THE B+ TREE INDEX

CS 564- Fall 2016

ACKs: *Jignesh Patel, AnHai Doan*

RECAP

- We have the following query:

```
SELECT *
FROM Sales
WHERE price > 100 ;
```

- How do we organize the file to answer this query efficiently?

INDEXES

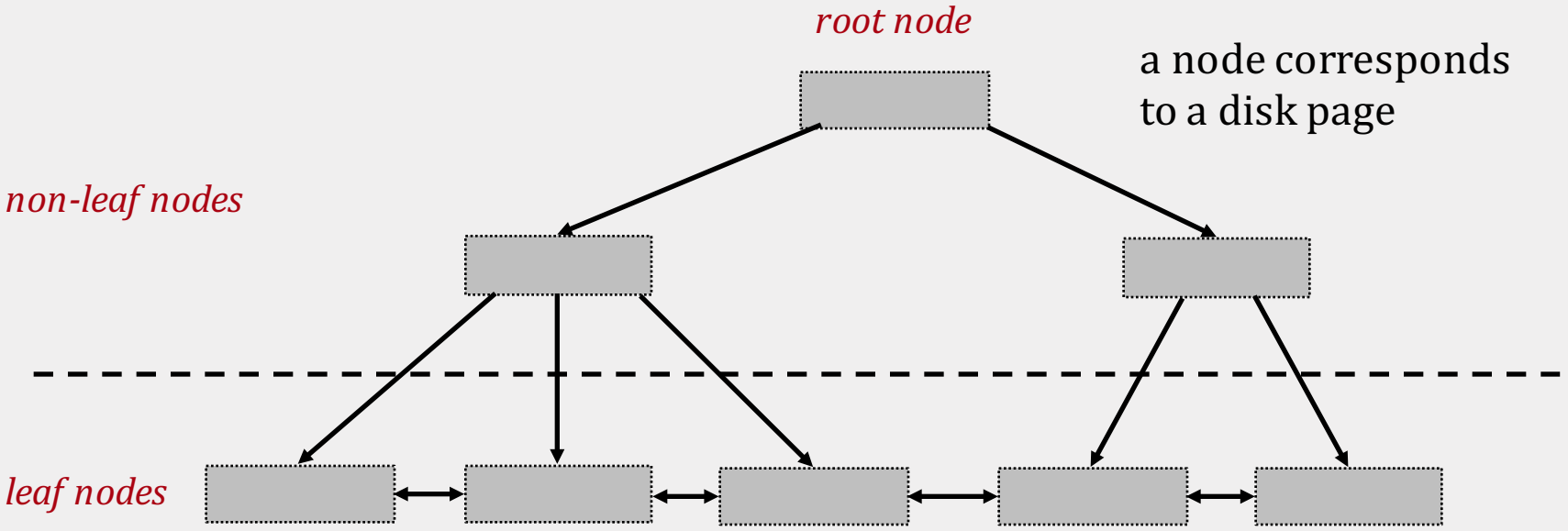
Two main types of indexes

- Hash index:
  - good for equality search
  - in expectation $O(1)$ I/O cost for search and insert
- B+ tree index:
  - good for range and equality search
  - $O(log_F(N))$ I/O cost for search, insert and delete

The B+ Tree Index

- a dynamic tree-structured index
  - adjusted to be always height-balanced
- supports efficient equality and range search
- widely used in many DBMSs
  - SQLite uses it as the default index
  - SQL Server, DB2, …

B+ Tree Index Basics



data entries

- exist *only* in the leaf nodes
- are sorted according to the search key

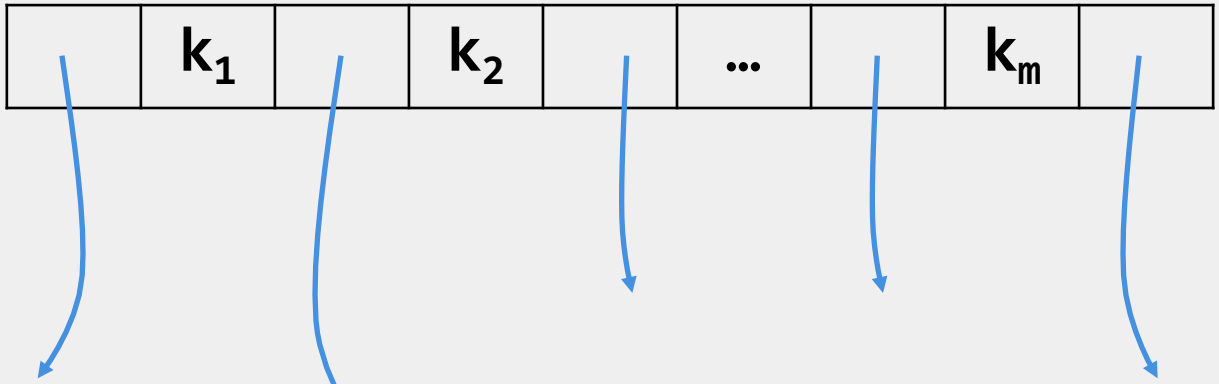
B+ Tree Basics

- **d** is the *order* of the tree

- Each node contains $d \leq m \leq 2d$ entries
  - minimum 50% occupancy at all times
- The root can contain $1 \leq m \leq 2d$ entries

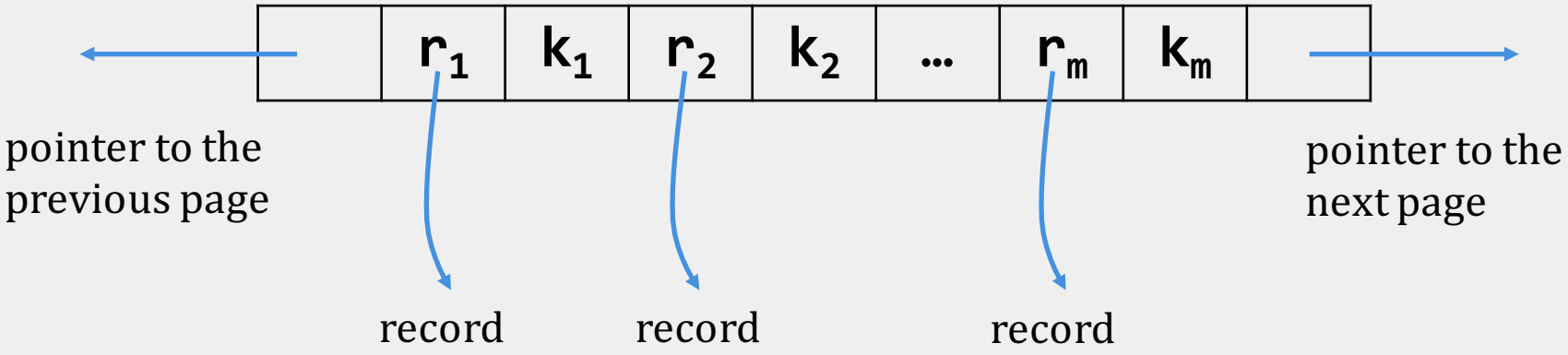
Non-Leaf Node

- An non-leaf node with *m* entries has *m+1* pointers to lower-level nodes



Leaf Node

- A leaf node with *m* entries has
  - *m* pointers to the data records (rids)
  - pointers to the next and previous leaves



B+ Trees in Practice

- typical order = 100
- typical fill factor = 67%
  - average node fanout = 133
- typical B+ tree capacities:
  - *height 4*: 133^4 = 312,900,700 records
  - *height 3*: 133^3 = 2,352,637 records
- It can often store the top levels in buffer pool:
  - *level 1* = 1 page = 8 KB
  - *level 2* = 133 pages = 1 MB
  - *level 3* = 17,689 pages = 133 MB

B+ Tree Operations

A B+ tree supports the following operations:

- equality search
- range search
- insert
- delete
- bulk loading

B+ Tree: Search

- start from root
- examine index entries in non-leaf nodes to find the correct child
- traverse down the tree until a leaf node is reached
- non-leaf nodes can be searched using a binary or a linear search

B+ Tree: Insert

- find correct leaf node **L**
- insert data entry in **L**
  - If **L** has enough space, DONE!
  - Else, we must split **L** (into **L** and a new node **L'**)
    - redistribute entries evenly, **copy up** the middle key
    - insert index entry pointing to **L'** into parent of **L**
- This can propagate recursively to other nodes!
  - to split a non-leaf node, redistribute entries evenly, but **push up** the middle key

B+ Tree: Delete

- find leaf node **L** where entry belongs
- remove the entry
  - If **L** is at least half-full, DONE!
  - If **L** has only *d-1* entries,
    - Try to **re-distribute**, borrowing from sibling
    - If re-distribution fails, **merge L** and sibling
- If a merge occurred, we must delete an entry from the parent of **L**

B+ Tree Cost

The cost of an insert/delete operation in a B+ tree is $O\left(log_F(N)\right)$ I/Os

- F = node fanout (number of children)
- N = # leaf pages

DUPLICATES

- duplicate keys: many data entries with the same key value
- Solution 1:
  - All entries with a given key value reside on a single page
  - Use overflow pages
- Solution 2:
  - Allow duplicate key values in data entries
  - Modify search operation