Tree-Structured Indexes

Chapter 9

Introduction

- As for any index, 3 alternatives for data entries $k^*$:
  - A data record with key value $k$
  - A $<k$ rid of data record with search key value $k$
  - A $<k$ list of rids of data records with search key $k$
- Choice is orthogonal to the indexing technique used to locate data entries $k^*$.
- Tree-structured indexing techniques support both range searches and equality searches.
- ISAM: static structure; B+ tree: dynamic, adjusts gracefully under inserts and deletes.

Range Searches

- "Find all students with gpa > 3.0"
  - If data is in sorted file, do binary search to find first such student, then scan to find others.
  - Cost of binary search can be quite high.
- Simple idea: Create an 'index' file.
  - Can do binary search on (smaller) index file!

ISAM

- Index file may still be quite large. But we can apply the idea repeatedly!

Comments on ISAM

- File creation: Leaf (data) pages allocated sequentially, stored by search key, then index pages allocated, then space for overflow pages.
- Index entries: <search key value, page id>; they 'direct' search for data entries, which are in leaf pages.
- Search: Start at root; use key comparisons to go to leaf. Cost $\propto \log N$; $F = #\text{entries/index pg}$, $N = #\text{leaf pgs}$
- Insert: Find leaf data entry belongs to, and put it there.
- Delete: Find and remove from leaf; if empty overflow page, deallocate.
- Static tree structure: inserts/deletes affect only leaf pages.

Example ISAM Tree

- Each node can hold 2 entries; no need for 'next-leaf-page' pointers. (Why?)
**B+ Tree: The Most Widely Used Index**

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

**Example B+ Tree**

- Search begins at root, and key comparisons direct it to a leaf (as in ISAM).
- Search for \( 5^*, 15^* \), all data entries \( \geq 24^* \) ...

**B+ Trees in Practice**

- Typical order: 100. Typical fill-factor: 67%.
  - Average fanout = 1.33
- Typical capacities:
  - Height 4: 133^3 = 2,915,007,600 records
  - Height 5: 133^4 = 2,392,637 records
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,669 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 \( L_1 \) and a new node \( L_2 \))
      - Redistribute entries evenly, if possible,
      - Insert index entry pointing to \( L_2 \) into parent of \( L \).
  - This can happen recursively.
    - To split index node, redistribute entries evenly, but push up middle key. (Contrast with leaf splits.)
  - Split reduces tree; root split increases height.
    - Tree growth gets \( \log \) \( N \) after \( \log \) \( N \) top.
**Inserting 8* into Example B+ Tree**

- Observe how minimum occupancy is guaranteed in both leaf and index page 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, we can avoid split by re-distributing entries; however, this is usually not done in practice.

---

**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.

**Example Tree After (Inserting 8*, Then) Deleting 19* and 20***

- Deleting 19* is easy.
- Deleting 20* is done with re-distribution. Notice how middle key is copied up.

---

**... And Then Deleting 24***

- Must merge.
- Observe 'loss' of index entry (on right), and 'pull down' of index entry (below).

**Example of Non-leaf Re-distribution**

- Tree is shown below during deletion of 24*. (What could be a possible initial tree?)
- In contrast to previous example, can re-distribute entry from left child of root to right child.
**After Re-distribution**

- 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**

- Important to increase fan-out. (Why?)
- Key values in index entries only 'direct traffic'; can often compress them.
  - E.g., if we have adjacent index entries with search key values Demon Yogurt, David Smith and Devendra Murthy, we can abbreviate David Smith to D. (The other keys can be compressed too ...)
  - Is this correct? Not-quite. What if there is a data entry named Jones? (Can't do David Smith to D.)
  - In general, while compressing, must leave each index entry greater than every key value (in any subtrees) to its left.
- Insert/delete must be suitably modified.

**Bulk Loading of a B+ Tree**

- If we have a large collection of records, and we want to create a B+ tree on such data, 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 sequential storage of leaves.
- **Option 2**: **Bulk Loading**
  - Has advantages for concurrency control.
  - Fewer I/Os during build.
  - Leaves will be stored sequentially (and linked, of course).
  - Can control "fill factor" on pages.

**A Note on 'Order'**

- **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.
  - 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 (duplicate) 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; \( \log N \) cost.
  - High fanout (F) means depth rarely more than 3 or 4.
  - Almost always better than maintaining a sorted file.

Summary (Contd.)

- Typically, 67% occupancy on average.
- Usually preferable to ISAM, modulo locking considerations; adjusts to growth gracefully.
- If data entries are data records, splits can change ids!
- 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.