Lecture #09: Indexes II

15-445/645 Database Systems (Fall 2025)
https://15445.courses.cs.cmu.edu/fall2025/
Carnegie Mellon University
Andy Pavlo

1 Index vs. Filters

An **index** is a data structure of a subset of a table's attributes that are organized and/or sorted to the location of specific tuples using those attributes, and it answers the question (where is the data?). While a **filter** is a data structure that answers set membership queries (does this element exist in the set?). If the DBMS knows an element is not in a set, we save time finding it in the set while it does not exist. For example, within a chained hash table, we can put a filter at each bucket pointer. If the filter says negative, we then know the key is not in the chain thus saving our time traversing through the whole chain, which is costly. We need both indexes and filters in a database, and sometimes putting a filter on the index will help speed up operations.

2 Bloom filter

A **Bloom filter** is a probabilistic filter implemented with bitmap. By probabilistic, it means a Bloom filter does not always give the correct answer to a set membership query (false positives). However, a Bloom filter guarantees that it will never has false negatives.

This implies that false positives could happen. The false positive rate can be calculated via Bloom Filter Calculator.

A Bloom filter needs to define

- Size of the bitmap
- Numbers of hash functions to use

Bloom filter API will only support two operations, and it is not possible to delete an element from the filter:

Insert(x)

For insertion, the pre-defined hash functions are used on the inserted element x. For each function's output hash value, we modular it with the bitmap size, then set the corresponding position in the bitmap to one. See Figure 1.

Lookup(x)

For lookup, a similar operation is done on element x. Each hash function takes x as input and modular the output value with bitmap size. If any of the corresponding positions in the bitmap is not one, a false is returned. Otherwise a true is returned (one has to go to the set and see if it's actually in it).

Other Variations

- **Counting Bloom filter**: Instead of bits, use integers to count the number of occurrences of akey in a set. This supports dynamically adding and removing keys.

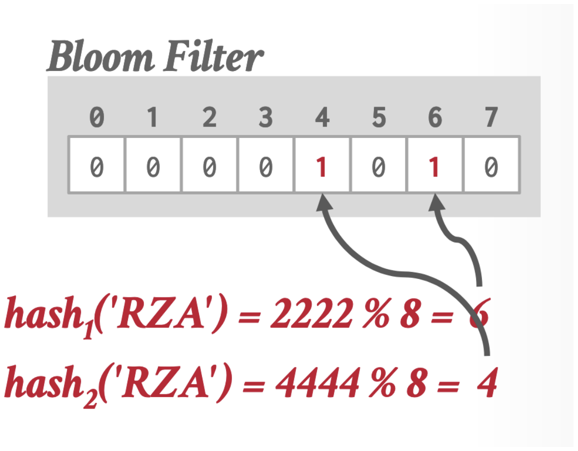


Figure 1: Insert 'RZA' into a Bloom filter with two hash functions

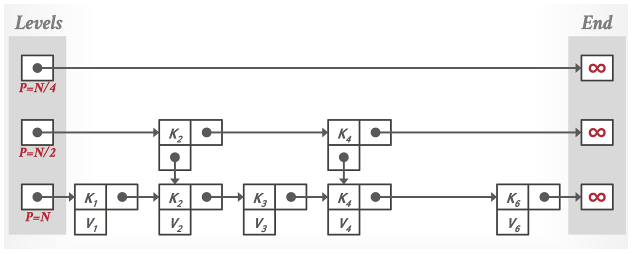


Figure 2: An overview of a skip list

- **Cuckoo filter**: Same idea to Cuckoo Hash, but store fingerprints of elements instead. Also supports dynamically adding and removing keys.
- **Succinct range filter**: An immutable compact trie that supports approximate exact matches and range filtering.

3 Skip List

A *Skip List* uses multiple levels of linked lists to skip some nodes and thus traverse faster. See Figure 2. Each level has $\frac{1}{2}$ the keys of the level below it: The bottom level contains all the keys in sorted order. Second level links every other key, and so on.

Like the B+tree, it stores keys in an ordered manner. However, it does not require rebalancing during insertion or deletion, and still provides $O(\log n)$ approximate search times. It is commonly seen in an in-memory data structure such as memtable.

Find

Go to the top-level linked list and traverse until the value is about to be greater than the target. Then go down to the next level and traverse the same way until it reaches the bottom list and then the target key.

Insert

Coins are flipped to decide until which level this new node is going to insert. Insertions on different levels are done from bottom to top, in order to keep the whole data structure intact. Otherwise, a reader from another thread may come across a node and find out there is no pointer to the lower level while the insertion is still not finished.

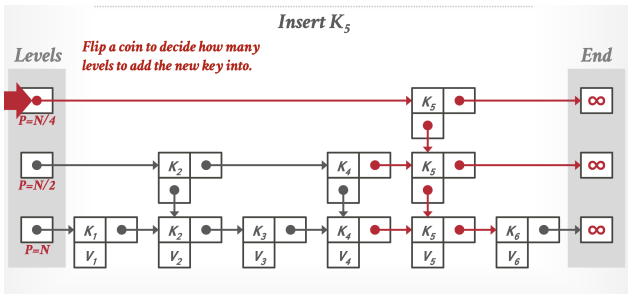

Insert K_5
Levels
Flip a coin to decide how many levels to add the new key into.
End
P=N/4
P=N/2
P=N
K_1 V_1 K_2 V_2 K_3 V_3 K_4 V_4 K_5 V_5 K_6 V_6
∞

Figure 3: The skip list in Figure 2 after K_5 inserted. And it succeeded in two coin flips.

Note that if the linked list is in one direction, each level's insertion could be done by an atomic pointer in-memory swap (swap K_4's next with K_5's next pointer), thus no latch is needed.

Delete

Every node will have a boolean field to mark whether it is deleted or not. At first, a node is marked as deleted instead of removed from the data structure directly to prevent other reader threads from visiting the dead object. Actual deletions are expensive, so it is usually done by a background thread in a fixed interval. Any reader can ignore the deleted node and keep traversing the way down. Later when the node object is to be deleted, the top node will be removed before the bottom one to keep the data structure intact.

Conclusion

Advantages:

- Less memory usage if not including reverse pointer compared to B+tree.
- No rebalancing is needed while inserting and deleting.

Disadvantages:

- Not disk/cache friendly because they do not optimize locality of reference
- Reverse search is non-trivial, it becomes tricky to handle both ascending and descending scans.

4 Trie

Because a B+tree does not provide information about whether a node exists below an inner node or not, it's essential to go down to the leaf node to find out a node does not exist. It costs one buffer pool page miss per tree level.

A *trie* is an order-preserving data structure that stores keys as digits. A trivial way is to make characters (a byte) as digits for strings or bits for other data types. These digits form a tree structure to represent prefixes of every entry inserted into this trie. The tree shape depends on keys and lengths, does not depend on insertion order. And it does not require rebalancing. All operations hve O(k) complexity where k is the length of the key.

The span of each trie level is the number of bits that each partial key/digit represents. If the digit and its prefix exist in the corpus, then a pointer to the next level node is stored at that digit, otherwise, a null is stored. So a n-way Trie means each node will have a fan-out of n.

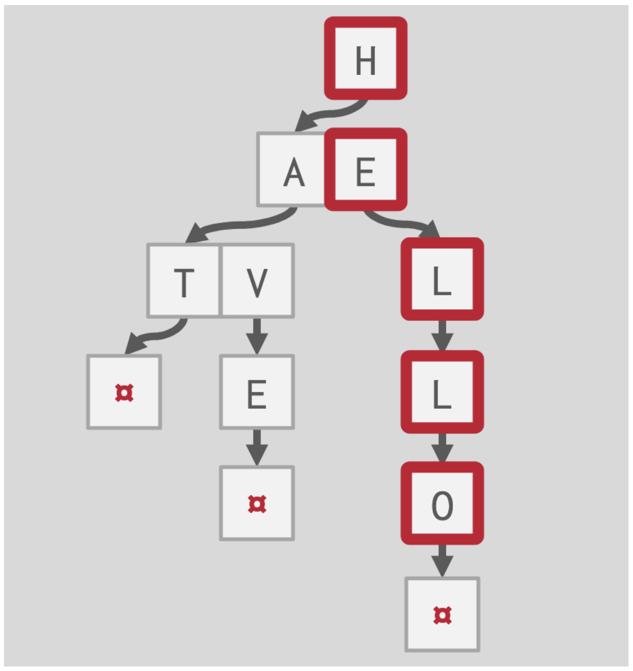


Figure 4: Finding a key "HELLO" in a trie

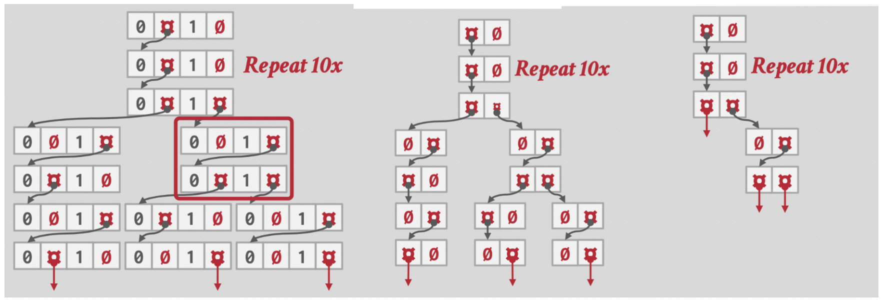


Figure 5: From left to right, horizontal compression and vertical compression are done on a trie with a one-bit span node

Compression

- If we have a known span of a level, we can horizontally compress the node to an array instead of a map. See Figure 5
- If a node has only a single child, we can vertically compress the nodes below the node as there are no branches down there. See Figure 5. This is also called Radix tree. False positives may happen because the full key is no longer embedded in the trie, so DBMS have to check the tuple a node points to see whether a key actually matches.

5 Inverted Index

The indexes we talked about before are only good for point or range searches. They do not support keyword search. For example, a query to find all Wikipedia articles that contain the word "Pavlo".

An inverted index stores an immutable mapping of terms to records (the list of records is called a posting list) that contain those terms in the target attribute.

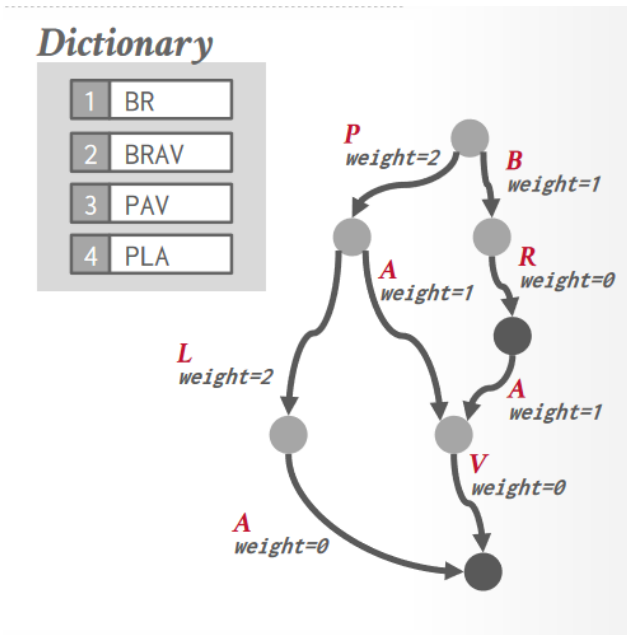


Figure 6: Finite state transducer to find the offset in the dictionary of a term

Lucene Implementation

Lucene is a specialized inverted index engine. The way it stores an inverted index is to have a trie-like data structure called a "finite state transducer" (Figure 6). Instead of storing pointers to a tuple like a trie, it stores weights on every edge. By traversing down to the key one is looking for, a rolling sum of the weights will eventually give the exact position the entry is stored in the mapping.

The data structure is immutable because the weights have been precomputed and fixed to specific dictionary entries. So for a batch of inserts, we will build an immutable transducer. To support incremental updates, a separate transducer is built to hold the newly inserted keys, and there will be a background job that merges transducers together.

In the dictionary of terms, since it's immutable and is built ahead of time, compression techniques such as delta compression, and bit-packing. Pre-aggregation is also supported to accelerate the query time of aggregation queries that group on terms.

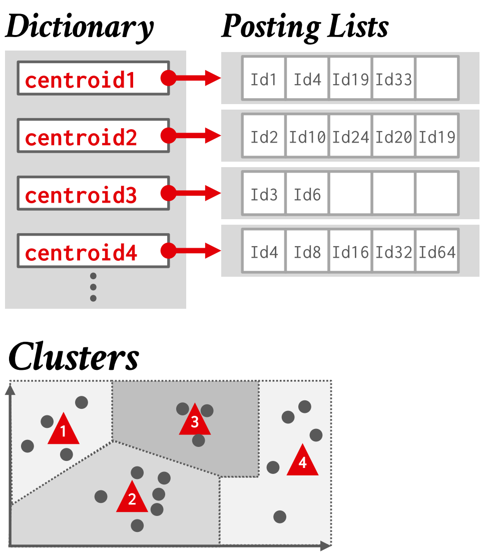
PostgreSQL Implementation

PostgreSQL's *Generalized Inverted Index* uses a B+tree to build the term dictionary. The value of this B+tree leaf node depends on the size of the posting list. For small-sized posting lists, it will be a sorted list of record IDs. For large-sized posting lists, additional B+tree structures will be built to hold the record IDs.

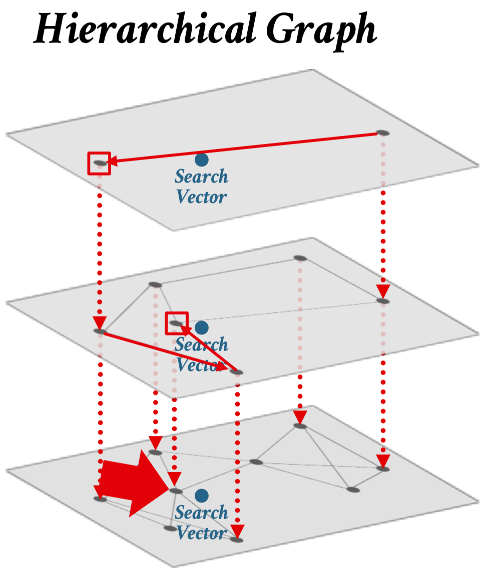
A separate pending list is used to avoid small incremental updates. It logs updates and accumulates to a bulk insert to the dictionary.

Enhancements

- Rankings: DBMS can rank search results based on the frequency of terms in each record relative to other records.
- Tokenizers: Split terms into n-grams to support fuzzy text searches and autocomplete ("did you mean")



(a) IVFFlat Example



(b) HNSW Example

Figure 7: Vector Indexes

6 Vector Index

Inverted indexes can support keyword search, but not the semantic meaning of the content (i.e., the keyword has to be contained in the content exactly). Suppose an application wants to search for records that are **related** to a certain topic, for example, "*hip-hop groups with songs about slinging*". In that case, we need another method other than the inverted index.

Large language models are known for generating **embeddings** for texts, an one-dimensional array of floating point numbers. Embeddings are geometrically close to each other if they have similar semantic meanings, therefore a vector index specialized for nearest-neighbor search can help in the query we were discussing.

However, there is no correct answer to this kind of query compared to those traditional queries we have seen before. There is also a need to filter data before or after finding nearest neighbors.

Inverted Indexes

Partition vectors into smaller groups by a clustering algorithm and then build an inverted index that maps cluster centroids to records. As shown in Figure 7a, the DBMS can compute k-means clustering on embeddings to find centroids. Then assign each embedding to the cluster with its nearest centroid. To search the nearest neighbors, use the same clustering algorithm to locate the query to a group, then look up all the vectors within that group (might also want to look at the groups nearby to improve accuracy).

Example: IVFFlat.

Preprocessing and quantization can be performed on the index to reduce the dimension and thus speed up the lookup time, while the original vectors are still preserved at their original locations.

Graph

The DBMS builds a graph that represents the neighbor relationship between vectors, where each node represents a vector and its edges link to its n nearest neighbors (Figure 7b). To find a match for a given vector, enter the graph and then greedily choose the next edge that moves closer to that vector. The DBMS can also use multiple levels of graphs with the idea similar to skip lists to speed up the search.

Example: FAISS, HNSWlib

7 Optimizations

Partial Indexes

Create an index on a subset of the entire table by adding a WHERE clause to the index definition. This potentially reduces its size and the cost of maintaining it. For example, we could partition indexes by date ranges, creating a new index for each month and year.

Index Include Columns

Embed additional columns in indexes to support index-only queries The extra columns are nly stored in the leaf nodes and are not part of the search key. If all attributes a query needs are available in an index, then the DBMS does not need to retrieve the tuple.