Lecture #08: Indexes & Filters I

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

1 Indexes

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

An *index* is a replica of a subset of a table's attributes that is organized and/or sorted for efficient access to the location of specific tuples. So instead of performing a sequential scan, the DBMS can perform a lookup on the 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 having 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, insertions, and deletions in $\mathbb{O}(log(n))$. It is optimized for disk-oriented DBMSs that read/write large blocks of data as it converts what would have potentially been random I/O to sequential I/O. range scan

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 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 fanout, or the maximum number of children a node can have) with the following properties:

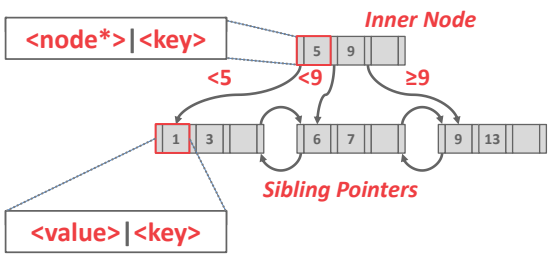
- It is perfectly balanced (i.e., every leaf node is at the same depth).
- Every inner node other than the root is at least half full ($\frac{m}{2} - 1 \leq$ num of keys $\leq m - 1$).
- Every inner node with k keys has k+1 non-null children.

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. Although it is not necessary according to the definition, arrays at every node are almost always sorted by the keys. Two approaches for representing 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.

MySQL

PostgreSQL



m = 4
branch: $\lceil \frac{m}{2} \rceil \sim m$ (2~4)
key : $\lceil \frac{m}{2} \rceil - 1 \sim m-1$ (1~3)

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.

★ The difference between B-Tree & B+Tree ↑

Depending on the index specification, the DBMS will place null keys in either the first leaf node (i.e., `NULL FIRST`) or the last leaf node (i.e., `NULL LAST`). Sibling pointers (introduced by B^{link}-Tree) can be added across each level of the B+Tree, enabling range scans to be performed without backtracking and allowing efficient splits and merges during insertions and deletions from the B+Tree.

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. Overflow!
   - Otherwise split L into two nodes L_1 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 borrow from a sibling.
   - If borrowing fails, merge L and a sibling.

   Underflow!
3. If a merge occurred, you must delete the entry in the parent pointing to L.

Composite Index 复合索引

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

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

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.

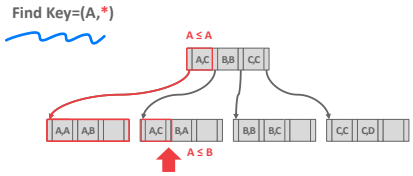


Figure 2: To perform a prefix search on a B+Tree, one looks at the first attribute on the key, follows the path down, and performs a sequential scan across the leaves to find all the keys that one wants.

Duplicate Keys

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

The first approach is to *append record IDs as part of the key*. Since each tuple's record ID is unique, this will ensure that all the keys are identifiable. The DBMS can use partial key lookup to find tuples.

Don't do this!

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 聚集索引 Index Order – Tuple Physical Order

The table is stored in the sorted 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.) e.g. MySQL

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 underfilled 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 strategies that keep underfilled nodes in the tree and rebuild the tree later, which make 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.

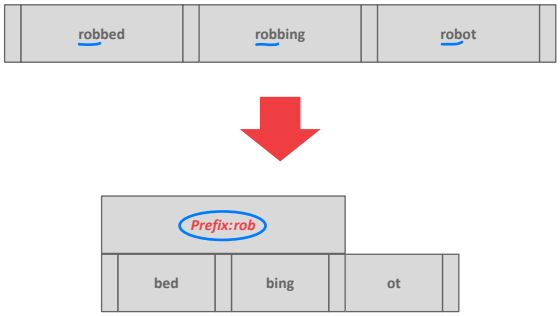


Figure 3: An example of prefix compression. Since the keys are in lexicographic order, they are likely to share some prefix.

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. $K_1\ V_1\ K_1\ V_2 \rightarrow K_1\ V_1\ V_2$

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.

We don't need the entire key!

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.

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.

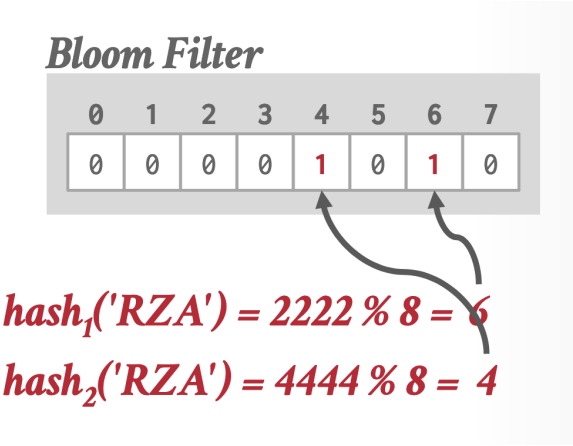


Figure 1: Insert 'RZA' into a Bloom filter with two hash functions

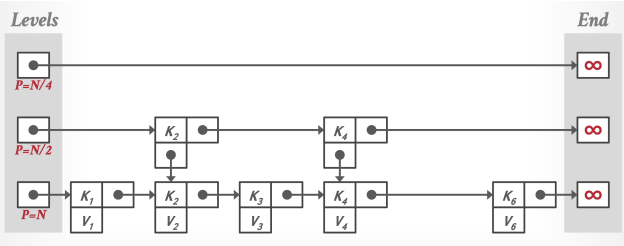


Figure 2: An overview of a skip list

Lec# 7

- **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

Top-down

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

Bottom-up

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.

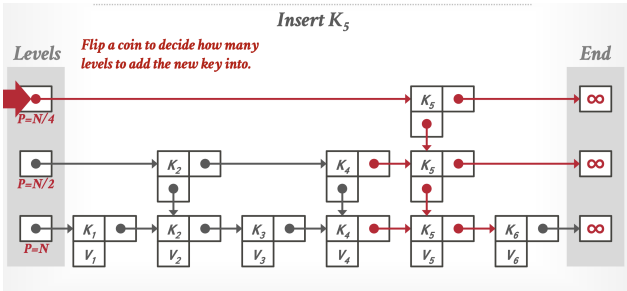


Figure 3: The skip list in Figure 2 after K_5 inserted. And it succeeded in two coin flips.

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

Bottom-up & Logical → Physical

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.

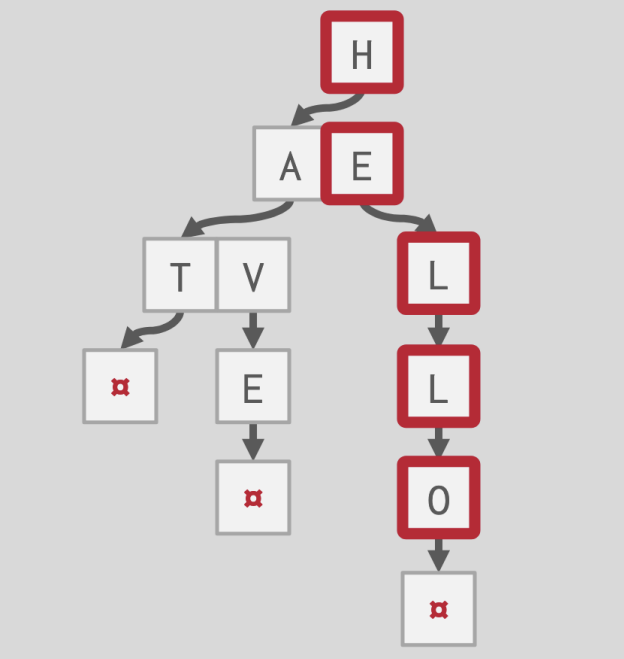


Figure 4: Finding a key "HELLO" in a trie

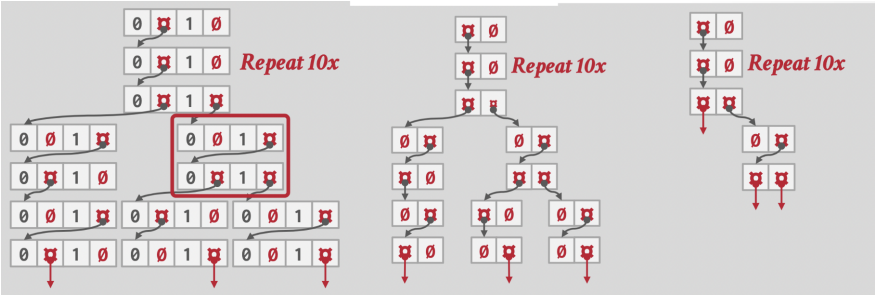


Figure 5: From left to right, horizontal compression and vertical compression are done on a trie with a one-bit span node

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.

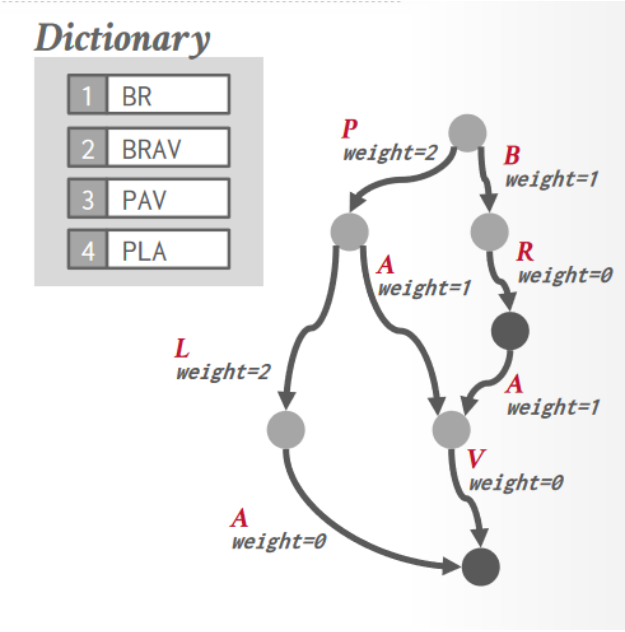


Figure 6: Finite state transducer to find the offset in the dictionary of a term

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.

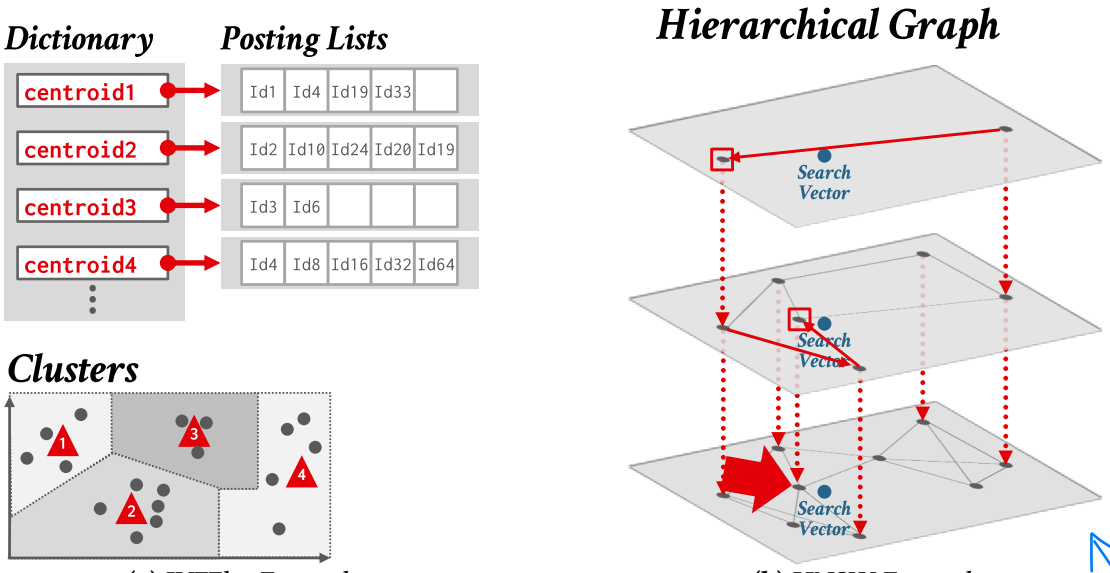
PostgreSQL Implementation

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

INCLUDE

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.

Lecture #10: Index Concurrency Control

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

1 Index Concurrency Control

So far, we assumed that the data structures we have discussed are single-threaded. However, most DBMSs needs to allow multiple threads to safely access data structures to take advantage of additional CPU cores and hide disk I/O stalls.

There are some systems that use a single-threaded model as some arguments say that it would be faster without latching (e.g. Redis). Additionally, an easy way to convert a single-threaded data structure to a multi-threaded one is to use a single read-write lock to guard the entire structure, but this is not an efficient method.

A *concurrency control* protocol is the method that the DBMS uses to ensure "correct" results for concurrent operations on a shared object from multiple workers. We use the term "workers" here to more generically describe multiple concurrent accessors: some DBMSs use processes (e.g. Postgres) while others use threads, etc.

A protocol's correctness criteria can vary:

- **Logical Correctness:** This means that the worker is able to read values that it expects to read, e.g. a thread should read back the value it had written previously.
- **Physical Correctness:** This means that the internal representation of the object is sound, e.g. there are no pointers in the data structure that will cause a worker to read invalid memory locations.

For the purposes of this lecture, we only care about enforcing physical correctness. We will revisit logical correctness in later lectures.

2 Locks vs. Latches

There is an important distinction between locks and latches when discussing how the DBMS protects its internal elements.

Locks

A lock is a higher-level, logical primitive that protects the contents of a database (e.g., tuples, tables, databases) from other transactions. Transactions will typically hold a lock for its entire duration. Database systems can expose to the user the locks that are being held as queries are run. There should be some higher-level mechanism to detect deadlocks and rollback changes.

Latches

Latches are the low-level protection primitives used for critical sections the DBMS's internal data structures (e.g., data structure, regions of memory) from other workers. Latches are held for a short period for a simple

operation in a database system (i.e., page latch). Unlike with locks, it is the worker's responsibility to avoid deadlocks / roll back changes (rather than another mechanism within the system).

There are two modes for latches:

- **READ:** Multiple workers are allowed to read the same item at the same time. A worker can acquire the latch in read mode even if another thread has already acquired it in read mode.
- **WRITE:** Only one worker is allowed to access the item. A worker cannot acquire a write latch if another thread holds the latch in any mode. A worker holding a write latch also prevents other worker from acquiring a read latch.

r r ✓ rw ✗ ww ✗

3 Latch Implementations

There are some key goals we want to achieve when implementing latches:

- Small memory footprint (ideally measured in bytes)
- Fast execution path when there is no contention
- Decentralised management of latches
- Avoid expensive system calls OS is not your friend!

The underlying primitive that used to implement a latch is through atomic instructions that modern CPUs provide. With this, a thread can check the contents of a memory location to see whether it has a certain value.

- **Atomic Instruction Example: compare-and-swap (CAS)**
  Atomic instruction that compares contents of a memory location M to a given value V
  - If values are equal, installs new given value V' in M
  - Otherwise, operation fails

There are several approaches to implementing a latch in a DBMS. Each approach has different trade-offs in terms of engineering complexity and runtime performance. These test-and-set steps are performed atomically (i.e., no other thread can update the value in between the test and set steps.

Test-and-Set Spin Latch (TAS)

Spin latches are a more efficient alternative to an OS mutex as it is controlled by the DBMSs. A spin latch is essentially a location in memory that threads try to update (e.g., setting a boolean value to true). A thread performs CAS to attempt to update the memory location. The DBMS can control what happens if it fails to get the latch. It can choose to try again (for example, using a while loop) or allow the OS to deschedule it. Thus, this method gives the DBMS more control than the OS mutex, where failing to acquire a latch gives control to the OS.

- **Example:** `std::atomic<T>`
- **Advantages:** Latch/unlatch operations are efficient (single instruction to lock/unlock on x86).
- **Disadvantages:** Not scalable nor cache-friendly because with multiple threads, the CAS instructions will be executed multiple times in different threads. These wasted instructions will pile up in high contention environments; the threads look busy to the OS even though they are not doing useful work. Furthermore, in a non-uniform memory architecture polling the memory location of the latch may be especially expensive if it is located in the local cache/memory of another CPU.

busy-waiting

Blocking OS Mutex

One possible implementation of latches is the OS built-in mutex infrastructure. Linux provides the futex (fast user-space mutex), which is comprised of (1) a spin latch in user-space and (2) an OS-level mutex. If the DBMS can acquire the user-space latch, then the latch is set. It appears as a single latch to the DBMS even though it contains two internal latches. If the DBMS fails to acquire the user-space latch, then it goes down into the kernel and tries to acquire a more expensive mutex. If the DBMS fails to acquire this second mutex, the OS de-schedules the thread and notifies the thread when the mutex becomes available.

OS mutex is generally a bad idea inside of DBMSs as it is managed by OS and has large overhead.

- **Example:** `std::mutex` ➡ pthread_mutex_t ➡ futex
- **Advantages:** Simple to use and requires no additional coding in DBMS.
- **Disadvantages:** Expensive and non-scalable (about 25 ns per lock/unlock invocation) because of OS scheduling. OS has to do its own bookkeeping with its own latching to manage which threads should be asleep / which shoud be notified.

Reader-Writer Latches

Mutexes and Spin Latches do not differentiate between reads/writes (i.e., they do not support different modes). The DBMS needs a way to allow for concurrent reads, so if the application has heavy reads it will have better performance because readers can share resources instead of waiting.

A Reader-Writer Latch allows a latch to be held in either read or write mode. It keeps track of how many threads hold the latch and are waiting to acquire the latch in each mode. Reader-writer latches use one of the previous two latch implementations as primitives and have additional logic to handle reader-writer queues, which are queues requests for the latch in each mode. Different DBMSs can have different policies for how it handles the queues.

One thing to notice is that different reader-writer lock implementations have different waiting policies. There are reader-preferred, writer-preferred, and fair reader-writer locks. The behavior differs in different operating systems and pthread implementations.

- **Example:** `std::shared_mutex` ➡ pthread_rwlock_t ➡ pthread_mutex_t/pthread_cond_t
- **Advantages:** Allows for concurrent readers.
- **Disadvantages:** The DBMS has to manage read/write queues to avoid starvation. Larger storage overhead than spin Latches due to additional meta-data.

4 Hash Table Latching

It is easy to support concurrent access in a static hash table due to the limited ways threads access the data structure. For example, all threads move in the same direction when moving from slot to the next (i.e., top-down). Threads also only access a single page/slot at a time. Thus, deadlocks are not possible in this situation because no two threads could be competing for latches held by the other. When we need to resize the table, we can just take a global latch on the entire table to perform the operation.

Latching in a dynamic hashing scheme (e.g., extendible) is a more complicated scheme because there is more shared state to update, but the general approach is the same.

There are two approaches to support latching in a hash table that differ on the granularity of the latches:

- **Page Latches:** Each page/block has its own Reader-Writer latch that protects its entire contents. Threads acquire either a read or write latch before they access a page. This decreases parallelism

because potentially only one thread can access a page at a time, but accessing multiple slots in a page will be fast for a single thread because it only has to acquire a single latch. trade-off

- **Slot Latches:** Each slot has its own latch. This increases parallelism because two threads can access different slots on the same page. But it increases the storage and computational overhead of accessing the table because threads have to acquire a latch for every slot they access, and each slot has to store data for the latches. The DBMS can use a single mode latch (i.e., Spin Latch) to reduce meta-data and computational overhead at the cost of some parallelism.

It is also possible to create a latch-free linear probing hash table directly using compare-and-swap (CAS) instructions. Insertion at a slot can be achieved by attempting to compare-and-swap a special "null" value with the tuple we wish to insert. If this fails, we can probe the next slot, continuing until it succeeds.

5 B+Tree Latching

The challenge of B+Tree latching is preventing the two following problems:

- Threads trying to modify the contents of a node at the same time.
- One thread traversing the tree while another thread splits/merges nodes.

Latch crabbing/coupling is a protocol to allow multiple threads to access/modify B+Tree at the same time. The basic idea is as follows:

1. Get latch for the parent.
2. Get latch for the child.
3. Release latch for the parent if the child is deemed "safe". A "safe" node is one that will not split, merge, or redistribute when updated. In other words, a node is "safe" if
   - **for insertion**: it is not full.
   - **for deletion**: it is more than half full.

Note that read operations do not need to worry about the "safe" condition (as read operations will not change the size of any node in the tree).

Basic Latch Crabbing Protocol:

- **Find:** Start at the root and go down, repeatedly acquire latch on the child and then unlatch parent.
- **Insert/Delete:** Start at the root and go down, obtaining `X` latches as needed. Once the child is latched, check if it is safe. If the child is safe, release latches on all its ancestors.

The order in which latches are released is not important from a correctness perspective. However, from a performance point of view, it is better to release the latches that are higher up in the tree since they block access to a larger portion of leaf nodes.

Optimistic Latch Crabbing Protocol: The problem with the basic latch crabbing algorithm is that transactions **always** acquire an exclusive latch on the root for every insert/delete operation. This limits parallelism. Instead, one can assume that having to resize (i.e., split/merge nodes) is rare, and thus transactions can acquire shared latches down to the leaf nodes. Each transaction will assume that the path to the target leaf node is safe, and use `READ` latches and crabbing to reach it and verify. If the leaf node is not safe, then we abort and do the previous algorithm where we acquire `WRITE` latches.

- **Find:** Same algorithm as before. Assume safe at first
- **Insert/Delete:** Set `READ` latches as if for find, go to leaf, and set `WRITE` latch on leaf. If the leaf is not safe, release all previous latches, and restart the transaction using the original Insert/Delete protocol.

Some latch implementations allow for upgrading a read latch to a write latch while holding the latch. In

this way it may be possible to not restart the entire traversal of the B+tree when a leaf is not safe, but it is less straightforward and possibly requires already holding some ancestor read latches.

Leaf Node Scans

The threads in these protocols acquire latches in a "top-down" manner. This means that a thread can only acquire a latch from a node that is below its current node. If the desired latch is unavailable, the thread must wait until it becomes available. Given this, there can never be deadlocks. similar to hash tables

However, leaf node scans are susceptible to deadlocks because now we have threads trying to acquire exclusive locks in two different directions at the same time (e.g., thread 1 tries to delete, thread 2 does a leaf node scan). Index latches do not support deadlock detection or avoidance.

Thus, the only way programmers can deal with this problem is through coding discipline. The leaf node sibling latch acquisition protocol must support a "no-wait" mode. That is, the B+tree code must cope with failed latch acquisitions. Since latches are intended to be held (relatively) briefly, if a thread tries to acquire a latch on a leaf node but that latch is unavailable, then it should abort its operation (releasing any latches that it holds) quickly and restart the operation. Some trade-off could be made here is to wait a little more before restarting itself, the wait time can depend on how much work has been done so far for the thread.

kill itself

In the case the index is a high-contention data structure where a leaf scan thread could possibly fail consistently, should there be a mechanism to solve this, a higher level system thread scheduler detecting this and trying to schedule out other threads to get the leaf scan thread done. This mechanism does not necessarily need to be incorporated in a data structure's implementation but should rely on a higher level part of the system.