work they work well
  - We’ll see a good case called “index-only” plans or **covering** indexes.
- Con:
  - Guesses? (time and space)

Covering Indexes

By_Yr_Index

Published	BID
1866	002
1869	001
1877	003

We say that an index is **covering** *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 Russian_Novels
WHERE Published > 1867
```

High-level Categories of 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 *(not covered)*
  - There are variants of this basic structure to deal with IO
  - Called ***linear*** or ***extendible hashing-*** IO aware!

The data structures we present here are “IO aware”

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

[DB-WS11a.ipynb]

2. B+ Trees

What you will learn about in this section

1. B+ Trees: Basics

2. B+ Trees: Design & Cost

3. Clustered Indexes

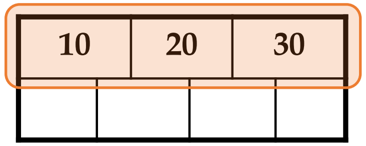
B+ Trees

- 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

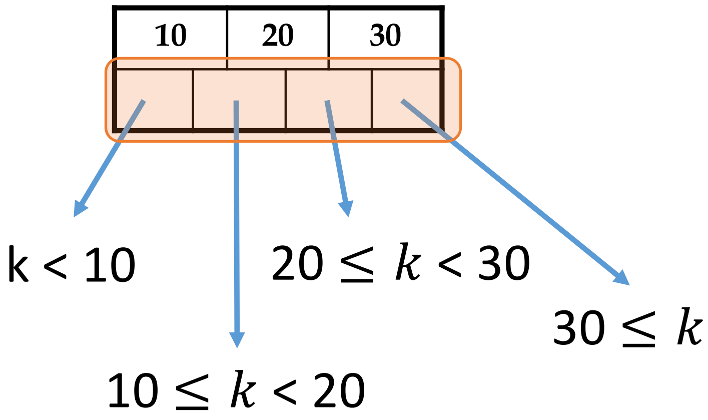


Parameter ***d*** = the degree

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

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

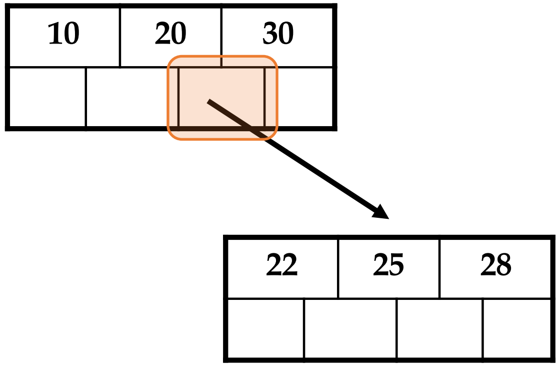
B+ Tree Basics



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

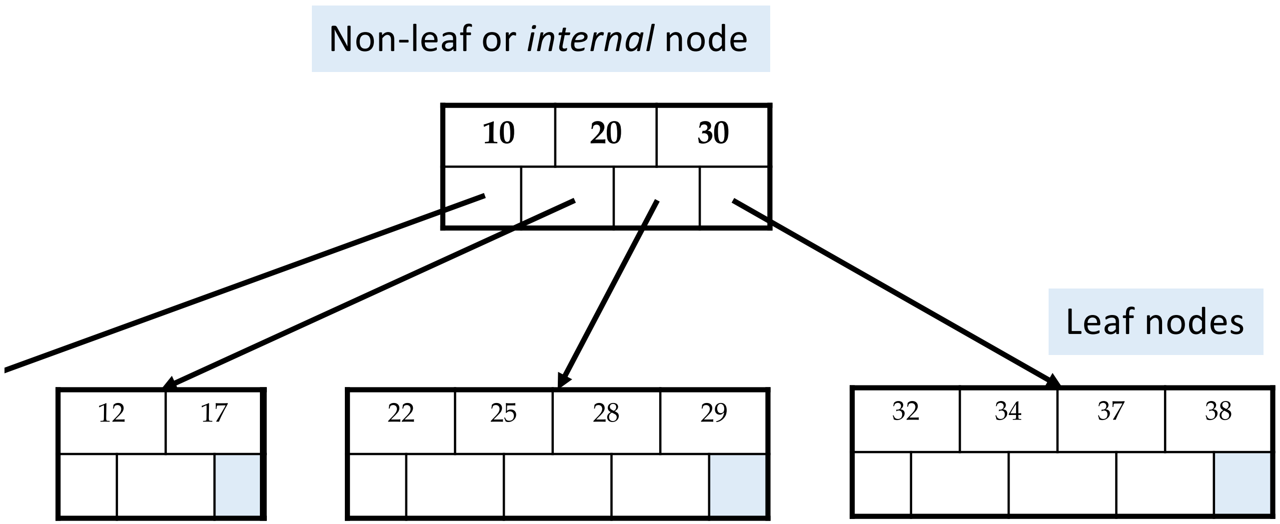
B+ Tree Basics

Non-leaf or *internal* node



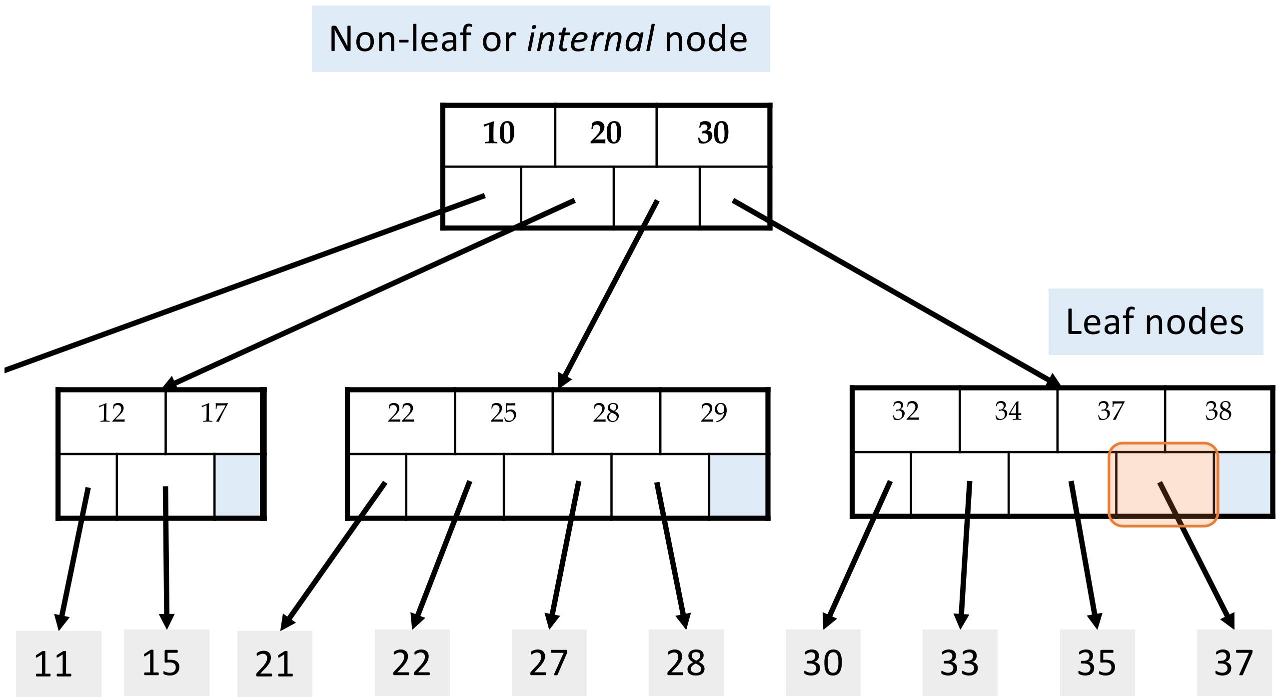
For each range, in a *non-leaf* node, there is a **pointer** to another node with keys in that range

B+ Tree Basics



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

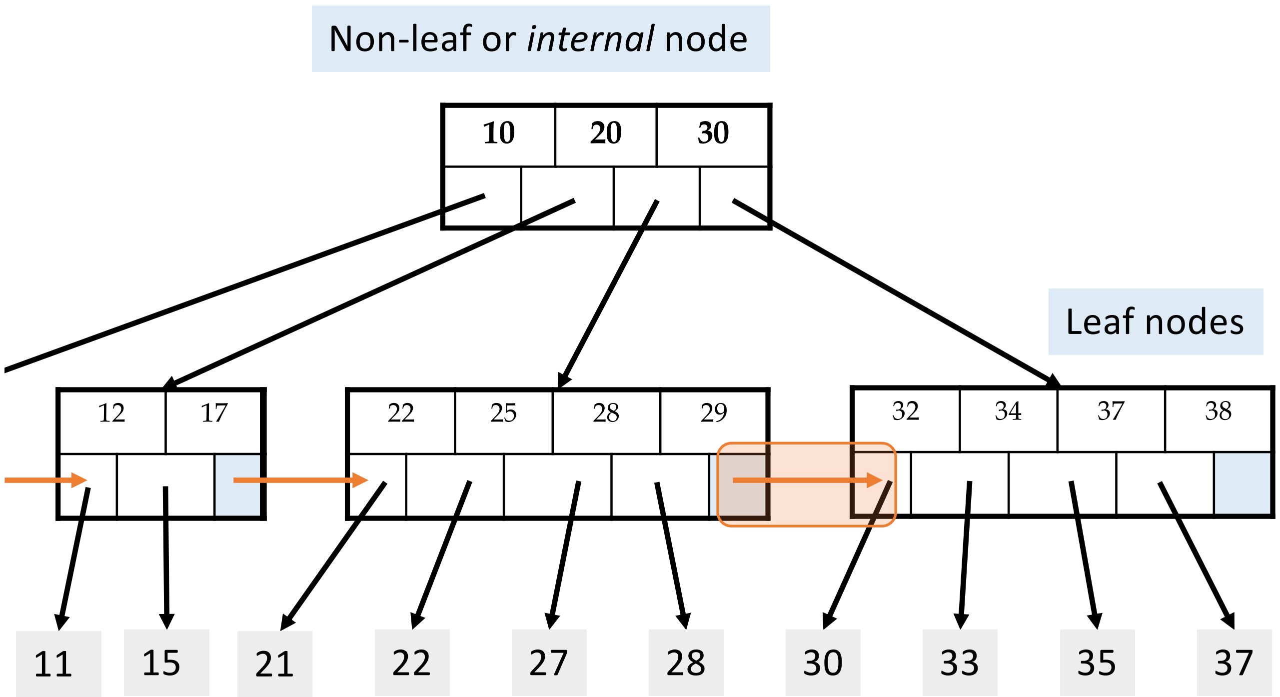
B+ Tree Basics



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

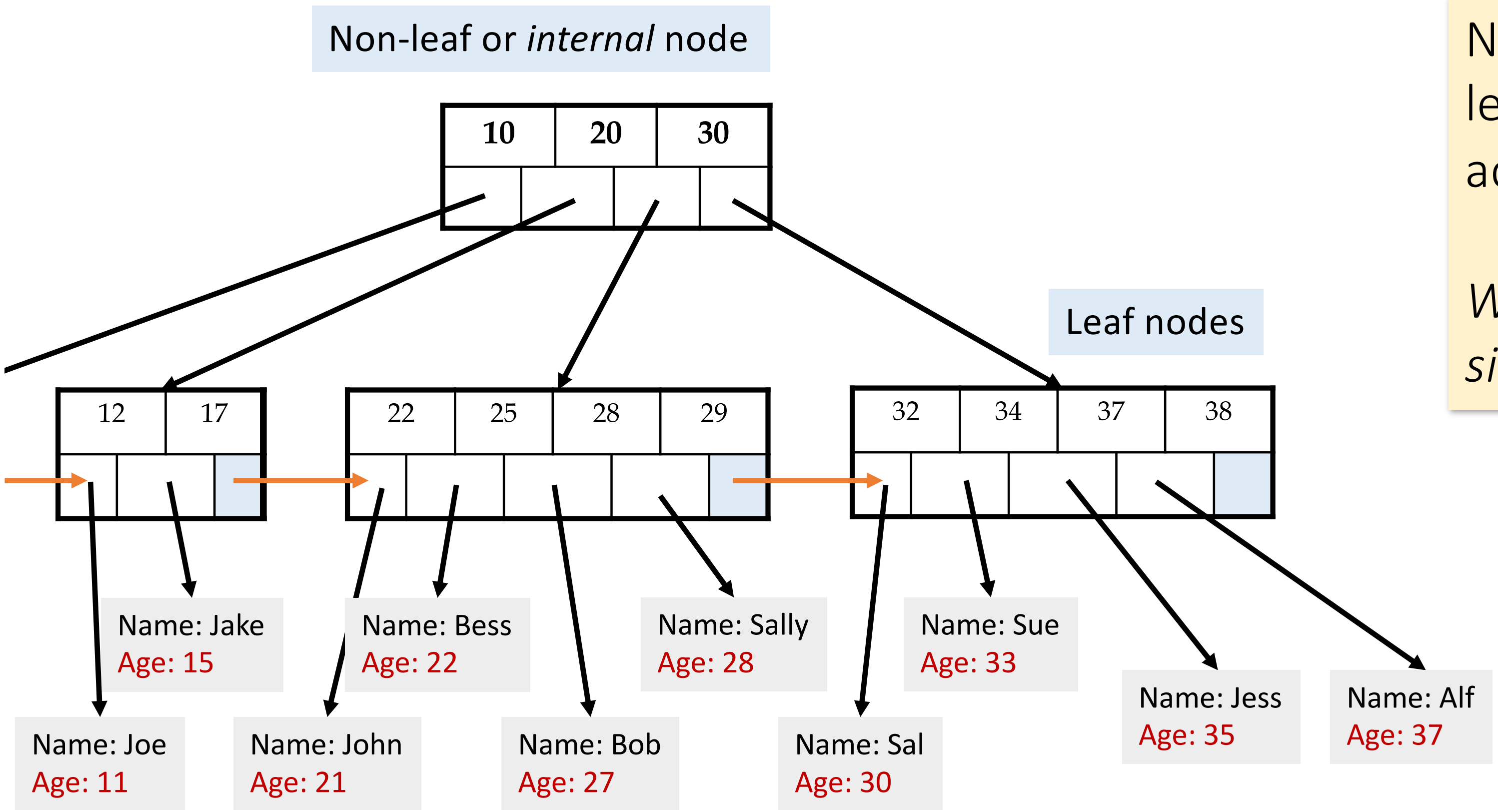


Leaf nodes also have between *d* and *2d* keys, and are different in that:

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

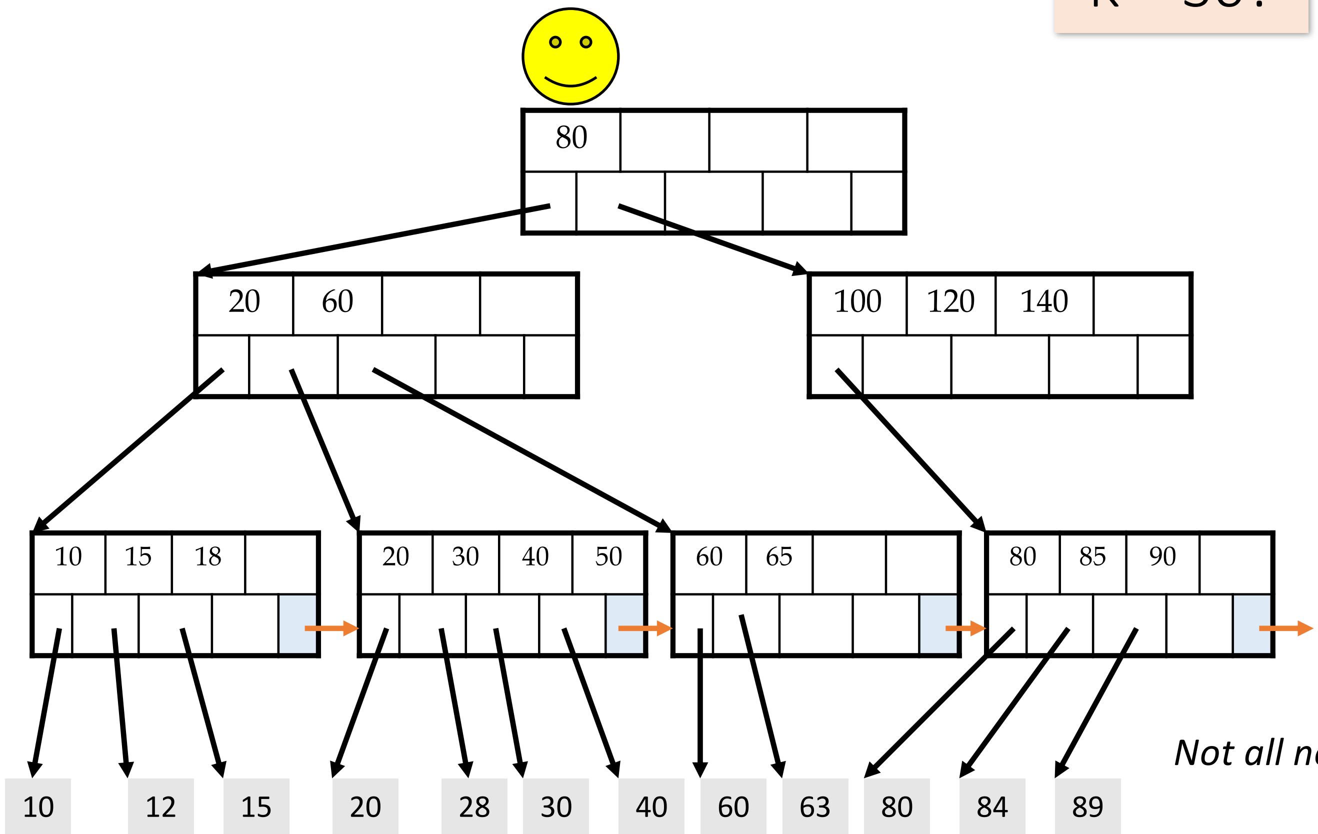
K = 30?

30 < 80

30 in [20,60)

30 in [30,40)

To the data!



Not all nodes pictured

B+ Tree Range Search Animation

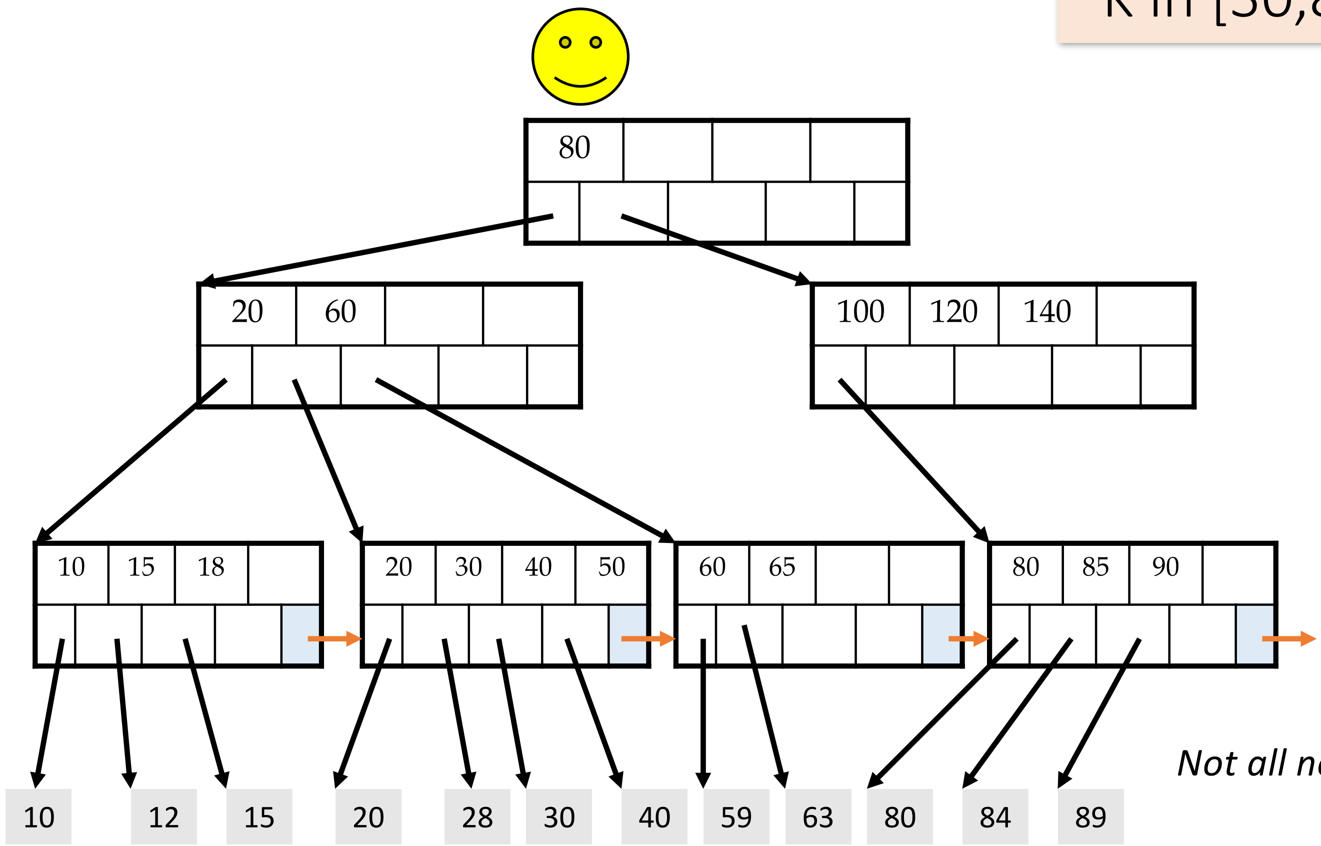
K in [30,85]?

30 < 80

30 in [20,60)

30 in [30,40)

To the data!



Not all nodes pictured

B+ Tree Design

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

- Example:
  - Key size = 4 bytes
  - Pointer size = 8 bytes
  - Block size = 4096 bytes

NB: Oracle allows 64K = 2^16 byte blocks
→ d <= 2730

- We want each *node* to fit on a single *block/page*
  - 2d x 4 + (2d+1) x 8 <= 4096 → ***d <= 170***

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***)
- This means that the **depth of the tree is small** → getting to any element requires very few IO operations!
  - Also can often store most or all of the B+ Tree in main memory!
- 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 <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!

The known universe contains ~10^{80} particles… what is the height of a B+ Tree that indexes these?

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

B+ Tree Range Search Animation

K in [30,85]?

How many IOs did our friend do?

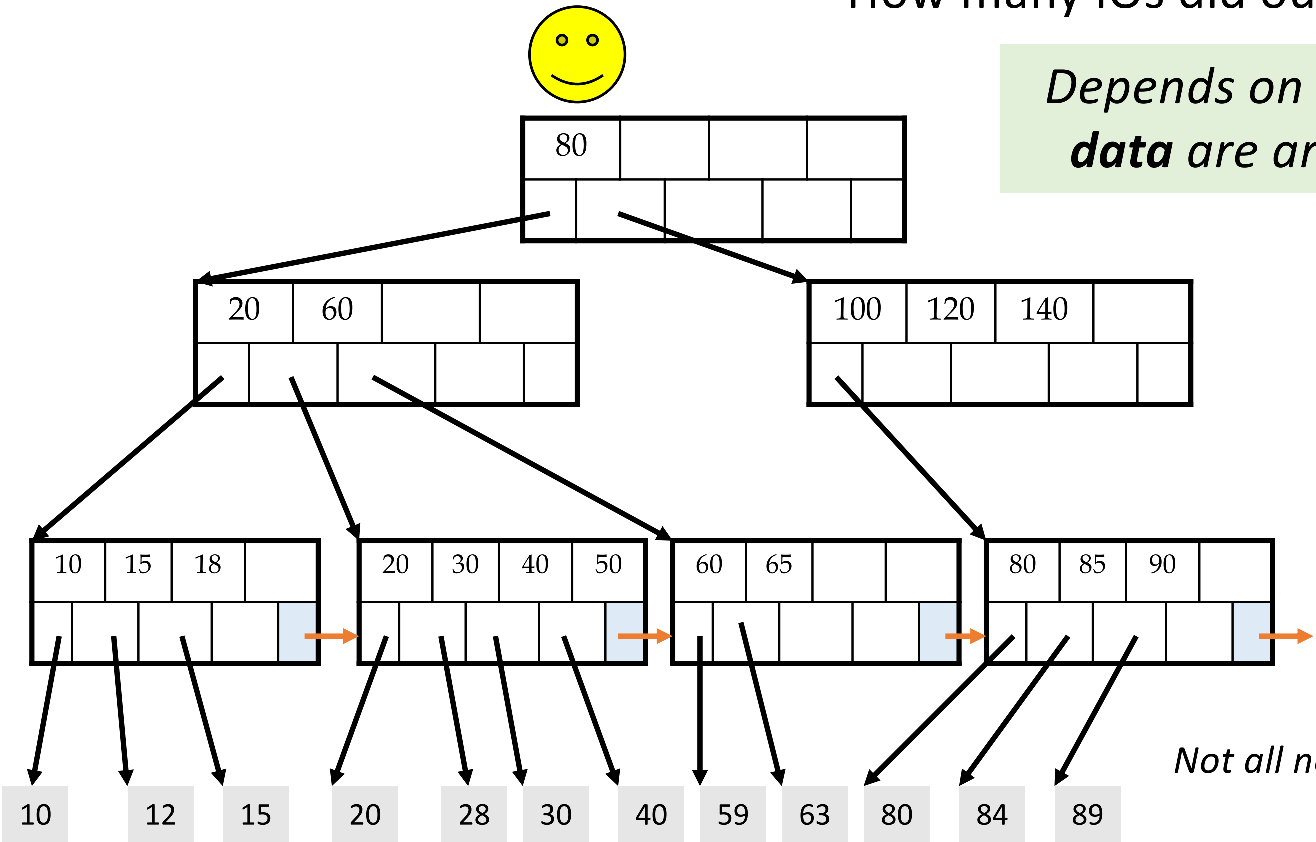
*Depends on **how the data** are arranged*

30 < 80

30 in [20,60)

30 in [30,40)

To the data!

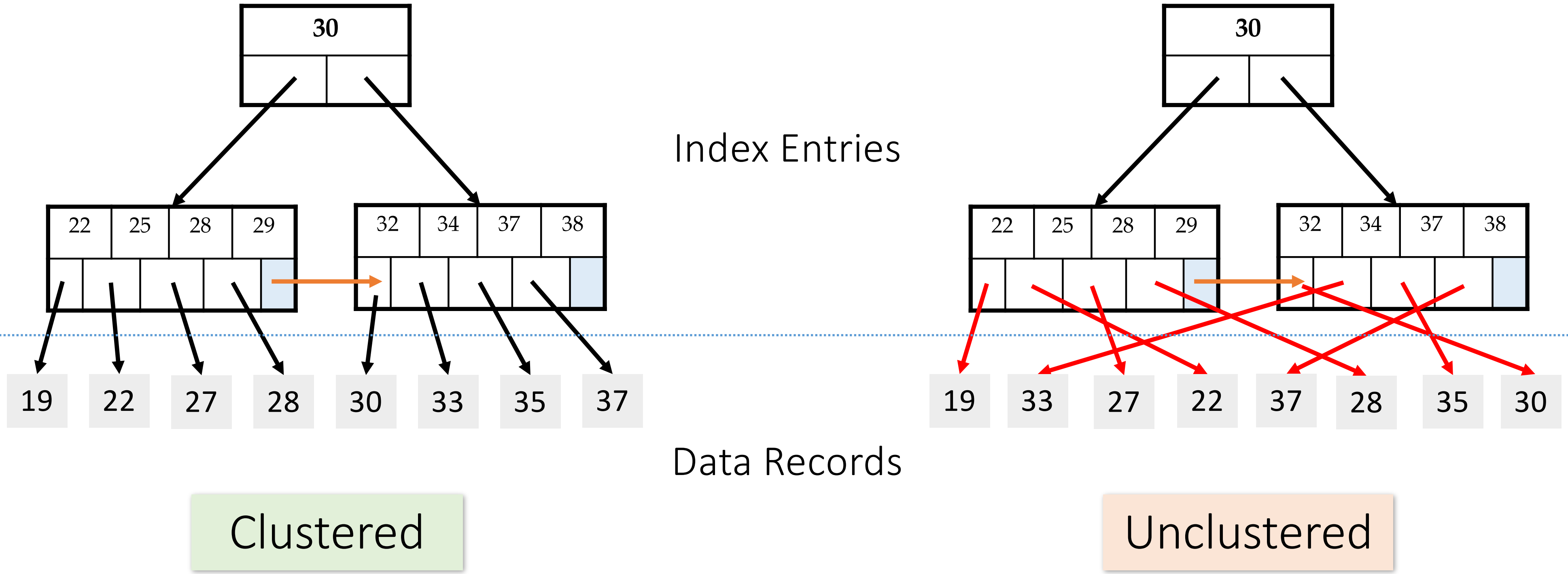


Not all nodes pictured

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!**

Fast Insertions & Self-Balancing

- We won't go into specifics of B+ Tree insertion algorithm, but has several attractive qualities:

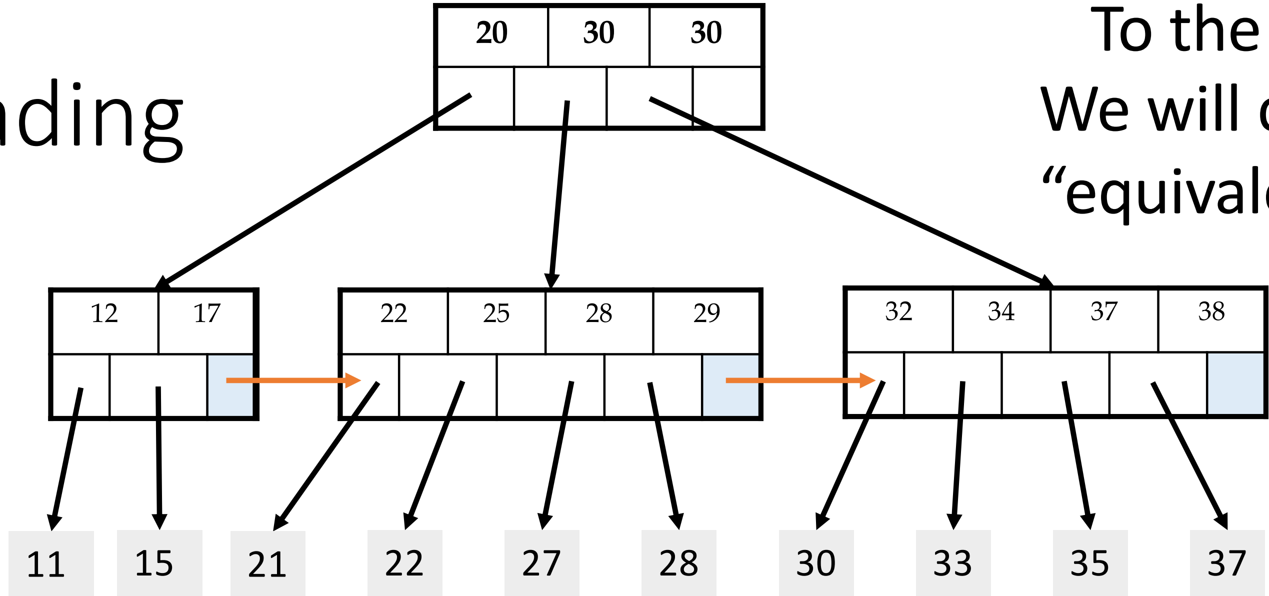
  - **~ Same cost as exact search**

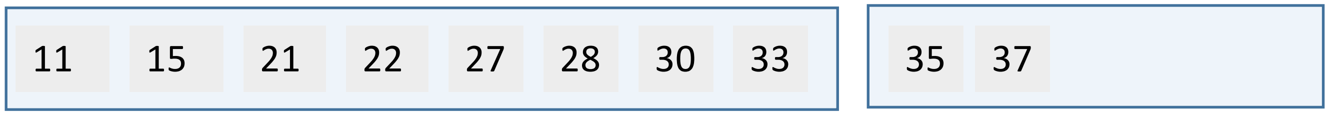
  - ***Self-balancing:*** B+ Tree remains **balanced** (with respect to height) even after insert

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

Bulk Loading

To the board!
We will create an
“equivalent” tree





Input: Sorted File
Output: B+ Tree

Message: Bulk Loading is faster!

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