Lecture #08: Indexes & Filters I

15-445/645 Database Systems (Fall 2024)
`https://15445.courses.cs.cmu.edu/fall2024/`
Carnegie Mellon University
Andy Pavlo

1 Table Indexes

There are a several data structures choices in a DBMS for purposes such as internal meta-data, core data storage, temporary data structures, or table indexes. Our focus in this lecture will be on table indexes. In the previous lecture we considered hash tables, which is not always the best option for table indexes since they cannot support range scans nor partial key look-ups.

A *table index* is a replica of a subset of a table's columns that is organized and/or sorted for efficient access using a subset of those attributes. So instead of performing a sequential scan, the DBMS can perform a lookup on the table index to find certain tuples more quickly. The DBMS ensures that the contents of the tables and the indexes are always logically in sync.

There exists a trade-off between the number of indexes to create per database. Although more indexes makes looking up queries faster, indexes also use storage and require maintenance. Plus, there are concurrency concerns with respect to keeping them in sync. It is the DBMS's job to figure out the best indexes to use to execute queries.

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 DBMS's 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. The primary difference between the original *B-Tree* and the B+Tree is that B-Trees stores keys and values in *all nodes*, while B+ trees store values *only in leaf nodes*. 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 (where M represents the maximum number of children a node can have) 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.

For leaf nodes, the keys are derived from the attribute(s) that the index is based on. Though it is not necessary according to the definition of the B+Tree, arrays at every node are almost always sorted by the keys. Two approaches for leaf node values are *record IDs* and *tuple data*. Record IDs refer to a pointer to the location of the tuple, usually the primary key. Leaf nodes that have tuple data store the the actual contents of the tuple in each node.
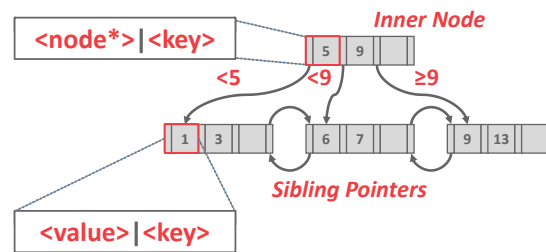
Figure 1: B+ Tree diagram

For inner nodes, the values contain pointers to other nodes, and the keys can be thought of as guide posts. They guide the tree traversal but do not represent the keys (and hence their values) on the leaf nodes. What this means is that you could potentially have a key in an inner node (as a guide post) that is not found on the leaf nodes. Although it must be noted that conventionally inner nodes posses only those keys that are present in the leaf nodes.

Depending on the index specification, the DBMS will cluster null keys in either the first leaf node (i.e., NULL FIRST) or the last leaf node (i.e., NULL LAST).

Insertion

To insert a new entry into a B+Tree, one must traverse down the tree and use the inner nodes to figure out which leaf node to insert the key into.

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

Deletion

Whereas in inserts we occasionally had to split leaves when the tree got too full, if a deletion causes a tree to be less than half-full, we must merge in order to re-balance the tree.

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 the sibling.
    - If redistribution fails, merge L and the sibling.
3. If a merge occurred, you must delete the entry in the parent pointing to L.

Composite Index

The key is composed of multiple attributes. We can create composite index like:

```
CREATE INDEX abc_index ON table (a, b, c);
```

Then we can use the index for queries like:

```
SELECT a, b, c FROM table
WHERE a = 1 AND b = 2 AND c = 3;
```

Note that the 'AND' operator is mostly necessary. Conditions with 'OR' are generally not supported.

Selection Conditions

Because B+Trees are in sorted order, look ups have fast traversal and also do not require the entire key. The DBMS can use a B+Tree index if the query provides any of the attributes of the search key. This differs from a hash index, which requires all attributes in the search key.
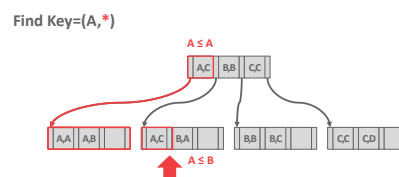
Figure 2: To perform a prefix search on a B+Tree, one looks at the first attribute on the key, follows the path down and performs a sequential scan across the leaves to find all they keys that one wants.

Duplicate Keys

There are two approaches to duplicate keys in a B+Tree.

The first approach is to *append record ID*s as part of the key. Since each tuple's record ID is unique, this will ensure that all the keys are identifiable.

The second approach is to allow leaf nodes to spill into *overflow nodes* that contain the duplicate keys. Although no redundant information is stored, this approach is more complex to maintain and modify.

Clustered Indexes

The table is stored in the sort order specified by the primary key, as either heap- or index-organized storage. Since some DBMSs always use a clustered index, they will automatically make a hidden row id primary key if a table doesn't have an explicit one, but others cannot use them at all.

Index Scan Page Sorting

Since directly retrieving tuples from an unclustered index is inefficient, the DBMS can first figure out all the tuples that it needs and then sort them based on their page id. This way, each page will only need to be fetched exactly once.

3   B+Tree Design Choices

3.1   Node Size

Depending on the storage medium, we may prefer larger or smaller node sizes. For example, nodes stored on hard drives are usually in the order of megabytes in size to reduce the number of seeks needed to find data and amortize the expensive disk read over a large chunk of data, while in-memory databases may use page sizes as small as 512 bytes in order to fit the entire page into the CPU cache as well as to decrease data fragmentation. This choice can also be dependent on the type of workload, as point queries would prefer as small a page as possible to reduce the amount of unnecessary extra info loaded, while a large sequential scan might prefer large pages to reduce the number of fetches it needs to do.

3.2   Merge Threshold

While B+Trees have a rule about merging underflowed nodes after a delete, sometimes it may be beneficial to temporarily violate the rule to reduce the number of deletion operations. For instance, eager merging could lead to thrashing, where a lot of successive delete and insert operations lead to constant splits and merges. It also allows for batched merging where multiple merge operations happen all at once, reducing the amount of time that expensive write latches have to be taken on the tree.

There are merge strategy that keeps small nodes in the tree and rebuilds it later, which made the tree unbalanced (as in Postgres). We will not discuss this in the lecture.

3.3   Variable Length Keys

Currently we have only discussed B+Trees with fixed length keys. However we may also want to support variable length keys, such as the case where a small subset of large keys lead to a lot of wasted space. There are several approaches to this:

1. **Pointers**
   Instead of storing the keys directly, we could just store a pointer to the key. Due to the inefficiency of having to chase a pointer for each key, the only place that uses this method in production is embedded devices, where its tiny registers and cache may benefit from such space savings.
2. **Variable Length Nodes**
   We could also still store the keys like normal and allow for variable length nodes. This is generally infeasible and largely not used due to the significant memory management overhead of dealing with variable length nodes.
3. **Padding**
   Instead of varying the key size, we could set each key's size to the size of the maximum key and pad out all the shorter keys. In most cases this is a massive waste of memory, so you don't see this used by anyone either.
4. **Key Map/Indirection**
   The method that nearly everyone uses is replacing the keys with an index to the key-value pair in a separate dictionary. This offers significant space savings and potentially shortcuts point queries (since the key-value pair the index points to is the exact same as the one pointed to by leaf nodes). Some databases (e.g. PostgreSQL) allow overflow within a given node to maintain a fixed number of keys by storing excess data in overflow pages.

3.4   Intra-Node Search

Once we reach a node, we still need to search within the node (either to find the next node from an inner node, or to find our key value in a leaf node). While this is relatively simple, there are still some tradeoffs

to consider:

1. **Linear**
   The simplest solution is to just scan every key in the node until we find our key. On the one hand, we don't have to worry about sorting the keys, making insertions and deletes much quicker. On the other hand, this is relatively inefficient and has a complexity of $\mathbb{O}(n)$ per search. This can be vectorized using SIMD (or equivalent) instructions.

2. **Binary**
   A more efficient solution for searching would be to keep each node sorted and use binary search to find the key. This is as simple as jumping to the middle of a node and pivoting left or right depending on the comparison between the keys. Searches are much more efficient this way, as this method only has the complexity of $\mathbb{O}(\ln(n))$ per search. However, insertions become more expensive as we must maintain the sort of each node.

3. **Interpolation**
   Finally, in some circumstances we may be able to utilize interpolation to find the key. This method takes advantage of any metadata stored about the node (such as max element, min element, average, etc.) and uses it to generate an approximate location of the key. For example, if we are looking for 8 in a node and we know that 10 is the max key and $10 - (n + 1)$ is the smallest key (where n is the number of keys in each node), then we know to start searching 2 slots down from the max key, as the key one slot away from the max key must be 9 in this case. Despite being the fastest method we have given, this method is only seen in academic databases due to its limited applicability to keys with certain properties (like integers) and complexity.

4   Optimizations

4.1   Prefix Compression

Most of the time when we have keys in the same node there will be some partial overlap of some prefix of each key (as similar keys will end up right next to each other in a sorted B+Tree). Instead of storing this prefix as part of each key multiple times, we can simply store the prefix once at the beginning of the node and then only include the unique sections of each key in each slot.
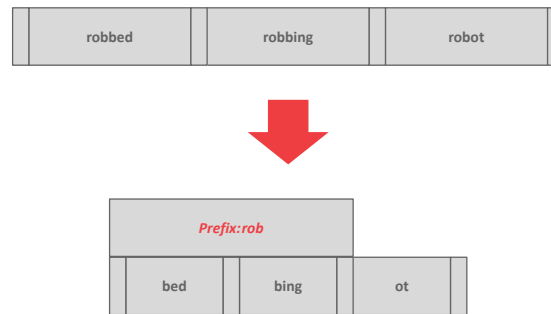
Figure 3: An example of prefix compression. Since the keys are in lexicographic order, they are likely to share some prefix.

4.2   Deduplication

In the case of an index which allows non-unique keys, we may end up with leaf nodes containing the same key over and over with different values attached. One optimization of this could be only writing the key once and then following it with all of its associated values.

4.3   Suffix Truncation

For the most part the key entries in inner nodes are just used as signposts and not for their actual key value (as even if a key exists in the index we still have to search to the bottom to ensure that it hasn't been deleted). We can take advantage of this by only storing the minimum prefix that is needed to correctly route probes into the correct node.

4.4   Pointer Swizzling

Because each node of a B+Tree is stored in a page from the buffer pool, each time we load a new page we need to fetch it from the buffer pool, requiring latching and lookups. To skip this step entirely, we could store the actual raw pointers in place of the page IDs (known as "swizzling"), preventing a buffer pool fetch entirely. Rather than manually fetching the entire tree and placing the pointers manually, we can simply store the resulting pointer from a page lookup when traversing the index normally. Note that we must track which pointers are swizzled and deswizzle them back to page ids when the page they point to is unpinned and victimized.

4.5   Bulk Insert

When a B+Tree is initially built, having to insert each key the usual way would lead to constant split operations. Since we already give leaves sibling pointers, initial insertion of data is much more efficient if we construct a sorted linked list of leaf nodes and then easily build the index from the bottom up using the first key from each leaf node. Note that depending on our context we may wish to pack the leaves

as tightly as possible to save space or leave space in each leaf to allow for more inserts before a split is necessary.

4.6 Write-Optimized B+ Tree

Split / merge node operation are expensive. Therefore, some variants of B-Tree, such as Bϵ-Tree, logs changes in the internal node and lazily propagates the updates down to the leaf node later.