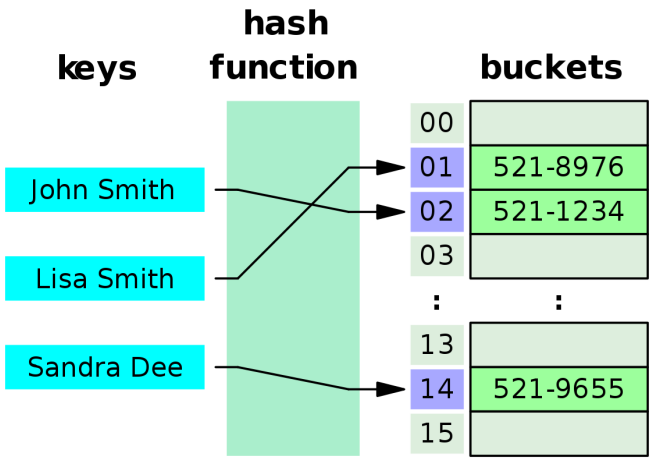
# B+ Tree vs Hash Index (and when to use them)

*By Cal Mitchell*

Indexes are the fundamental unit of database performance. There are many index types, but the two most common are the B+ Tree and the hash index.

B+ trees are the default index type for most database systems and are more flexible than hash indexes. They offer excellent lookup and insertion times when configured correctly, and my personal opinion is that you should stick with B+ trees unless you're trying to optimize an *extremely* performance sensitive table.

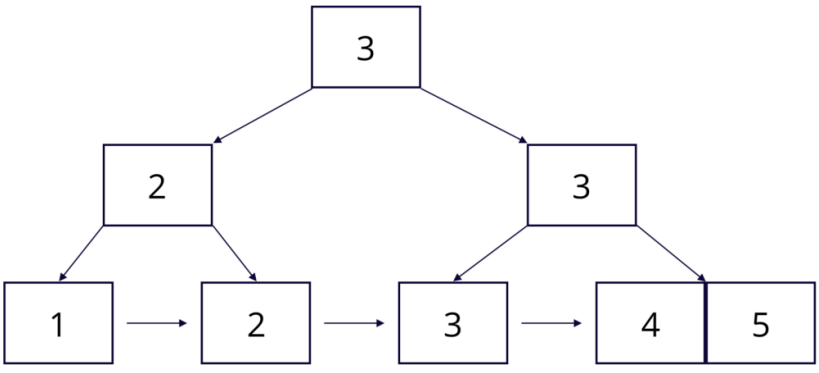



For example, later on, I will show that it only takes 4 disk I/Os to find a value in 8 TB of data using a properly configured B+ tree. No, that's not a typo. Unless you absolutely need to save 3 disk I/Os, then you should preserve the flexibility of a B+ tree.

## Intro to each index type

Let's start by introducing the B+ tree and hash index.

### B+ Tree index

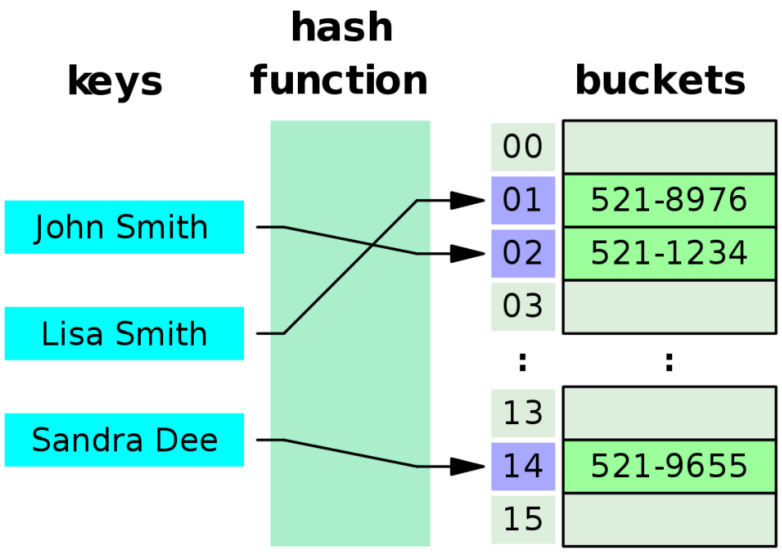


A B+ Tree is a tree data structure with some interesting characteristics that make it great for fast lookups with relatively few disk IOs.

- A B+ Tree can (and should) have many more than 2 children per node.
- A B+ Tree is self balancing.
- A B+ Tree holds keys in internal nodes, but only holds values in the leaf nodes (the nodes on the bottom).
- A B+ Tree is sorted (values ascend from left leaf node to right leaf node).

**Here is a great tool for visualizing what a B+ Tree looks like, and how they behave on insertions / other operations**!

# Hash index




A hash index takes the key of the value that you're indexing and hashes it into buckets. In case you're not aware, a hash function is something that takes an input, and produces a different, somewhat random, and hopefully unique, output.

- "Somewhat random" - Sometimes you actually want the hash function to place certain things somewhat close together on disk, because they are frequently read together. In this instance, you would want to use a "location sensitive hash function".
- "Hopefully unique" - Sometimes two different inputs produce the same output. This is called a hash collision. In this case, the database usually uses a different hash function to resolve the collision. Hashing these values allows you to do 0(1) equality lookups based on the value of the indexed column. All you have to do to find the exact location of the rows with that value is hash the value you're trying to find, and look in the place where the hash value leads you!

# When to use B+ Tree vs Hash Index

If you are 100% certain that a column will only ever need to be looked up via direct equality (eg... where id = X), then you can use a hash index. Even in that case, unless that table is at the core of your business and is becoming a bottleneck for system performance in general, I would just use a B+ tree anyways. Why? Because you're really not giving up very much!

# B+ Tree

## Flexibility

- The default index type
- Quite flexible! Allows you to use <, >, =, != operators, as well as others depending on your DB implementation.
- If you don't know what to use, just use a B+ Tree
- Excellent for integer and date columns, where you will frequently compare values.
- Is very fast for "range style" queries, such as selecting records from a certain date range.
- B+ Trees store links between each leaf node, essentially creating a sorted doubly linked list!

## Performance

Lookup times: log f (N/F) disk IOs, where:

- f - The "fanout" of your B+ Tree, or how many children each node has.
- N - The number of disk pages needed to hold the indexed keys
- F - The "slack" of our tree. Usually 2/3.
- Slack, in this instance, is how many nodes we build with empty values. Keeping empty values in the tree allows us to do inserts without moving a ton of data around, and keeps the overhead of maintaining the index lower.

**Example**

- Let's say you're indexing a table with 1 trillion records that has an 8 byte ID column. The ID column alone would be 8 trillion bytes (8 TB). As in, just the *keys* of your data, the index itself, is 8 TBs. An enormous table, in other words.
- Assume your DB blocks are 8 KB (a typical size).
- N is 1 million pages (8 TB / 8 KB) to hold indexed values.
- f of 5460
- F of 2/3
- Applying the formula above, plus one extra disk IO to find the root node, comes out to (drum roll please)… 4 disk IOs to look up a single value in 8 TB of data! Wowza!

If you actually have DB tables where just the *index* is 8 TB, then you are probably at Google. Cool! Hire me! For most people however, your indexes will probably be small enough to fit in RAM, and won't require any IOs at all!

# Hash Index

### Flexibility

Only allows exact equality (and by extension, inequality) operators.

### Performance

Lookup times: 0(1). It's hard to beat that!

## Conclusion

I hope this article shed some light on when you should use each type of index.