Principles of Database Management Systems

4.1: B-Trees

Pekka Kilpeläinen
(after Stanford CS245 slide originals by Hector Garcia-Molina, Jeff Ullman and Jennifer Widom)

• B-Trees
  - a commonly used index structure
  - nonsequential, “balanced” (access paths to different records of equal length)
  - adapts well to insertions & deletions
  - consists of blocks holding at most n keys and n+1 pointers, and at least half of this
  - (We consider a variation actually called a B+ tree)

B+Tree Example

n=3

Root

Sample non-leaf

to keys to keys to keys to keys
< 120 120 ≤ k<150 150 ≤ k<180 ≥180

Sample leaf node:

Don’t want nodes to be too empty

• Number of pointers in use:
  - at internal nodes at least \( \lceil (n+1)/2 \rceil \)
    (to child nodes)
  - at leaves at least \( \lfloor (n+1)/2 \rfloor \)
    (to data records/blocks)

\[ x = \min \{ n \in \mathbb{Z} | n \geq x \} \]
\[ x = \max \{ n \in \mathbb{Z} | n \leq x \} \]
### B+tree rules

1. All leaves at the same lowest level (balanced tree)

2. Pointers in leaves point to records except for “sequence pointer”

### Insert into B+tree

First lookup the proper leaf;

(a) simple case
- leaf not full: just insert (key, pointer-to-record)
(b) leaf overflow
(c) non-leaf overflow
(d) new root

### Example

(a) Insert key = 32

(b) Insert key = 7
(c) Insert key = 160

(d) New root, insert 45

Deletion from B+tree

Again, first lookup the proper leaf;

(a): Simple case: no underflow; Otherwise ...
(b): Borrow keys from an adjacent sibling 
(if it doesn’t become too empty); Else ...
(c): Coalesce with a sibling node

(d): Cases (a), (b) or (c) at non-leaf

(b) Borrow keys
- Delete 50

(c) Coalesce with a sibling
- Delete 50

(d) Non-leaf coalesce
- Delete 37
B+tree deletions in practice
- Often, coalescing is not implemented
  - Too hard and not worth it!
  - Later insertions may return the node to its
    required minimum size
  - Compromise: Try redistributing keys with a sibling;
    if not possible, leave it there
  - If all accesses to the records go through the B-tree,
    can place a "tombstone" for the deleted record at
    the leaf

Why B-trees Are Good?
• B-tree adapts well to insertions and deletions, maintaining balance
  - DBA does not need to care about reorganizing
  - Split/merge operations rather rare
    (How often would nodes with 200 keys be split?)
  – Access times dominated by key-lookup
    (i.e., traversal from root to a leaf)

Efficiency of B-trees
• For example, assume 4 KB blocks, 4 byte
  keys and 8 byte pointers
• How many keys and pointers fit in a node
  (= index block)?
  - Max n s.t. (4*n + 8*(n+1)) ≤ 4096 B ?
    → n=340; 340 keys and 341 pointers fit in a node
    → 171 ... 341 pointers in a non-leaf node

Efficiency of B-trees (cont.)
• Assume an average node has 255 pointers
  → a three-level B-tree has $255^2 = 65025$ leaves with
    total of $255^1$ or about 16.6 million pointers to records
  → If root block kept in main memory, each
    record can be accessed with 2+1 disk I/Os;
    if all 256 internal nodes are in main memory, record
    access requires 1+1 disk I/Os
    (256 x 4 KB = 1 MB; quite feasible!)

Outline/summary
• B trees
  • popular index structures with graceful
    growth properties
  • support range queries like
    "... WHERE 100 < Key < 200"
    (→ Exercises)
• Next: Hashing schemes