Primary and Secondary Organizations

• Primary file organization:
  - determines how the files of records are physically placed on the disk
  - heap files, sequential files (search key), hashed files (hash key)

• Secondary organization:
  - allows efficient access to the records of a file based on alternate fields that those used for the primary file organization

Index Structures

• Index: secondary access structure used to speed up the retrieval of records in response to certain search conditions

• Most frequently used index types:
  - Single-level ordered indexes: primary index, clustering index and secondary index
  - Multilevel indexes: search trees (B- and B*- trees)

Search Trees (1)

• Search tree: special case of tree used to guide the search for a record based on a value of the search field

• A search tree (k) is a tree that each node of it contains at most 2k search values and 2k+1 tree pointers

V1 < V2 < ... < Vq

We select suitable k so that a whole node fits in one disk page
**Search Trees (2)**

- **Trade-off:**
  - Performance of data structure maintenance (insert/delete may require restructuring)
  - Speed of searching
  - Space required (redundancy...)

- **Desired ("the ideal tree"):**
  - Low tree height \( h \) (high width = high branching factor)
    - Access steps \( h \leq \log_k(n) \)
  - Balancing (all leaf nodes at same level)
  - Low maintenance / modification efforts required
  - Each node at least 50% full

**B- vs. B*-Trees**

- **B-tree:** fulfils the desired requirements of the ideal tree
- **Structure:**
  - Similar structure with the generic search tree
  - Internal node data values together with data pointers

- **B*-tree:** Enhanced B-tree
- **Structure:**
  - Data values and pointers stored only in leaf nodes
  - Internal nodes contain only some data values for navigating
  - Internal nodes at most 2\(k\) data values, leaf nodes at most \(2k^*\) data values
  - Leaf nodes linked together to provide ordered access

**B*-Trees**

- **Basic Ideas (k, k\(*\):**
  - Internal nodes containing \(n \in [k, 2k]\) data values, \(n + 1\) pointers to subtrees
  - Leaf nodes containing \(n \in [k^*, 2k^*]\) data values and data pointers, 2 pointers for seq. access (leaf nodes chained in both directions)
  - Root is leaf or inner node with \(n \in [1, 2k]\) keys
  - Data values of each leaf are sorted

- Internal nodes include search values and tree pointers without data pointers => more info packed in comparison with the B-tree => fewer levels reducing the search time

**B*-Trees: Search Operation**

- **Search for a value p:**
  - Start from the root node
  - Search every node from left to right
  - Compare each value \(V\) with \(p\) and branch accordingly...
  - When in leaf, search from left to right (success?)

- (Search cost: \(h\) page accesses)
B*-Trees: Insert Operation (1)

- Insert a value $p$:
  - Find leaf node to insert (search operation)
  - If leaf node not full, insert (success)
  - If leaf node full, the node overflows and must be split. Leave in this node $k^* + 1$ values and move the remaining $k^*$ to a new leaf node. Replicate last value of original node in parent node. (success)
  - The split can propagate up to the root and even create a new level

B*-Trees: Insert Operation (2)

2 4 6 7 22 24 26 90

Insert 5
2 4 5 6 7 22 24 26 90

Insert 25
2 4 5 6 7 22 24 25 26 90

B*-Trees: Delete Operation (1)

- Delete a value $p$:
  - Goal: delete the $p$ occurrence from the leaf node and any internal node
  - Find leaf node containing value (search operation)
  - Delete the value from leaf node
  - If the leaf remains half full, then success
  - If the leaf is not half full (underflow) then merge with left or right neighbour and redistribute so that both are at least half full. If not, merge to one leaf.
  - The underflow and merging can propagate up to the root and even reduce the number of the levels

B*-Trees: Delete Operation (2)

5 7

Insert 5
2 4 5 6 7 22 24 26 90

Insert 25
2 4 5 6 7 22 24 25 26 90

5 7

Delete 7
2 4 5 6 22 24 25 26 90

4 6 25

Delete 7
2 4 5 6 22 24 25 26 90
B*-Trees: Delete Operation (3)

Delete 5

(step 1)

2 4 6

22 24 25

26 90

(step 2)

6 25

2 4 6

22 24 25

26 90

References

- Bibliography of Introduction to Databases
- Course “Indexstrukturen für Datenbanken” (Professor Seidl)