Indexes

Remy Wang

May 2026

One of the main reasons DBs are fast is thanks to *indexes*. An index is a data structure for efficiently accessing parts of a table.

It's like the index of a book, where key terms are displayed in sorted order pointing to which pages they appear on.

A powerful index data structure is the B+ Tree, also used in file systems.

B+ Tree addresses 2 problems of a simple sorted array:

1. Binary searching a sorted array requires many expensive random accesses
2. Inserting into the array requires moving a lot of data

We can fix the second problem by storing the data in a *binary search tree*, where each node stores the "middle element" for that level.

In a BST, insertion only touches parts of the tree (the path from the root) which is more efficient than inserting into a sorted array.

A B+ Tree can be thought of as an "n-ary" search tree: each level of the tree can store multiple values, and each node can have multiple children.

The best way to understand B+ Trees is by playing with them, and the inimitable Vincent Lin built a wonderful tool for this:

https://bptvisualizer.netlify.app

The rest of the slides should be read while you play with the visualizer.

Click Import > Use Initial Example > Import

This creates the example tree. Click “Show full nodes” or press F

There are 2 kinds of notes in a B+ Tree: internal nodes and leaf nodes.

All actual data live in the leafs, while the internal nodes work as guides to search the leafs

Each node has a *capacity* which is the number of values it stores.

Some of the entries may be empty in each node.

Each non-empty value v in an internal node points to two children: the left pointer points to a subtree where all values are $< v$, and the right pointer to a subtree where values are $\geq v$.

So when searching, we start from the root, compare the searched value with each entry in the node, then follow the appropriate pointer.

Try searching for a value using the web tool!

That was a *point query* when we just search for one single value.

B+ Trees also support range queries, where we look for bunch of values falling in a range.

For example, to find values in the range [0, . . . , 66], we first search for the left boundary 0, then keep scanning to the right until we find 66 (or a larger number).

This explain what the dashed lines connecting the leafs are for: they are "next pointers" that let you continue scanning to the next leaf when doing a range query.

A B+ Tree must be *balanced*, meaning all leafs must be at the same level

Every node except for the root must also be at least half-full (give or take 1).

In our example, because the capacity is 3, a node must at least hold 1 entry ($3/2 = 1.5$, rounding down to 1).

You can increase the capacity and start inserting random nodes to see that each node (minus the root) never goes below half-full.

Because of these 2 invariants, the height of the B+ Tree is at most (roughly) $\log_{c/2} N$ where c is the capacity and N is the number of data items stored.

This means while searching the tree, we need no more than $\log_{c/2} N$ random reads, which is very small with a large c.

Inserting into a B+ Tree is more involved.

The first step is to search for the right slot to insert into. This follow the same algorithm as searching.

Once the slot is found, we tentativly insert into that slot.

If doing so does not exceed capacity, we're done!

To see an example, load the inital tree again and try inserting 9.

But if the tentative insertion exceeds the leaf capacity, we *split the leaf* in half.

The two halfs become siblings in the new tree, and we need to make a new parent for them.

To do that, take the first value from the second half, and insert that into the parent of the old node.

To see an example, try inserting 10 in the web tool (after 9 is inserted)

In the last case, creating the new parent does not exceed the capacity of the parent node, so we're done.

But when it does, we need to also split the parent! This works a little differently.

When an internal node's capacity is exceeded, we split it into *3 parts*:

- the first part is still the first half
- the second part contains just a single value right after the first half
- the last part contains the remainig values

In other words, we single out the first value from the second half.

To see an example, now insert 11 and 12 to the web tool (remember you can jump back to any earlier step by clicking on the history on the left). Inserting 12 first triggers a split at the leaf, which adds a new value to the parent; but this in turn overfills the parent, so the parent further splits, and the middle part is *promoted* to *its* parent, i.e. the root.

The reason we're splitting into 3 parts is that we don't want to duplicate values on internal nodes, so we *move* the first value in the second half instead of copying it.

At a leaf, because we must store all data values, we have no choice but copy the first value from the second half to the new parent.

Another useful data structure (not strictly an index) is the Bloom filter.

In real DB workloads, most probes into a hash table actually fail, because when joining 2 tables, the system opts to build the hash map on the smaller table and scan the large table to probe.

But as there are more rows from the large side, most probes would fail.

A Bloom filter helps us *fail faster*

A Bloom filter is an approximate data structure that remembers a set of values.

It's approximate, because you can ask it "has this element been added to your set?", and it'll answer "probably yes", or "definitely not".

In other words, Bloom filters can have *false positives*, but never *false negatives*.

The filter itself is simply implemented as a bit vector, initialized to be all-0.

To add an element x to a Bloom filter, first hash x with k independent hash functions to get k positions.

Then, set those positions to 1 in the bit vector.

Example: a Bloom filter with an 8-bit vector and $k = 2$ hash functions.

0 0 0 0 0 0 0 0

Add "cat": $h_1(\text{cat}) = 1$, $h_2(\text{cat}) = 5$

0 1 0 0 0 1 0 0

Add "dog": $h_1(\text{dog}) = 5$, $h_2(\text{dog}) = 6$

0 1 0 0 0 1 1 0

To check if a given value is in the filter, simply hash that value with the same hash functions, and check if the corresponding bits are 1.

```
0 1 0 0 0 1 1 0
  *       *
```

Query “cat”: positions 1 and 5 are both 1 $\rightarrow$ “probably yes”

```
0 1 0 0 0 1 1 0
    *       *
```

Query "fox": $h_1(\text{fox}) = 2$, $h_2(\text{fox}) = 6 \rightarrow$ "definitely not"

```
0 1 0 0 0 1 1 0
  *         *
```

Query "cow": $h_1(\text{cow}) = 1$, $h_2(\text{cow}) = 6$.

Positions 1 and 6 are both 1 $\rightarrow$ "probably yes" — but "cow" was never added! This is a *false positive*.

A Bloom filter is useful because it can very compactly store a very large set.

So for each hash table, we can create a Bloom filter storing all the hash keys, and the filter will be much smaller than the table and can fit in fast memory (e.g. CPU cache).

Then, before we search the hash table, we can first check the Bloom filter to see if the key to be searched is present.

Because we expect most probes to fail and checking the Bloom filter is fast, this will let us "fail faster" than doing the expensive hash table lookup.