Lecture #07: Tree Indexes I

15-445/645 Database Systems (Fall 2019)
https://15445.courses.cs.cmu.edu/fall2019/
Carnegie Mellon University
Prof. Andy Pavlo

1 Indexes

This lecture continues our discussion of data structures. We are going to focus on table indexes.

A *table index* is a replica of a subset of a table's columns that is organized in such a way that allows the DBMS to find tuples more quickly than performing a sequential scan. The DBMS ensures that the contents of the tables and the indexes are always in sync.

It is the DBMS's job to figure out the best indexes to use to execute queries. There is a trade-off on the number of indexes to create per database (indexes use storage and require maintenance).

2 B+Tree

A **B+Tree** is a self-balancing tree data structure that keeps data sorted and allows searches, sequential access, insertion, and deletions in $O(log(n))$. It is optimized for disk-oriented DBMSs that read/write large blocks of data.

Almost every modern DBMS that supports order-preserving indexes uses a B+Tree. There is a specific data structure called a **B-Tree**, but people also use the term to generally refer to a class of data structures. Modern B+Tree implementations combine features from other B-Tree variants, such as the sibling pointers used in the B^{link}-Tree.

Formally, a B+Tree is an M-way search tree with the following properties:

- It is perfectly balanced (i.e., every leaf node is at the same depth).
- Every inner node other than the root is at least half full ($M/2 - 1 <=$ num of keys $<= M - 1$).
- Every inner node with k keys has k+1 non-null children.

Every node in a B+Tree contains an array of key/value pairs:

- Arrays at every node are (almost) sorted by the keys.
- The value array for inner nodes will contain pointers to other nodes.
- Two approaches for leaf node values
  1. Record IDs: A pointer to the location of the tuple
  2. Tuple Data: The actual contents of the tuple is stored in the leaf node

2.1 Insertion

1. Find correct leaf L.
2. Add new entry into L in sorted order:
   - If L has enough space, the operation done.
   - Otherwise split L into two nodes L and L_2. Redistribute entries evenly and copy up middle key. Insert index entry pointing to L_2 into parent of L.
3. To split an inner node, redistribute entries evenly, but push up the middle key.

2.2 Deletion

1. Find correct leaf L.
2. Remove the entry:
   - If L is at least half full, the operation is done.
   - Otherwise, you can try to redistribute, borrowing from sibling.
   - If redistribution fails, merge L and sibling.
3. If merge occurred, you must delete entry in parent pointing to L.

3 B+Tree Design Decisions

Node Size:

- The optimal node size for a B+Tree depends on the speed of the disk. The idea is to amortize the cost of reading a node from disk into memory over as many key/value pairs as possible.
- The slower the disk, then the larger the idea node size.
- Some workloads may be more scan-heavy versus having more single key look-ups.

Merge Threshold:

- Some DBMS do not always merge when it's half full.
- Delaying a merge operation may reduce the amount of reorganization.
- It may be better for the DBMS to let underflows to occur and then periodically rebuild the entire tree to re-balance it.

Variable Length keys:

- Pointers: Store keys as pointers to the tuples attribute (very rarely used).
- Variable-length Nodes: The size of each node in the B+Tree can vary, but requires careful memory management. This approach is also rare.
- Key Map: Embed an array of pointers that map to the key+value list within the node. This is similar to slotted pages discussed before. This is the most common approach.

Non-Unique Indexes:

- Duplicate Keys: Use the same leaf node layout but store duplicate keys multiple times.
- Value Lists: Store each key only once and maintain a linked list of unique values.

Intra-Node Search:

- Linear: Scan the key/value entries in the node from beginning to end. Stop when you find the key that you are looking for. This does not require the key/value entries to be pre-sorted.
- Binary: Jump to the middle key, and then pivot left/right depending on whether that middle key is less than or greater than the search key. This requires the key/value entries to be pre-sorted.
- Interpolation: Approximate the starting location of the search key based on the known low/high key values in the node. Then perform linear scan from that location. This requires the key/value entries to be pre-sorted.

4 B+Tree Optimizations

Prefix Compression:

- Sorted keys in the same leaf node are likely to have the same prefix.
- Instead of storing the entire key each time, extract common prefix and store only unique suffix for each key.

Suffix Truncation:

- The keys in the inner nodes are only used to "direct traffic", we do not need the entire key.
- Store a minimum prefix that is needed to correctly route probes into the index.

Bulk Inserts:

- The fastest way to build a B+Tree from scratch is to first sort the keys and then build the index from the bottom up.
- This will be faster than inserting one-by-one since there are no splits or merges.