Tree-Structured Indexes

Yanlei Diao
UMass Amherst
Feb 20, 2007

Introduction

- As for any index, 3 alternatives for data entries \( k^* \):
  - Data record with key value \( k \)
  - \(<k, \text{rid of data record with search key value } k>\)
  - \(<k, \text{list of rids of data records with search key } k>\)
- Choice is orthogonal to the indexing technique used to locate data entries \( k^* \).
  - Tree structured indexing: ISAM, B⁺tree, R-tree, …
  - Hash indexing: linear hashing, extensible hashing, …

Tree Structured Indexes

- Tree-structured indexing techniques support both range searches and equality searches.
- Tree structures with search keys on value-based domains
  - ISAM: static structure
  - B⁺ tree: dynamic, adjusts gracefully under inserts and deletes.
- Tree structures with the search key on multi-dimensional objects
  - R-tree, R⁺-tree, discussed later
ISAM

- Leaf pages contain sorted data records (e.g., Alt 1 index).
- Non-leaf part directs searches to the data records; static once built!
- Inserts/deletes: use overflow pages, bad for frequent inserts.

Comments on ISAM

- Main problem
  - Long overflow chains after many inserts, high I/O cost for retrieval.
- Advantages
  - Simple when updates are rare.
  - Leaf pages are allocated in sequence, leading to sequential I/O.
  - Non-leaf pages are static; for concurrent access, no need to lock non-leaf pages!
- Good performance for frequent updates? B+tree!

B+ Tree: Most Widely Used Index

- Height-balanced given arbitrary inserts/deletes.
  - \( F = \text{fanout}, \, N = \# \text{leaf pages}, \, H = \log_F N \).
- Minimum 50% occupancy (except for root).
  - Each non-root node contains \([n/2], n\) entries, where \(n\) is the max \# of keys in a node, called order of the tree.
  - Root node can have \([1, n]\) entries.
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 >= 24* ...

---

B+ Trees in Practice

- Typical order: 200. 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 levels 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 \(L\).
- Put data entry onto \(L\).
  - If \(L\) has enough space, done!
  - Else, must split \(L\) (into \(L\) and a new node \(L2\))
    - Redistribute entries evenly, copy up middle key.
    - Insert index entry pointing to \(L2\) into parent of \(L\).
- This can happen recursively
  - To split index node, redistribute entries evenly, but push up middle key. (Contrast with leaf splits.)
- Splits “grow” tree; root split increases height.
  - Tree growth: gets wider or one level taller at top.
**Previous Example**

![Inserting 8*]

**Inserting 8* into Example B+ Tree**

- Minimum occupancy is guaranteed in both leaf and index pg splits.
- Note difference between **copy-up** and **push-up**.
- Reasons for this?

![Entry to be inserted in parent node. Note that 5 is copied up and continues to appear in the root.]

![Entry to be inserted in parent node. Note that 17 is pushed up and only appears once in the node. Compare this with a leaf split.]

**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 between siblings; but not usually 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 $\lceil n/2 \rceil - 1$ 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 entry (pointing to $L$ or sibling) from parent of $L$.
- Merge could propagate to root, decreasing height.

Current B+ Tree

Example Tree After Deleting 19* and 20*

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

... And Then Deleting 24*

Example of Non-leaf Re-distribution

- Must merge.
- Must merge.
- Pull down of index entry (below).
- Pull down of index entry (below).

- 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., adjacent index entries with search key values [Dave Jones, David Smith and Devarakonda Murthy]
  - Can we abbreviate David Smith to Dav?
    - *Not correct!* Can only compress David Smith to Dav.
    - In general, while compressing, must leave each index entry greater than every key value (in any subtree) to its left.
- Insert/delete must be suitably modified.

Bulk Loading of a B+ Tree

- Have a large collection of records, and want to create a B+ tree on some field. Doing so by repeatedly inserting records?
  - Slow due to repeated traversals and splits
  - Significant locking overhead.
  - Not necessarily the optimal structure. An example?
  - Low storage utility. An example?
- **Bulk Loading** can be done much more efficiently!
**Bulk Loading Algorithm**

- **Initialization**:
  - Sort all data entries
  - Insert pointer to the first (leaf) page in a new (root) page.

**Bulk Loading Algorithm (Contd.)**

- Index entries for leaf pages always enter into r*, right-most index page just above leaf level.
- When the r* node fills up, it splits.
- Split may go up right-most path to the root.

**Summary of Bulk Loading**

- **Option 1**: multiple inserts.
  - Slow due to I/O cost and locking overhead.
  - No give sequential storage of leaves.
  - Sometimes low storage utility.
- **Option 2**: Bulk Loading
  - 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 (n) 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 (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; \( \log_2 N \) 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 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.