Understanding B-Tree

vishal rana Follow 8 min read · Jan 22, 2023

As the amount of data being stored continues to grow, understanding how B-trees work and how to use them effectively is becoming increasingly important for developers and database administrators alike.

Introduction

A B-tree is a type of balanced tree data structure that is optimized for storage and retrieval of large amounts of data. B-tree is often used in file systems and databases to index large amounts of data.

B-trees has an ability to maintain a balanced tree structure, even as the number of keys in the tree increases.

Before jumping into the B-tree, let's understand how data is being stored on the disk and then we'll be able to understand the use case for b-tree.

Taking an example of traditional disk space that includes blocks to store data. ***Each block has a fixed size in which data is being stored.***

Traditional hard disk drives (HDD) the block size is usually 512 bytes, while in solid-state drives (SSD) the block size can vary and it can be 4KB, 8KB or

even 16KB.

Let's understand with an example

we have an employee table with some details of our employees:

EMP_ID	EMP_NAME	EMP_GENDER	EMP_PHONE	EMP_ROLE	EMP_AGE	EMP_EXPERIENCE	EMP_DEPT
00001	smith	M	xxx-xxx-xxxx	01	25	4	ENG
00002	allen	M	xxx-xxx-xxxx	01	27	8	ENG
00003	ward	M	xxx-xxx-xxxx	02	32	10	ENG
00004	jones	M	xxx-xxx-xxxx	03	34	12	ENG
00005	Blake	M	xxx-xxx-xxxx	04	21	1	ENG
00006	clark	M	xxx-xxx-xxxx	05	19	0	ENG
00007	harry	M	xxx-xxx-xxxx	06	45	22	ENG
00008	John	M	xxx-xxx-xxxx	01	32	10	ENG
00008	James	M	xxx-xxx-xxxx	02	46	25	ENG
00009	sam	F	xxx-xxx-xxxx	03	19	0	ENG

We can calculate the size of storing one row (an employee) data.

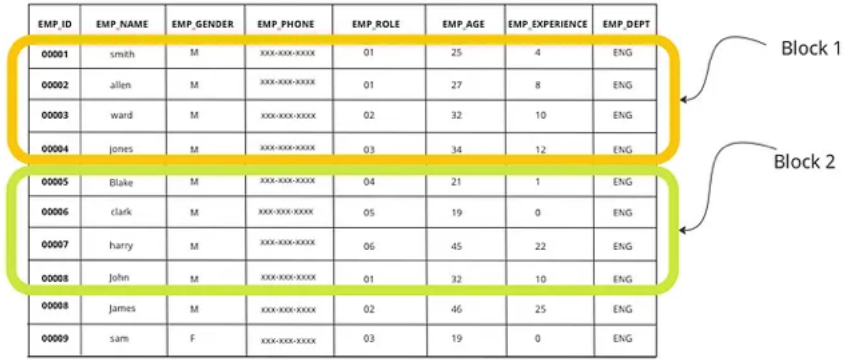
```
EMP_Id - > 20 bytes
EMP_NAME - > 50 bytes
EMP_GENDER - > 4 bytes
EMP_ROLE - > 8 bytes
EMP_AGE - > 10 bytes
EMP_PHONE - > 10 bytes
EMP_EXPERIENCE - > 10 bytes
EMP_DEPT - > 8 bytes

Total bytes we need to store 1 record -> 120 bytes for single employee
```

To store a single employee details in the database , we need **120 bytes** of

storage. As we know that we have a block size of 512 bytes, so according to this, we can store 4 records in a single block of storage.

```
No. of records to be stored in block  =  Block size / Record size
```

we can store record 1–4 in block 1 and 5–8 in block 2 and so on.



how group of records can be stored in multiple blocks

Suppose we have **10,000 employees**, we need **2500 blocks***(keeping in mind, single block can store 4 employees)* to store this amount of data. Suppose we want to fetch a record from a database, then we need to parse all those 2500 block to fetch that particular record from the database.

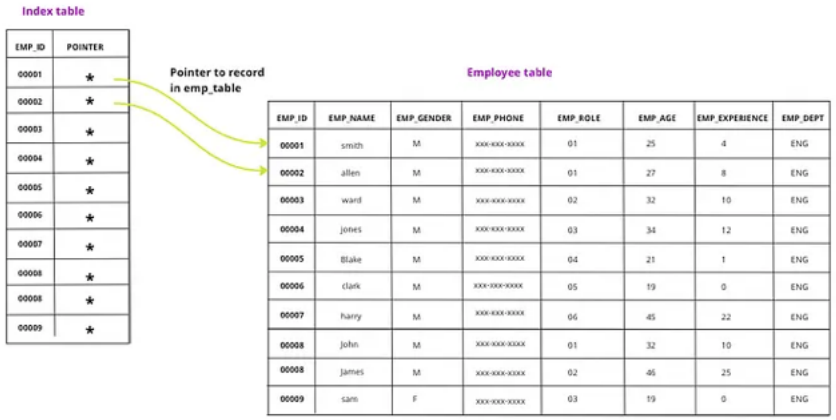
This is an issue, how can we fix this. Here comes the **INDEXING** part. we can add an index on top of our database that will improve our fetch query to get the record from disk that we required.

Let us understand how will indexing can improve this.

Suppose, we want to fetch an employee details from a `emp_id` without indexing we need to go through each block and match the `emp_id` to get the data. But with indexing , we can create a index in our `emp_id` key and make search quick.

Here, how it will work:

we create another index table that will only have `emp_id` and the pointer to the actual record on the disk, although indexes are also stored on the disk, but they are fast compared to the matching data from the emp_table itself.



index contain emp_id and pointing to actual record in database

Index table is also stored on the disk.

> *We also need to store index table on disk , let's calculate how much space it will a acquire.*

```
EMP_Id - > 20 bytes
POINTER - > 6 bytes

Total bytes we need to store 1 entry -> 26 bytes
```

```
No. of records to be              =   Block size / Record size
stored in block

we can store  = 512/26 = 19 records
```

we can store 19 records in a single block, and to store 10,000 records we need to use 526 blocks.

> *So now, we need to search only 526 blocks in order to find a record in database. taking it down from 2500 blocks to 526 blocks.*

This is just one index, we can add index top of another index to improve the performance of search that is what we called as multi-level index.

Most of the database uses b-tree for the indexing and making it really fast to access records.

> *B tree has a worst case time complexity of O(log n)*

Here are few databases that uses b-tree for indexing

> ***MySQL**: MySQL uses B-tree indexes as the default index type for its InnoDB storage engine.*
>
> ***PostgreSQL**: PostgreSQL uses B-tree indexes as the default index type.*
>
> ***SQLite**: SQLite uses B-tree indexes as the default index type.*
>
> ***Oracle Database**: Oracle Database uses B-tree indexes for its indexes.*

> ***Microsoft SQL Server**: SQL Server uses B-tree indexes as the default index type.*
>
> ***MongoDB**: MongoDB uses B-tree indexes for its indexing method.*

Let's understand how B-tree works in depth

B-tree works by dividing the data into smaller, manageable blocks called **nodes**. Each node contains a fixed number of keys and pointers to its child nodes.

let's say we have a B-tree that stores integers and its ***minimum degree "t" is 3***. Each node in the tree can contain at least **3–1=2 keys** and **3 children**.

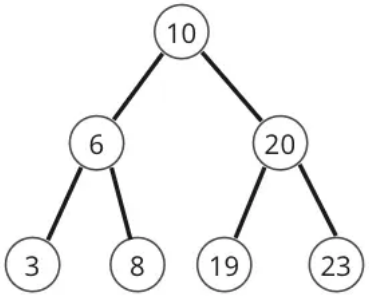
As you know in Binary tree we have at-most of 2 children and each node can contain a single value.

1

Top highlight

Binary tree properties are:

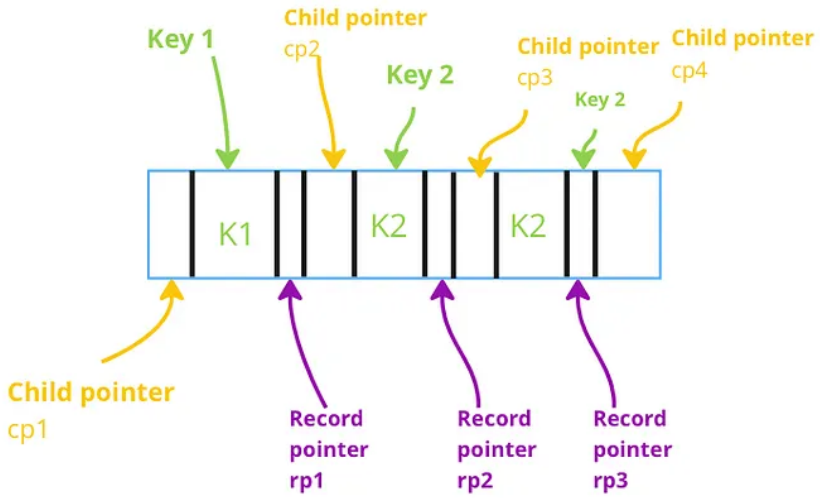
- Left nodes are less than parent and right nodes are greater than parent
- Each node can have max of 2 children



Binary tree

In B-tree , each node can have multiple keys

Here is what a nodes looks like in b-tree



> *Each node will have key, record pointer(points to actual record on disk) and a block pointer(points to the child node) also called child pointer.*

As you can observe, we can have multiple keys in a node. The number of keys can be defined using the **minimum degree "t"**, ***which is the minimum number of keys that a non-root node can have.*** This means that a B-tree can have at least **t-1 keys** and **t children** for each non-root node.

> *In the above diagram, the minimum degree is 4, it means we can have 3 keys and 4 children.*

B -Tree have some properties that we need to satisfy:

- Root node can have minimum of 2 children

- All leaf nodes must be at same level
- Each node should have t/2 children (t is the minimum degree)

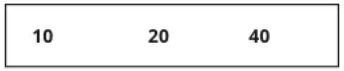
Let's try to create a B-tree and understand its practically how it works

For this example, we'll have **minimum degree t = 4** and data will be

```
keys = 10,20,40,50,65,80,90,25,30
```

As we know , each node can have t-1 keys and t children. so we can store 3 keys in a node and 4 child pointers.

Let's add `1`, `20`, `40` to node one after another.



Adding 10 , 20 , 30 one after another until we reach keys limit i.e-3

Adding 50

Now, we need to add `50`, but we have reached our keys limit (i.e-3). So at this point, we need to split.

we need to create another node and split the current node so that we have the balance between our nodes and we need to promote a root.

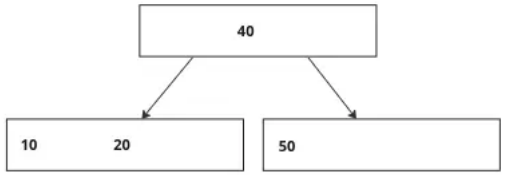
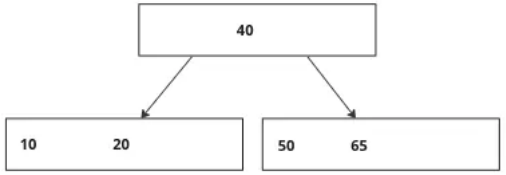


Figure 2.1

Before we have `1`, `20`, `40` in one node, but as we need to create another node, we need to split the node and promote the last key in the node as root, as we have done in the above diagram (Figure 2.1).

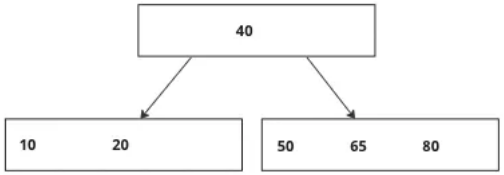
Adding 65

Let's add another key that is `65`. `65` is greater than `40`, we need to check if right node has available space, as the Right node has only 1 key(50), we can add `65` there.



Added key 65 to right of 40 as space is available for new key

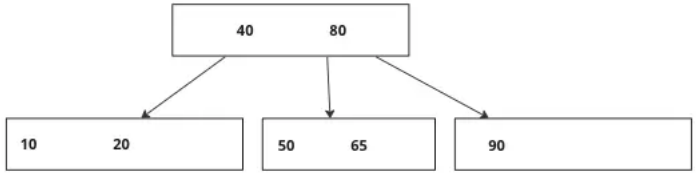
Adding 80



Added key 80 to right of 40 as space is available for new key

Adding 90

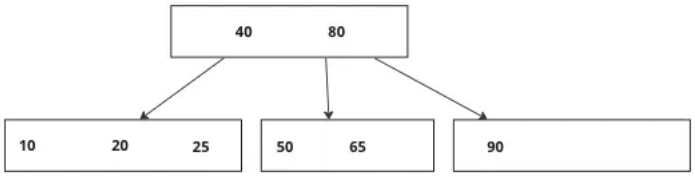
Here, when we try to add 90 in the right node of root , we observe that we have reached the limit and no more keys can be added , so we need to split the node and create new node.



80 will be promoted as root and will merge in root as space is present and 90 will create a new node

Adding 25

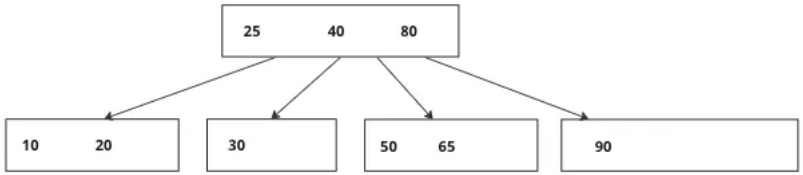
As, `25` is less than `40`, so we need to add that in the left of the root node, and left node has space for one more key. so we can `25` there.



25 added to the left node

Adding 30

As, 30 is less than root, we need to add that to the left node of root, but as left node is also reached its limit of 3 keys, we need to split the node, promote a root and create new node.



25 is promoted as root, as root has space for new key, it is added in root and 30 will create a new node

This is how b-tree construction takes place.

When a query is made to a B-tree, the tree is searched for the requested data in a similar way as a binary search tree. The process typically starts at the root node of the tree and compares the value of the requested data with the keys stored in the current node.

If the requested data is less than the key value, the search continues on the left child of the current node. If the requested data is greater than the key

value, the search continues on the right child of the current node. This process is repeated recursively until the requested data is found, or it is determined that the data is not present in the tree.

> *In B-tree, the search process can be faster because of the large number of children at each node, which allows for more efficient storage of data*

Hopefully this gives you an understand of how b-tree is constructed .

> B-tree construction is a bottoms up approach, as we start from bottom and start constructing the tree upwards.

. . .

If you like this blog and want to read more content like this do check out my other blogs & follow for email updates.

Do checkout these blogs:

- Deep dive into System design
- Most commonly used algorithms.
- Consistent Hashing
- Bit , Bytes And Memory Management
- CAP Theorem Simplified
- Event Driven Architecture

Other Life changing blogs for productivity and Focus:

- 5 Essential Habits for Busy Professionals
- Changing The Life Over A Year
- Mindset of an entrepreneur
- How To Control Your Mind

B Tree | Programming | Database | Software Development | Data Structures

Written by vishal rana

466 followers · 1 following

Explorer

Follow

Responses (2)

Ishu

What are your thoughts?

Ilya Chubarov
Mar 14, 2024

> Binary tree properties are:

I'm sorry to be a nerd, but the correct word here is **Binary Search Tree**

1 Reply

Saikat
Mar 13, 2024

Will the db fetch each nodes on every iteration . or db will laod the entire b-tree into memory to search?

Reply

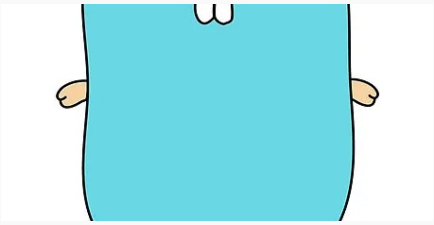
More from vishal rana

vishal rana

Deep Dive Into MongoDB And Its Architecture

MongoDB(nosql) is a widely adopted database that doesn't follow the traditional...

Jan 31, 2023 50 1



vishal rana

Go Architecture Explained

Go or Golang is one of the language for which adoption is increasing day by day at a rapid...

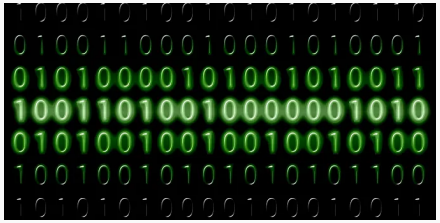
Jan 26, 2023 4 1

vishal rana

Insertion Sort Using Javascript

Insertion sort works with sorting the elements while comparing the previous parsed array t...

Nov 14, 2022 73



vishal rana

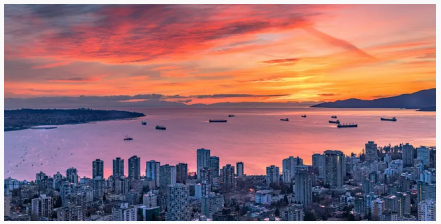
Bit , Bytes And Memory Management

Every software developer should understand what is a bit, bytes and how memory...

Jan 4, 2023 27

See all from vishal rana

Recommended from Medium

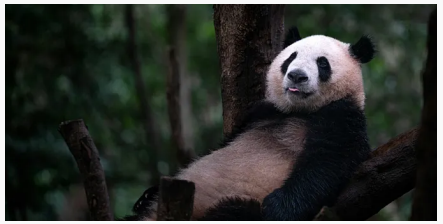


In Acing the Software Engineer Int... by LiveRunG...

Trie

Read full article for free:
https://liverungrow.medium.com/trie-...

Jun 9 3

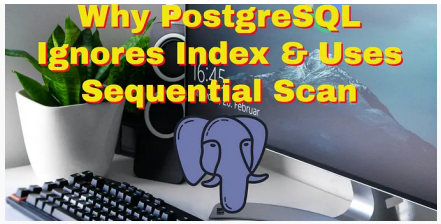


Ibrahim Salami (Ibby)

10 Powerful Ways to Filter Your Data Like a Pro in Python Pandas

Photo by Howen on Unsplash

Oct 1 1



Arunangshu Das

5 Key Differences: Embedding vs Referencing in MongoDB

When working with MongoDB, one of the most critical architectural decisions you'll...

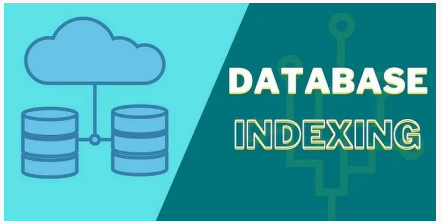
May 7 3

In Towards Dev by Nakul Mitra

Why PostgreSQL Uses Sequential Scan Even If Index Exists

PostgreSQL is known for its intelligent and cost-based query planner. However, many...

May 12 33



Sanath Shetty

Understanding Tree-Based Database Indexes: B-Trees,...

Database indexing structures are foundational to database performance. The...

May 12 1

EveEveYe

Trie

A Trie also known as a digital tree or prefix tree, is a specialized search tree data...

Jul 26

See more recommendations