Lecture 09: Tree Based Indexes

15-445/645 Database Systems (Fall 2017)
Carnegie Mellon University
Prof. Andy Pavlo

Indexes

1. A table index is a replica of a subset of a table's columns
2. The DBMS ensures that the contents of the tables and the indexes are always in sync
3. It is the DBMS's job to figure out the best indexes to use to execute queries
4. There is a trade-off on the number of indexes to create per database (indexes use storage and require maintenance)

B+Tree

1. 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
2. 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))
   (a) Nodes can have more than two children
   (b) Great for sequential access
3. A B+Tree is an M-way search tree with eh following properties
   (a) Perfectly balanced (every leaf node is at the same depth)
   (b) Every inner node other than the root is at least half full ($M/2 - 1 <=$ num of keys $<= M - 1$)
   (c) Every inner node with k keys has k+1 non-null children
4. B+Tree Nodes: Every node in a B+ tree contains an array of key/value pairs
   (a) Arrays at every node are always sorted
   (b) The keys will always be the column or columns that you built your index on
   (c) Values will differ is node is inner or leaf
   (d) Two approaches for leaf node values
      i. Record IDs: A pointer to the location of the tuple
      ii. Tuple Data: The actual contents of the tuple is stored in the leaf node

B+Tree Operations

Inserts:

1. Find correct leaf L
2. Put data entry into L in sorted order
   (a) If L has enough space, done!
   (b) Else 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

Deletes:

1. Find correct leaf L
2. Remove the Entry
   (a) If L is at least half full, done!
   (b) Else, you can try to redistribute, borrowing from sibling
   (c) If redistribution fails, merge L and sibling
3. If merge occurred, you must delete entry in parent pointing to L

Bulk Inserts:

1. The fastest/best way to build a B+Tree is to first sort the keys and then build the index from the bottom up
2. Faster than inserting one by one

B+Tree Design Decisions

Merge Threshold:

1. Some DBMS do not always merge when it's half full
2. Delaying a merge operation may reduce the amount of reorganization

Non-Unique Indexes:

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

Variable Length keys:

1. Pointers: store keys as pointers to the tuples attribute (very rarely used)
2. Variable length nodes: The size of each node in the B+Tree can vary, but requires careful memory management
3. Key Map: Embed an array of pointers that map to the key+value list within the node (most common approach)

Prefix compression:

1. The keys in the inner nodes are only used to ”direct traffic”, we don’t need the entire key
2. Store a minimum prefix that is needed to correctly route probes into the index

Skip List

1. A linked list with multiple levels of extra points that skip over intermediate nodes
2. In general, a level has half the keys of the level below it
3. To Insert a new key, flip a coin to decide how many levels to add the new key into
4. Provides approx. O(log(n)) search
5. To Delete, first **logically** remove a key from the index by setting a flag to tell threads to ignore it, and then **physically** remove the key once we know that no other thread is holding the reference
6. Advantages over B+ Tree
   (a) Uses less memory than B+Tree
   (b) Insertions and deletions do not require re-balancing
7. Disadvantages to B+ Tree
   (a) Not disk/cache friendly because they do not optimize locality
   (b) Invoking a random number generator multiple times is slow
   (c) Reverse search is non-trivial

Radix Tree

1. Uses digital representation of keys to examine prefixes one-by-one instead of comparing entire key
2. Height of tree depends on the length of keys
3. Does not require re-balancing
4. The path to a leaf nodes represents the key of the leaf
5. Differs from a trie in that there is not a node for each element in key, nodes are consolidated to represent the largest prefix before keys differ
6. Not all attribute types can be decomposed into binary comparable digits for a radix tree

Index Optimizations

1. Partial indexes: Create an index on a subset of the entire table. Potentially reduces size and the amount of overhead to maintain it

2. Covering index: All attributes needed to process the query are available in an index, then the DBMS does not need to retrieve the tuple

3. Index Include columns: Embed additional columns in index to support index-only queries