CS222: Principles of Data Management

Notes #06
B+ Trees

Instructor: Chen Li
B+ Tree: Most Widely Used Index!

- Insert/delete at $\log_F N$ cost; keep tree height-balanced. (F = fanout, N = # leaf pages)
- Minimum 50% occupancy (except for root). Each node contains $d \leq m \leq 2d$ entries. The (mythical) $d$ is called the order of the B+ tree.
- Supports equality and range-searches efficiently.
Example B+ Tree

- Search begins at root, and key comparisons direct the search to a leaf (as in ISAM).
- *Ex:* Search for 5*, 15*, all data entries >= 24*, ...

Based on the search for 15*, we know it is not in the tree!
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 level(s) 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 \( L \) (using a search).
- Put data entry onto \( L \).
  - If \( L \) has enough space, done! (Most likely case!)
  - Else, must split \( L \) (into \( L \) and a new node \( L_2 \))
    - Redistribute entries “evenly” and copy up \( L_2 \)’s low key.
    - Insert new index entry pointing to \( L_2 \) into parent of \( L \).

- This can happen recursively.
  - To split an index node, redistribute entries evenly but push up the middle key. (Contrast with leaf splits!)
- Splits “grow” tree; root split increases its height.
  - Tree growth: gets wider or one level taller at top.
Inserting $8^*$ into Example B+ Tree

- Observe how minimum occupancy is guaranteed in both leaf and index pg splits.
- Note difference between copy-up and push-up; be sure you understand the reasons for this!
Example B+ Tree After Inserting 8*

- Notice that root was split, leading to increase in height.
- In this example, could avoid split by redistributing entries; however, not usually done in practice. (Q: Why is that?)
Deleting a Data Entry from a B+ Tree

- Start at root, find leaf $L$ where entry belongs.
- Remove the entry.
  - If $L$ is still at least half-full, done!
  - If $L$ has only $d-1$ entries,
    - Try to redistribute, borrowing from sibling (adjacent node with same parent as $L$).
    - If re-distribution fails, merge $L$ and sibling.
- If merge occurred, must delete search-guiding entry (pointing to $L$ or sibling) from parent of $L$.
- Merge could propagate to root, decreasing height.
Example Tree After (Inserting 8*, Then) Deleting 19* and 20* ... 

- Deleting 19* is easy.
- Deleting 20* is done with redistribution. Notice how new middle key is copied up.
... And Then Deleting 24*

- Must merge.
- Observe “toss” of index entry (on right), and “pull down” of index entry (below).
Example of Non-leaf Redistribution

- New/different example B+ tree is shown below during deletion of 24*
- In contrast to previous example, can redistribute entry from left child of root to right child.
After Redistribution

- Intuitively, entries are redistributed by "pushing through" the splitting entry in the parent node.
Prefix Key Compression

- It’s important to increase tree fan-out. (Why?)
- Key values in index entries only `direct traffic’; we can often compress them.
  - E.g., If we have adjacent index entries with search key values *Dannon Yogurt, David Smith* and *Devarakonda Murthy*, we can abbreviate *David Smith* to *Dav*. (The other keys can be compressed too ...)
    - Is this correct? Not quite! What if there is a data entry *Davey Jones*? (Then can only compress *David Smith* to *Davi*)
    - In general, while compressing, must leave each index entry greater than every key value (in any subtree) to its left.

- Insert/delete logic must be suitably modified.
If we have a large collection of records, and we want to create a B+ tree on some field, doing so by repeatedly inserting records is very slow.

**Bulk Loading** can be done much more efficiently!

**Initialization:** Sort all data entries, insert pointer to first (leaf) page in a new (root) page.
**Bulk Loading (Contd.)**

- Index entries for leaf pages 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 of Bulk Loading

- Option 1: multiple inserts.
  - Slow.
  - Does not give storage layout of leaves.
  - Also leaves a wake of half-filled pages.

- Option 2: **Bulk Loading**
  - Has advantages for concurrency control.
  - Fewer I/Os during build.
  - Leaves will be stored sequentially (and linked).
  - Can control “fill factor” on pages.
  - Can optimize non-leaf splits more than shown.
A Note on B+ Tree “Order”

- (Mythical) order ($d$) concept replaced by physical space criterion in practice (“at least half-full”).
  - Index pages can typically hold many more entries than leaf pages (RIDs vs. PID).
  - Variable sized records and search keys mean different nodes will contain different numbers of entries.
  - Even with fixed length fields, multiple records with the same search key value (duplicates) can lead to variable-sized data entries (if we use Alternative (3)).
Summary

- 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 height-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 (Cont’d.)

- Typically, 67% occupancy on average.
- Generally preferable to ISAM, modulo locking considerations; 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. Also one of the most heavily optimized components of a DBMS.