THE B+ TREE INDEX

CS 564- Fall 2015

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/Os and CPU performance 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

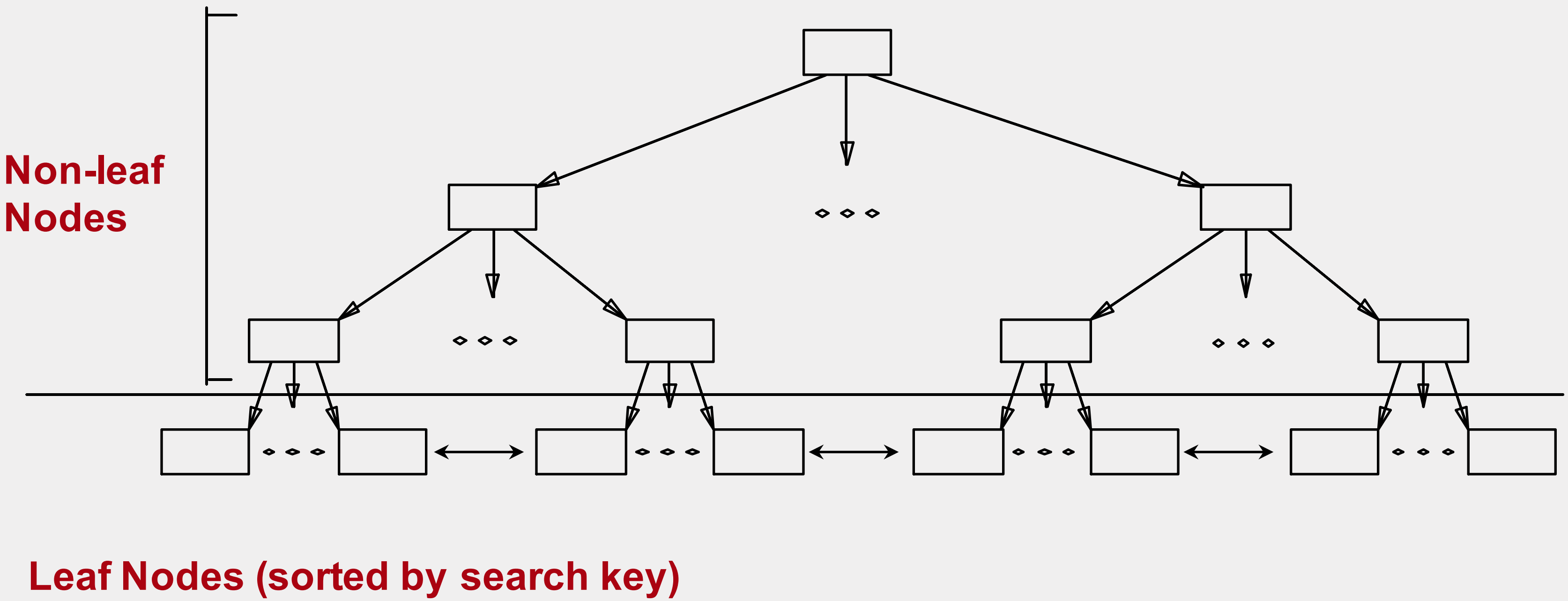
- dynamic tree-structured index
  - adjusted to be height-balanced
- supports efficient equality and range search
- widely used in many DBMSs (SQLite uses it for example)

B+ Tree Basics

- **d** = the order of the tree
- Each node contains $d \leq m \leq 2d$ entries
  - minimum 50% occupancy at all times
  - **exception**: the root can contain $1 \leq m \leq 2d$ entries

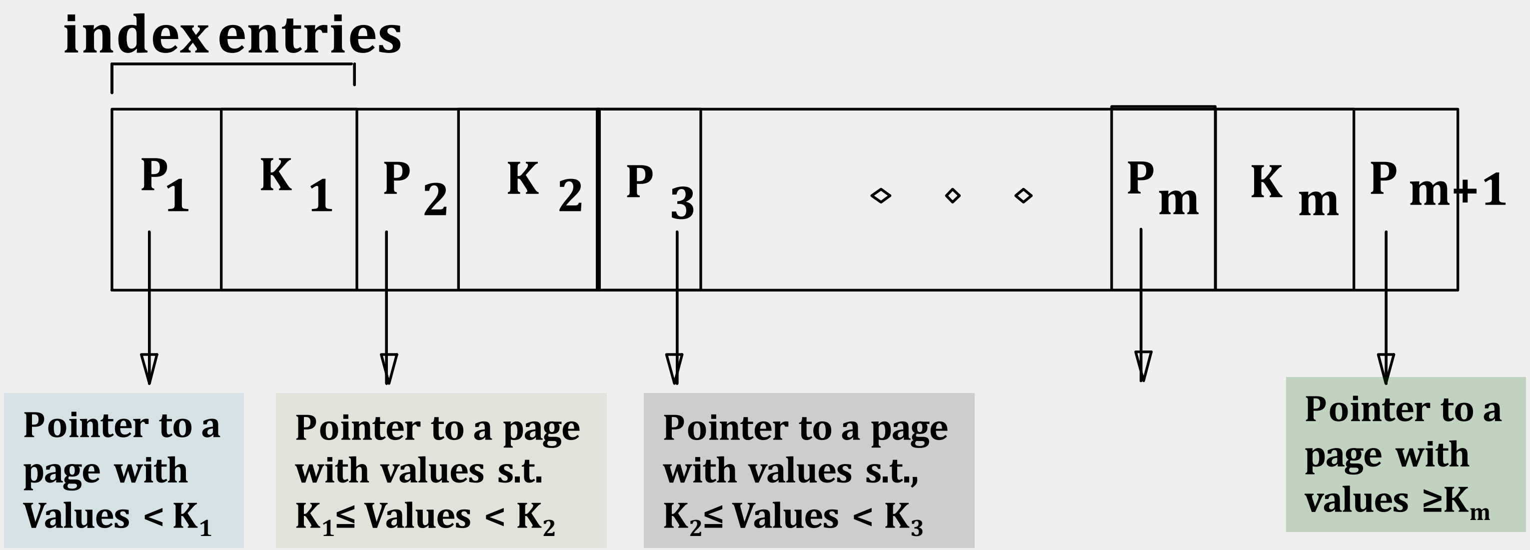
- The cost of an insert/delete is $O\left(log_F(N)\right)$ I/Os
  - F = fanout of a node
  - N = # leaf pages

B+ Tree Index Basics



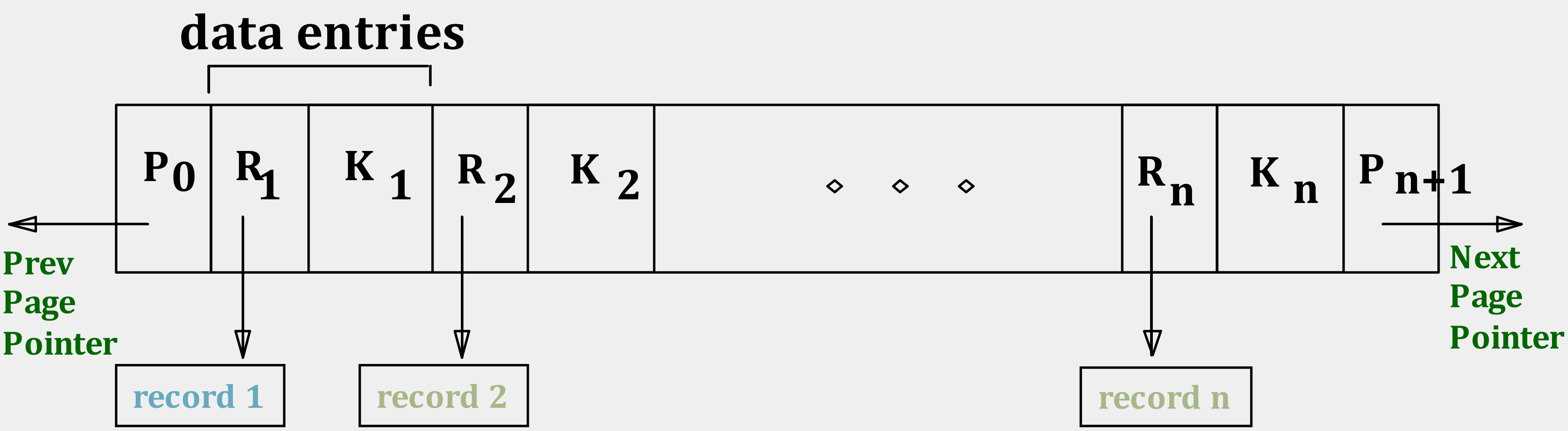
NON-LEAF NODE

- 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)
  - a pointer to the next leaf



B+ Tree in Practice

- Typical order: 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
- Can often hold 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 load

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, must split **L** (into **L** and a new node $\mathbf{L_2}$)
    - Redistribute entries evenly, **copy up** middle key
    - Insert index entry pointing to $\mathbf{L_2}$ into parent of **L**
- This can propagate recursively to other nodes!
  - To split non-leaf node, redistribute entries evenly, but **pushing 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**

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