B-Tree Index

by Sophia

WHAT'S COVERED

This lesson explores using B-tree indexes in databases, in three parts. Specifically, this lesson will cover:

1. Understanding B-Tree Indexes

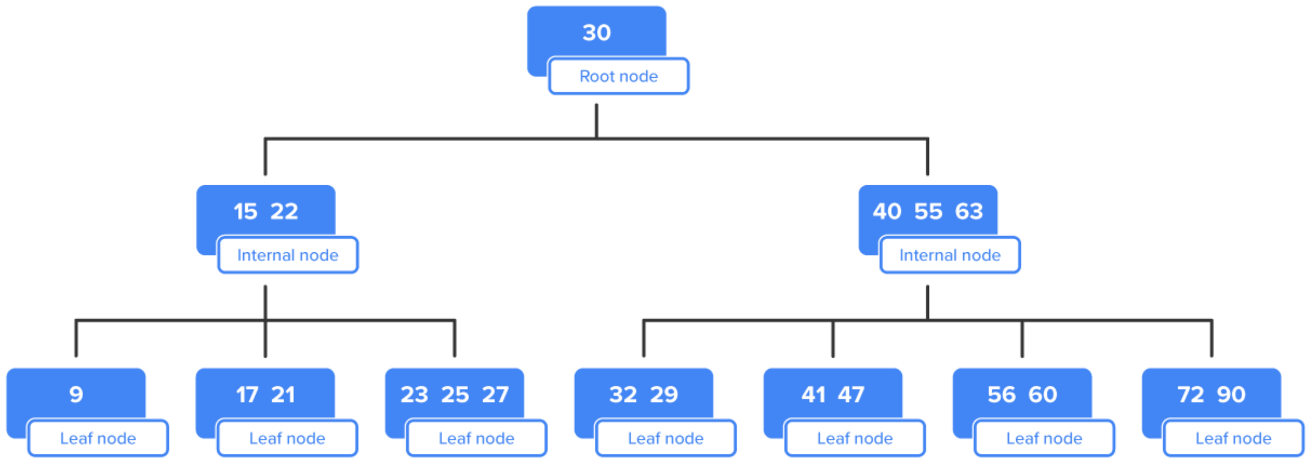
A **B-tree index** (short for balanced tree) is a common index type in relational databases.

The B-tree is composed of **nodes**. A **key** is a representation of the data you are searching for. Each node contains a different small subset of all the keys. A **value** is the data associated with a key.

The topmost node is called the **root node**, and all the other nodes are **child nodes**.

The search algorithm begins at the root node and then uses the keys to navigate through the nodes. At each node, it looks for the key it is seeking. If it doesn't find it, it moved to one of that node's child nodes.

Here's a very simple example. In the following tree, the root node contains one key: 30. The search begins there. Is the desired key 30? If yes, then the search is over. If not, it moves to one of the next-level child nodes. But which one?



Look closely and you'll see that the numbers in the left-hand node in the second level are less than 30, and the numbers in the right-hand node are more than 30. So, the search continues by first evaluating whether the number it is looking for is greater than or less than 30. Then it moves to either the left or right node accordingly.

At the next level, the algorithm evaluates another condition. Is the desired number less than the smaller number, between the two numbers, or greater than the larger number? Depending on the answer, it selects one of the nodes at the next level to move to.

The lines that connect one node level to the next are called **pointers**. The nodes that are between the root node and the leaf nodes are called **internal nodes**.

HINT

The number of keys within a node determines the number of keys at the next child level: It's always the number of keys plus 1. For example, in the above diagram, the second-level node on the left contains two keys and has three child nodes; the one on the right contains three keys and has four child nodes.

And so the algorithm goes, moving downward through the levels, until it finds the desired value. The diagram above shows only three levels, but an actual B-tree may have more. At the bottommost level, the nodes are called **leaf nodes**. The leaf node contains pointers to the table location containing the data being looked up.

B-trees maintain a **balanced structure**, meaning that all leaf nodes are at the same level. It achieves this balance by redistributing keys between nodes as needed when insertions or deletions occur.

TERMS TO KNOW

B-Tree Index

A type of index that uses an upside-down tree to create a hierarchical system of nodes that reference the data being searched.

Node

A logical unit within a B-tree that contains a set of keys and pointers.

Key

A representation of the data being searched for.

Value
The data associated with a key.

Root Node
The node at the top of a B-tree.

Child Node
A node that appears below another node (its parent node) in a B-tree.

Pointer
A connector between one node level and the next.

Internal Node
A node located between the root node and the leaf nodes in a B-tree.

Leaf Node
A node that contains pointers to the table location containing the data being looked up.

Balanced Structure
A B-tree index structure in which all leaf nodes are at the same level.

2. The Benefit of a B-Tree Index

In an indexless search, the database system would need to search the entire table to find the desired value. If there were 10,000 records, it would take up to 10,000 search operations, where it would go through the table line by line comparing the desired value to that record's value.

In contrast, a B-tree search can find any value in much fewer operations. In the above example, there are 20 keys, each one representing a unique value in the database, but finding any key will take a maximum of three operations—one at each level of the tree. Some keys will take fewer than that to find.

3. When Is a B-Tree Index Useful?

When you add indexes to tables, you generally do not have to worry about the best index type to use because the database will handle it for you. However, as a database administrator, it is useful to understand the different index types so you can choose the type that would provide the most benefit depending on the way the table is used. Let's look at some examples of situations where a B-tree would or would not be very useful.

Let's say you have a query that uses a pattern-matching operator such as LIKE with a constant value that is anchored at the start of the string. A B-tree index can be a great help with that. For example, we could look for patterns of the name that starts with “Wal”:

```
SELECT *
```

```
FROM track
WHERE name LIKE 'Wal%';
```

Or for customers that have an email address starting with “ro”:

```
SELECT *
FROM customer
WHERE email LIKE 'ro%';
```

However, the B-tree index would not be useful if we tried to find information in the middle or at the end of the string, like tracks that have “at” in the middle name:

```
SELECT *
FROM track
WHERE name LIKE '%at%';
```

Or customers that have the email with “gmail.com” as the domain name:

```
SELECT *
FROM customer
WHERE email LIKE '%@gmail.com';
```

Other queries on data are based on ranges. For example, you could have open-ended ranges like this:

```
SELECT *
FROM track
WHERE album_id >=5;
```

Or those that have specific ranges that contain values:

```
SELECT *
FROM track
WHERE album_id >= 5 AND album_id <=10;
```

This is the same as if we had the following:

```
SELECT *
FROM track
WHERE album_id BETWEEN 5 AND 10;
```

Each of those could potentially benefit from a B-tree index. A B-tree index is well suited for range queries because it allows for efficient traversal of the index to find all the entries within the range.

```
SELECT *
```

```
FROM track
WHERE album_id <= 5 AND album_id >=10;
```

Here we are looking for items with the album_id less than or equal to 5, while at the same time looking for the album_id being greater or equal to 10. As the album_id cannot be a value that simultaneously meets that criterion, no rows would be returned. More importantly, this would not fit the B-tree index well. Even if we use the OR operator, it will not be as efficient as having the overlapping range:

```
SELECT *
FROM track
WHERE album_id <= 5 OR album_id >=10;
```

4. When to Use a B-Tree Index

A B-tree index is great for instances where you have values that repeat only a few times or are completely unique. In our PostgreSQL database, for example, the track table's name column may have a few repeated names, but for the most part, the names on the tracks are all different. As such, a B-tree index would be very helpful.

A B-tree index is not so great in instances where you have just a few different values. For example, if you are searching on a Boolean data type field that contains only Yes or No values, a B-tree index would not be of much use at all because there are not a lot of different values that need to be waded through.

KEY CONCEPT

When considering any type of index, it's important to weigh the costs and benefits. Because a B-tree index must maintain a balanced structure, every time a record is inserted or deleted, the tree must be rebalanced, and keys must move between nodes. This takes processing time. As such, a B-tree index may not be the best choice for a table that is constantly or frequently gaining or losing records.

TRY IT

Your turn! Open the SQL tool by clicking on the LAUNCH DATABASE button below. Then, enter one of the examples above and see how it works. Next, try your own choices for which columns you want the query to provide.

SUMMARY

In this lesson, you learned how B-tree indexes function and how they benefit certain types of queries. You developed an **understanding of B-tree indexes**, learning that a B-tree index is composed of nodes that contain keys and pointers. A key is a representation of data values you are searching for. The topmost node is the root node. A search begins at the root node and progresses through internal nodes until it finds the key it is searching for. The bottom layer contains leaf nodes. A B-tree index

remains balanced, such that all leaf nodes are at the same level; it shuffles keys between nodes as needed to make that happen.

You learned that **the benefit of a B-tree index** is that it can dramatically decrease the number of search operations needed to locate data in a table with a large number of records. Then, you learned **when a B-tree is useful** by exploring several examples of situations where a B-tree index would or would not be useful. B-tree indexes are best in situations where records are not frequently being added and removed, and where the field being searched contains many different values, or even unique values for each record. When using B-tree indexes with pattern-matching operations, it works best when the pattern is a constant and is anchored at the start of the string. Lastly, you examined **when to use a B-tree index**, noting that B-tree indexes do not work well when searching for a pattern that is in the middle or end of a string, or when querying for a Boolean value.

Source: THIS TUTORIAL WAS AUTHORED BY DR. VINCENT TRAN, PHD (2020) AND FAITHE WEMPEN (2024) FOR SOPHIA LEARNING. PLEASE SEE OUR TERMS OF USE.

TERMS TO KNOW

B-Tree Index
A type of index that uses an upside-down tree to create a hierarchical system of nodes that reference the data being searched.

Balanced Structure
A B-tree index structure in which all leaf nodes are at the same level.

Child Node
A node that appears below another node (its parent node) in a B-tree.

Internal Node
A node located between the root node and the leaf nodes in a B-tree.

Key
A representation of the data being searched for.

Leaf Node
A node that contains pointers to the table location containing the data being looked up.

Node
A logical unit within a B-tree that contains a set of keys and pointers.

Pointer
A connector between one node level and the next.

Root Node

The node at the top of a B-tree.

Value

The data associated with a key.