Lecture 6

B+ Trees: Structure and Functions

Chapt. 10.3-10.8: Ramakrishnan & Gehrke

Demetris Zeinalipour
http://www.cs.ucy.ac.cy/~dzeina/courses/epl446
Lecture Outline
B+ Trees: Structure and Functions

• 10.3) Introduction to B+ Trees
• 10.4-10.6) B+Tree Functions: Search / Insert / Delete with Examples
• 10.7) B+ Trees in Practice.
  – Prefix-Key Compression (Προθεματική Συμπίεση Κλειδιών)
  – Bulk Loading B+Trees (Μαζική Εισαγωγή Δεδομένων)
Introduction to Tree Structures
(Εισαγωγή σε Δενδρικές Δομές)

• We will study two Tree-based structures:
  – **ISAM**: A *static* structure (does not *grow* or *shrink*).
    • Suitable when changes are infrequently;
    • Copes better with *Locking Protocols*
  – **B+ tree**: A *dynamic* data structure which adjusts efficiently under *inserts* and *deletes*.
    • Most widely used tree structure in DBMS systems!
    • Has similarly to ISAM, nodes with a high *fan-out* \( f \) (~133 children per node).
    • Similar to a Btree but different…
      – In a B+Tree, *data entries* are stored at the leaf level.
      – A Btree allows search-key values to appear only once;
        eliminates redundant storage of search keys (not suitable for DB apps where more index entries yield better search performance)
B+ Tree: Introductory Notes
(B+Tree: Εισαγωγικές Επισημάνσεις)

- Insert/delete at $\log_F N$ cost; keep tree balanced (ισοζυγισμένο). ($F =$ fanout, $N =$ # 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 (βαθμός του δένδρου).
- Supports equality and range-searches (αναζητήσεις ισότητας και διαστήματος) efficiently.

Index Entries
(Direct search)

Data Entries
("Sequence set")

d=2, f=3

\begin{tabular}{|c|c|}
\hline
22 & 34 \\
\hline
\end{tabular}
Example B+ Tree
(Παράδειγμα B+ Tree)

- Search begins at root, and key comparisons direct it to a leaf (as in ISAM).
- Search for 5*, 15*, all data entries >= 24* ...
- Based on the search for 15*, we know its not in the tree!
- Note that leaf pages (τερματικοί κόμβοι) are linked together in a doubly-linked list (as opposed to ISAM).
- That happens because ISAM nodes are allocated sequentially during Index construction time
  - consequently, no need to maintain the next prev-next-pointer.
B+ Trees in Practice
(B+Trees στην Πράξη)

• **Typical order (d):** 100 (i.e., 100 ≤ #children ≤ 200)

• **Typical fanout (f) = 133**
  – Typical fill-factor: 67% (133/200)

• **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 = $133^0 = 1$ page = 8 Kbytes
  – Level 2 = $133^1 = 133$ pages = ~1 MB (1064 KB)
  – Level 3 = $133^2 = 17,689$ pages = ~133 MB (141,512 KB)
B+ Tree Insertion Algorithm

1. **Find** correct leaf \( L \).
2. **Put** data entry onto \( L \).
   - If \( L \) has enough space, *done*!
   - Else *split* (διαμοίραση) \( L \) *(into \( L \) and a new node \( L_2 \))*
     - Redistribute (Ανακατένεμε) entries evenly between \( L \) and \( L_2 \), **copy up** (Αντιγραφή-Πρός-Τα-Πάνω) middle key.
     - Insert index entry pointing to \( L_2 \) into parent of \( L \).

\[
\begin{array}{cccc}
2^* & 3^* & 5^* & 7^*
\end{array}
\]

Assume we insert 8

**Copy up 5**: cannot just push-up 5 as every data entry needs to appear in a leaf node.

**Problem**: 5 won’t fit in parent of \( L_2 \). (see next slide)
B+ Tree Insertion Algorithm
(Αλγόριθμος Εισαγωγής στο B+Tree)

3. A parent needs to recursively **Push-Up** (Προώθηση-Προς-Πάνω) the **middle key** until the insertion is successful i.e.,
   - No need to **copy-up** as the latter will generate redundant index entries.
   - If Parent has enough space, done!
   - Else **split** (διαμοίραση) Parent
     - Redistribute (Ανακατένασμα) entries evenly, **push up** middle key.

4. Splits “grow” tree; root split increases **height** (ύψος)
   - Tree growth: gets **wider** or **one level taller at top**.

![Diagram](image)
Example B+ Tree After Inserting 8*

Root was split => That lead to increase in height from 1 to 2.

Minimum occupancy (d, i.e., 50%) is guaranteed in both leaf and index pages splits (for root page this constraint is relaxed)
  - Split occurs when adding 1 key to a node that is full (has 2d entries).
    Thus we will end up with two nodes, one with d and one with d+1 entries.
  - Can avoid split by re-distributing entries between siblings – (αδελφικοί κόμβοι); however, this is usually not done in practice. The borrowing practice is adopted only during deletions (see next).
B+ Tree Deletion Algorithm
(Αλγόριθμος Διαγραφής απο B+Tree)

• Start at root, **find leaf** \( L \) where entry belongs.
  – E.g., deleting 19 then 20

• **Remove the entry** \( K^* \) (not index entries).
  – If \( L \) is **at least half-full**, done! (e.g., after deleting 19*)
  – If \( L \) has only \( d-1 \) entries, (e.g., after deleting 20*)
    • Try to **re-distribute**, borrowing from *sibling* (adjacent node with same parent as \( L \)). (e.g., borrow 24* and update)
    • If re-distribution fails, **merge** \( L \) and sibling (see slide 12)

---

1) Borrow 24*
2) Copy-Up 27* to replace 24
3) Borrow 24*
B+ Tree Deletion Example

(Παράδειγμα Διαγραφής από B+Tree)

Delete

19, 20

Initial Tree

Final Tree
B+ Tree Deletion Algorithm
(Αλγόριθμος Διαγραφής από B+Tree)

- If re-distribution after delete fails then **merge L and sibling** (e.g., delete 24 => can’t borrow => merge)
- Now we also need to adjust **parent of L** (pointing to L or sibling). (i.e., delete 27)
- Merge could propagate to root, decreasing height.

```
| 2 3 | 5 7 8 | 14*16* | 22*24* | 27*29* | 33*34*38*39* |
```

```
| 30 |
| 33* | 34* | 38* | 39* |
```

Delete 24*

Merged {22} with {27, 29}
Merging propagates to sink
(Η Συγχώνευση διαδίδεται μέχρι τη ρίζα)

- But … occupancy Factor of L dropped below 50% (d=2) which is not acceptable.
- Thus, L needs to be either i) merged (συγχωνεύτει) with its sibling {5,13}
- or ii) redistributed (ανακατανεμηθεί) with its sibling (next slide)
Example of Non-leaf Re-distribution

(Παράδειγμα Ανακατανομής από Ρίζα)

- Let us assume the below tree.
  - Obviously the index node that contains 30 needs to be corrected as $d < 2$.

- In contrast to our previous example, we re-distribute entries from the left child of the root to right child.
After Re-distribution (Μετά την Ανακατανομή)

- Intuitively, entries are re-distributed by `pushing through` the splitting entry in the parent node.
- It suffices to re-distribute index entry with key 20; we’ve re-distributed 17 as well for illustration.
Prefix Key Compression
(Προθεματική Συμπίεση Κλειδιών)

• It is important to increase fan-out, as this allows to direct searches to the leaf level more efficiently.

• Index Entries are only to `direct traffic’, thus we can compress them.
  – E.g., “David Smith” could become “Davi” (as all entries on the left are smaller than “Davi”)
  – Consequently, we can fit more Index entries in Index pages!

![Diagram showing index entries and compressed key]

Davi
Summary of Bulk Loading (Μαζική Εισαγωγή Δεδομένων)

- **Scenario:** We want to construct a B+Tree on a pre-existing collection (υφιστάμενη συλλογή) of records.

- **Option 1:** multiple (individual) inserts.
  - Slow & Does not give sequential storage of leaves.

- **Option 2:** *Bulk Loading* (Μαζική Εισαγωγή).
  - **Idea:** Sort all data entries, insert pointer to first (leaf) page in a new (root).
  - **Effect:** Splits occur only on the right-most path from the root to leaves.
  - **Advantages:** i) Fewer I/Os during build and ii) Leaves will be stored sequentially (and linked, of course).

---

EPL446: Advanced Database Systems - Demetris Zeinalipour (University of Cyprus)
Bulk Loading with Example
(Μαζική Εισαγωγή με Παράδειγμα)

Main Idea of Bulk Loading:
Splits occur only on the right-most path from the root to the leaf level.
Bulk Loading with Example
(Mаζική Εισαγωγή με Παράδειγμα)

- **Index entries** always entered into **right-most index page** just above leaf level.

- When this **fills up**, it splits. (Split may go up **right-most path** to the root.)

- Much faster than repeated inserts, especially when one considers locking!
Summary
(Σύνοψη)

• **Tree-structured indexes** are ideal for **range-searches**, also good for **equality** searches.

• **ISAM** is a static structure.
  - Only **leaf pages** modified; **overflow** pages needed.
  - **Overflow chains** can degrade performance unless size of data set and data distribution stay constant.

• **B+ tree** is a dynamic structure.
  - Inserts/deletes leave tree balanced: \( \log_F N \) cost.
  - **High fanout** (\( F \)) means depth rarely more than 3 or 4.
  - Almost always **better** than maintaining a **sorted file**.
Summary
(Σύνοψη)

- Typically, 67% (e.g., 133/200) occupancy on average.
- Usually preferable to ISAM (minus locking considerations) as it adjusts to growth gracefully.
- If data entries are data records, splits can change rids!

- **Key compression** increases fanout, reduces height.
- Bulk loading can be much faster than repeated inserts for creating a B+ tree on a large data set.
- **Most widely used index** in database management systems because of its **versatility**. One of the most optimized components of a DBMS.