# Lecture #08: Tree Indexes II

**15-445/645 Database Systems (Fall 2020)**
https://15445.courses.cs.cmu.edu/fall2020/
Carnegie Mellon University
Prof. Andy Pavlo

## 1 B+Tree Design Choices

### 1.1 Node Size

Depending on the storage medium, we may prefer larger or smaller node sizes. For example, nodes stored on hard drives are usually on 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.

### 1.2 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 operation. 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.

### 1.3 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 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). Due to the small size of the dictionary index value, there is enough space to place a prefix of each key alongside the index, potentially allowing some index searching and leaf scanning to not even have to chase the pointer (if the prefix is at all different from the search key).

### 1.4 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.
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.

## 2 Optimizations

### 2.1 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 eachother 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.

### 2.2 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.

### 2.3 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 also take advantage of this by only storing the minimum differentiating prefix of each key at a given inner node. While we can make this as minimal as just a single differing character/digit, it may be beneficial to leave a few redundant digits on the end to make it less likely that an indeterminable insertion due to an identical prefix will occur. In that instance, we would have to search to the bottom of the tree to find the full key (or in the case that that key was deleted, the minimum or maximum key of the leaf node depending).

### 2.4 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.

### 2.5 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.

## 3 Additional Index Usage

Indices are a way to provide fast access to data items. There are a number of indexing methods and tools that databases can use to improve performance.

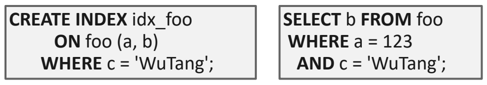
### Implicit Indexes

If you create a primary key or unique constraint, most DBMS’s will automatically create an *implicit index* to enforce integrity constraints but not referential constraints.

### Partial Indexes

In many situations, a user may not need an index for every tuple in the table and may only be interested in a subset of the data. A *partial index* is an index that has some condition or ”where clause” applied to it so that it includes a subset of the table. This potentially reduces the size and the amount of overhead to maintain the table and avoids polluting the buffer pool cache with unnecessary data. For instance, a common use of partial indexes is to partition indexes by date range (per month, year, etc.).

**Figure 1:** Partial Index example



```
CREATE INDEX idx_foo
     ON foo (a, b)
   WHERE c = 'WuTang';
```

```
SELECT b FROM foo
WHERE a = 123
  AND c = 'WuTang';
```

### Covering Indexes

If all attributes needed to process the query are available in an index, then the DBMS does not need to retrieve the tuple. The DBMS can complete the entire query just based on the data available in the index. In other words, *covering indexes* are only used to locate data records in the table and not to return data. This reduces contention on the DBMS's buffer pool resources.

**Figure 2:** Covering Index example

```sql
CREATE INDEX idx_foo
    ON foo (a, b);
```

```sql
SELECT b FROM foo
 WHERE a = 123;
```

### Index Include Columns

*Index include columns* allow users to embed additional columns in index to support index-only queries. These extra columns are stored in the leaf nodes and are not actually part of the search key.

**Figure 3:** Index Include Column example

```sql
CREATE INDEX idx_foo
            ON foo (a, b)
   INCLUDE (c);
```

```sql
SELECT b FROM foo
WHERE a = 123
  AND c = 'WuTang';
```

### Function/Expression Indexes

Indexes can also be created on functions or expressions. An index does not need to store keys in the same way that they appear in their base table. Instead, *function/expression indexes* store the output of a function or expression as the key instead of the original value. It is the DBMS's job to recognize which queries can use that index.

**Figure 4:** Function/Expression Index example

```sql
SELECT * FROM users
 WHERE EXTRACT(dow     ← day of week
            FROM login) = 2;
```

```sql
CREATE INDEX idx_user_login
   ON users (EXTRACT(dow FROM login));
```

## 4 Trie Index

Observe that the inner node keys in a B+Tree cannot tell you whether or not a key exists in the index. Every time you lookup a key, you must traverse to the leaf node.

A *Trie Index* allows us to know at the top of the tree whether or not a key exists. A trie index uses a digital representation of keys to examine prefixes one-by-one instead of comparing the entire key.

Trie indexes have a number of useful properties. For one, their shape only depends on key space and length and does not require rebalancing operations. Additionally, all operations have $O(k)$ complexity where $k$ is the length of the key. This is because the path to a leaf node represents the key of the leaf. The span of a trie level is the number of bits that each partial key/digit represents.

## 5 Radix Tree

A *radix tree* is a variant of a trie data structure. It is different from a trie index in that nodes with only a single child are omitted. Rather, nodes are consolidated to represent the largest prefix before keys differ.

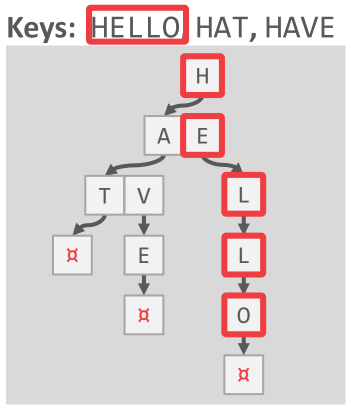


**Figure 5:** Trie Index diagram

Radix trees can produce false positives, so the DBMS must always check the original tuple to see whether a key matches.

The height of radix tree depends on the length of keys and not the number of keys like in a B+Tree. The path to a leaf nodes represents the key of the leaf. Not all attribute types can be decomposed into binary comparable digits for a radix tree.

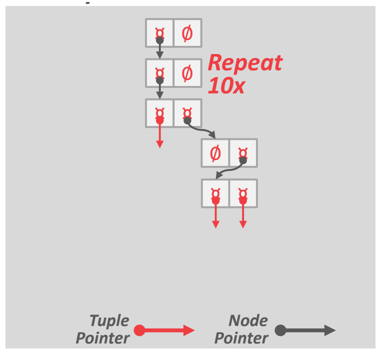


**Figure 6:** 1-bit Span Radix Tree diagram

## 6 Inverted Indexes

An *inverted index* stores a mapping of words to records that contain those words in the target attribute. These are sometimes called a *full-text search indexes* in DBMS's. Inverted indexes especially useful for keyword searches.

Most of the major DBMS's support inverted indexes natively, but there are specialized DBMS's where this is the only table index data structure available.

### Query Types

Inverted indexes allow users to do three types of queries that cannot be performed on B+Trees. Firstly, inverted indexes permit *phrase searches*, which locate records that contain a list of words in the given order.

They also allow *proximity searches* that record where two words occur within $n$ words of each other.

Thirdly, inverted indexes allow *wildcard searches*, which find records that contain words that match some pattern (e.g., regular expression).

### Design Decisions

The two primary decision decisions for developing inverted trees are what to store and when to update.

Inverted indexes need to be able to store at least the words contained in each record (separated by punctuation characters). They may also include additional information such as the word frequency, position, and other meta-data.

Updating an inverted index every time the table is modified is expensive and slow. Because of this, nverted indexes usually maintain auxiliary data structures to stage updates and then update the index in batches.