Overview of Storage and Indexing
Storing Data: Disks and Files

Chapters 8-9
Indexes

- An index on a file speeds up selections on the search key fields for the index.
  - Any subset of the fields of a relation can be the search key for an index on the relation.
  - Search key is not the same as key (minimal set of fields that uniquely identify a record in a relation).

- An index contains a collection of data entries, and supports efficient retrieval of all data entries $k^*$ with a given key value $k$.
  - Given data entry $k^*$, we can find record with key $k$ in at most one disk I/O.
B+ Tree Indexes

- Leaf pages contain *data entries*, and are chained (prev & next)
- Non-leaf pages have *index entries*; only used to direct searches:

```
+---------+---------+---------+
| P0      | K1      | P1      |
+---------+---------+---------+---------+---------+---------+
|         |   K2    |         |         |     Km   |         |
|         |   P2    |         |         |     Pm   |         |
```

Database Management Systems 3ed, R. Ramakrishnan and J. Gehrke
Example B+ Tree

Entries <= 17

Entries > 17

- Find 28*? 29*? All > 15* and < 30*
- Insert/delete: Find data entry in leaf, then change it. Need to adjust parent sometimes.
  - And change sometimes bubbles up the tree

Note how data entries in leaf level are sorted
**B+ Tree: Most Widely Used Index**

- Insert/delete at log \( F \) \( N \) cost; keep tree *height-balanced*. \((F = \text{fanout}, N = \# \text{leaf pages})\)
- Minimum 50% occupancy (except for root). Each node contains \( d \leq m \leq 2d \) entries. The parameter \( d \) is called the *order* of the tree.
- Supports equality and range-searches efficiently.

![Diagram of B+ Tree](image-url)
B+ Trees 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 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes
Inserting a Data Entry into a B+ Tree

- Find correct leaf $L$.
- Put data entry onto $L$.
  - If $L$ has enough space, done!
  - Else, must **split** $L$ (into $L$ and a new node $L2$)
    - Redistribute entries evenly, **copy up** middle key.
    - Insert index entry pointing to $L2$ into parent of $L$.
- This can happen recursively
  - To split index node, redistribute entries evenly, but **push up** middle key. (Contrast with leaf splits.)
- Splits “grow” tree; root split increases height.
  - Tree growth: gets **wider** or **one level taller at top**.
Inserting 8* into Example B+ Tree
Deleting a Data Entry from a B+ Tree

- Start at root, find leaf 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 (adjacent node with same parent as L).
    - If re-distribution fails, merge L and sibling.
- If merge occurred, must delete entry (pointing to L or sibling) from parent of L.
- Merge could propagate to root, decreasing height.
Deleting 19*, 20* into Example B+ Tree
Deleting 24* into Example B+ Tree
Hash-Based Indexes

- Good for equality selections.
- Index is a collection of *buckets*.
  - Bucket = *primary page* plus zero or more *overflow pages*.
  - Buckets contain data entries.
- *Hashing function* $h$: $h(r) =$ bucket in which (data entry for) record $r$ belongs. $h$ looks at the *search key* fields of $r$. 
Static Hashing

- # primary pages fixed, allocated sequentially, never de-allocated; overflow pages if needed.
- \( h(k) \mod M \) = bucket to which data entry with key \( k \) belongs. (\( M = \# \) of buckets)
Static Hashing (Contd.)

- Buckets contain data entries.
- Hash fn works on search key field of record r. Must distribute values over range 0 ... M-1.
  - \( h(key) = (a \times key + b) \) usually works well.
  - a and b are constants; lots known about how to tune h.
- Long overflow chains can develop and degrade performance.
  - Extendible and Linear Hashing: Dynamic techniques to fix this problem.
Alternatives for Data Entries (Contd.)

- **Alternative 1:**
  - If this is used, index structure is a file organization for data records (instead of a Heap file or sorted file).
  - At most one index on a given collection of data records can use Alternative 1. (Otherwise, data records are duplicated, leading to redundant storage and potential inconsistency.)
  - If data records are very large, # of pages containing data entries is high. Implies size of auxiliary information in the index is also large, typically.
Alternatives for Data Entries (Contd.)

- Alternatives 2 and 3:
  - Data entries typically much smaller than data records. So, better than Alternative 1 with large data records, especially if search keys are small. (Portion of index structure used to direct search, which depends on size of data entries, is much smaller than with Alternative 1.)
  - Alternative 3 more compact than Alternative 2, but leads to variable sized data entries even if search keys are of fixed length.
Index Classification

- **Primary vs. secondary:** If search key contains primary key, then called primary index.
  - *Unique* index: Search key contains a candidate key.

- **Clustered vs. unclustered:** If order of data records is the same as, or `close to`, order of data entries, then called clustered index.
  - Alternative 1 implies clustered; in practice, clustered also implies Alternative 1 (since sorted files are rare).
  - A file can be clustered on at most one search key.
  - Cost of retrieving data records through index varies greatly based on whether index is clustered or not!
Clustered vs. Unclustered Index

- Suppose that Alternative (2) is used for data entries, and that the data records are stored in a Heap file.
  - To build clustered index, first sort the Heap file (with some free space on each page for future inserts).
  - Overflow pages may be needed for inserts. (Thus, order of data recs is `close to’, but not identical to, the sort order.)