Working with Index:

B-TREE Indexing:

Introduction:
B-Tree is a self-balancing search tree. In most of the other self-balancing search trees (like AVL and Red-Black Trees), it is assumed that everything is in main memory. To understand the use of B-Trees, we must think of the huge amount of data that cannot fit in main memory. When the number of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to the main memory access time. The main idea of using B-Trees is to reduce the number of disk accesses. Most of the tree operations (search, insert, delete, max, min, ..etc) require O(h) disk accesses where h is the height of the tree. B-tree is a fat tree. The height of B-Trees is kept low by putting maximum possible keys in a B-Tree node. Generally, the B-Tree node size is kept equal to the disk block size. Since the height of the B-tree is low so total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree, Red-Black Tree, ..etc.
Time Complexity of B-Tree:

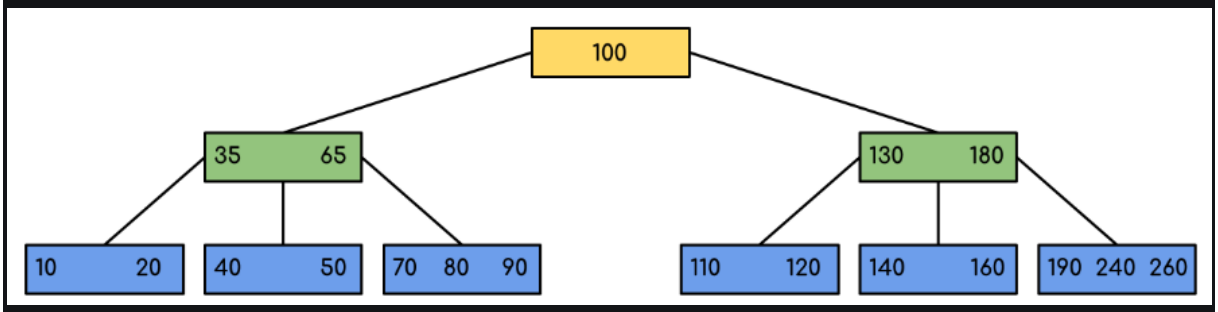
Sr. No.	Algorithm	Time Complexity
1.	Search	O(log n)
2.	Insert	O(log n)
3.	Delete	O(log n)

“n” is the total number of elements in the B-tree.
Properties of B-Tree:

1. All leaves are at the same level.
2. A B-Tree is defined by the term *minimum degree* ‘t’. The value of t depends upon disk block size.
3. Every node except root must contain at least t-1 keys. The root may contain minimum 1 key.
4. All nodes (including root) may contain at most 2*t – 1 keys.
5. Number of children of a node is equal to the number of keys in it plus 1.

6. All keys of a node are sorted in increasing order. The child between two keys k1 and k2 contains all keys in the range from k1 and k2.
7. B-Tree grows and shrinks from the root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.
8. Like other balanced Binary Search Trees, time complexity to search, insert and delete is O(log n).
9. Insertion of a Node in B-Tree happens only at Leaf Node.

Following is an example of B-Tree of minimum order 5. Note that in practical B-Trees, the value of the minimum order is much more than 5.



We can see in the above diagram that all the leaf nodes are at the same level and all non-leaf have no empty sub-tree and have keys one less than the number of their children.

Interesting Facts:

1. The minimum height of the B-Tree that can exist with n number of nodes and m is the maximum number of children of a node can have

   is:
2. The maximum height of the B-Tree that can exist with n number of nodes and t is the minimum number of children that a non-root node can have

   is: and

Traversal in B-Tree:

Traversal is also similar to Inorder traversal of Binary Tree. We start from the leftmost child, recursively print the leftmost child, then repeat the same process for remaining children and keys. In the end, recursively print the rightmost child.

Search Operation in B-Tree:

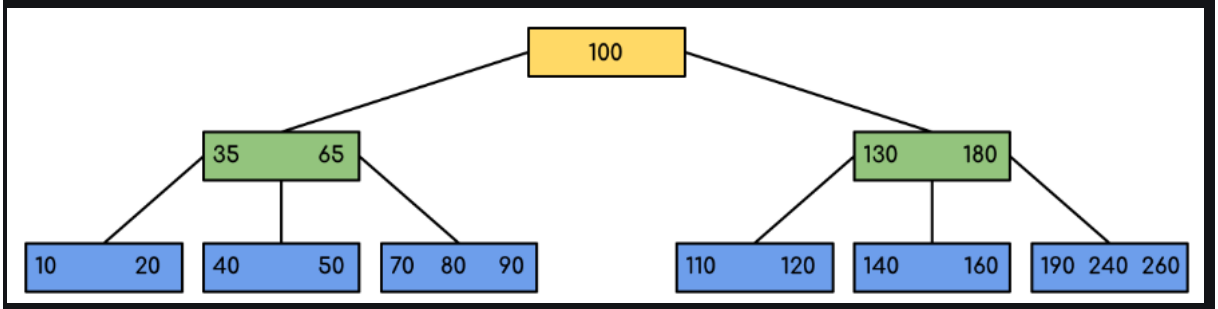
Search is similar to the search in Binary Search Tree. Let the key to be searched be k. We start from the root and recursively traverse down. For every visited non-leaf node, if the node has the key, we simply return the node. Otherwise, we recur down to the appropriate child (The child which is just before the first greater key) of the node. If we reach a leaf node and don't find k in the leaf node, we return NULL.

Logic:

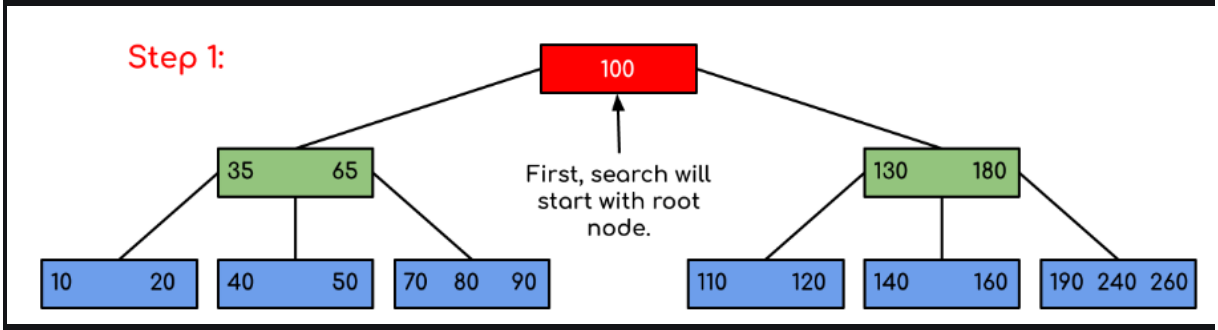
Searching a B-Tree is similar to searching a binary tree. The algorithm is

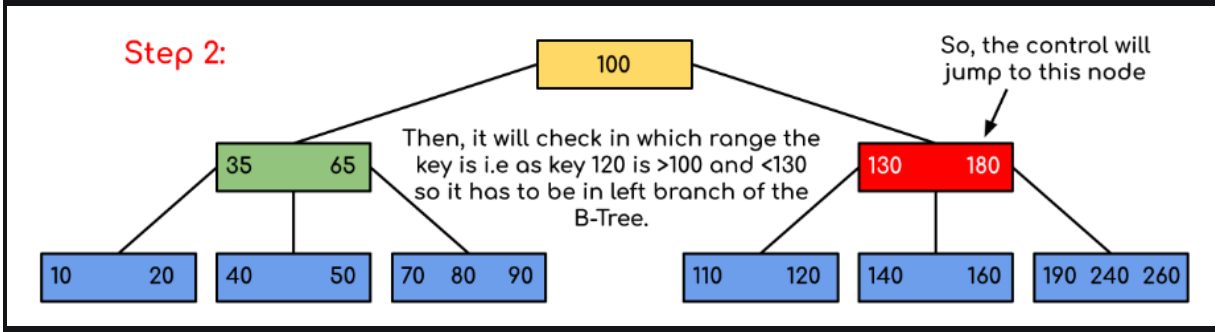
similar and goes with recursion. At each level, the search is optimized as if the key value is not present in the range of parent then the key is present in another branch. As these values limit the search they are also known as limiting value or separation value. If we reach a leaf node and don't find the desired key then it will display NULL.

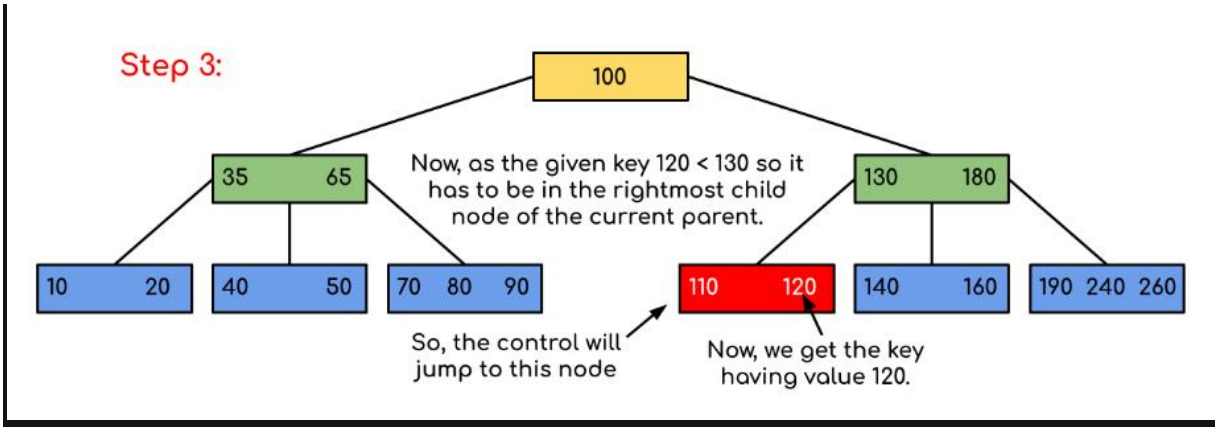
Example: Searching 120 in the given B-Tree.



Solution:







In this example, we can see that our search was reduced by just limiting the chances where the key containing the value could be present. Similarly if within the above example we've to look for 180, then the control will stop at step 2 because the program will find that the key 180 is present within the current node. And similarly, if it's to seek out 90 then as 90 < 100 so it'll go to the left subtree automatically and therefore the control flow will go similarly as shown within the above example.

Applications of B-Trees:

- It is used in large databases to access data stored in the disk
- Searching of data in a data set can be achieved in significantly less time using B tree
- With the indexing feature multilevel indexing can be achieved.
- Most of the servers also use B-tree approach.

An index is a data structure that allows us to add indexes in the existing table. It enables you to improve the faster retrieval of records on a database table. It creates an **entry** for each value of the indexed columns. We use it to quickly find the record without searching each row in a database table whenever the table is accessed. We can create an index by using one or more **columns** of the table for efficient access to the records.

When a table is created with a primary key or unique key, it automatically creates a special index named **PRIMARY**. We called this index as a **clustered index**. All indexes other than PRIMARY indexes are known as **a non-clustered** index or secondary index.

```
CREATE INDEX ind_1 ON t_index(col4);
```

By default, MySQL

Allowed index type **BTREE** if we have not specified the type of index. The following table shows the different types of an index based on the storage engine of the table.

SN	Storage Engine	Index Type
1.	InnoDB	BTREE
2.	Memory/Heap	HASH, BTREE
3.	MYISAM	BTREE

```
SELECT studentid, firstname, lastname FROM student WHERE class = 'CS';
```

EXPLAIN:

```
EXPLAIN SELECT studentid, firstname, lastname FROM student WHERE class = 'CS';

CREATE INDEX class ON student (class);

EXPLAIN SELECT studentid, firstname, lastname FROM student WHERE class = 'CS';

SHOW INDEXES FROM student;

DROP INDEX class ON student;

DROP INDEX PRIMARY ON student;
```

UNIQUE INDEX

```
CREATE UNIQUE INDEX index_name  ON table_name (index_column1, index_colu
mn2,...);
```

MySQL Clustered Index

An index is a separate data structure that allows us to add indexes in the existing table. It enables you to improve the faster retrieval of records on a database table. It creates an entry for each value of the indexed columns.

A clustered index is actually a table where the data for the rows are stored. It defines the order of the table data based on the key values that can be sorted in only one way. In the database, each table can have only one clustered index. In a relational database, if the table column contains a primary key or unique key, MySQL

allows you to create a clustered index named **PRIMARY** based on that specific column.

Characteristics

The essential characteristics of a clustered index are as follows:

- It helps us to store data and indexes at the same time.
- It stores data in only one way based on the key values.
- Key lookup.
- They are scan and index seek.
- Clustered index always use one or more column for creating an index.

Advantages

The main advantages of the clustered index are as follows:

- It helps us to maximize the cache hits and minimizes the page transfer.
- It is an ideal option for range or group with max, min, and count queries.
- At the start of the range, it uses a location mechanism for finding an index entry.

Disadvantages

The main disadvantages of the clustered index are as follows:

- It contains many insert records in a non-sequential order.
- It creates many constant page splits like data pages or index pages.
- It always takes a long time to update the records.
- It needs extra work for SQL queries, such as insert, updates, and deletes.

Clustered Index on InnoDB Tables

MySQL InnoDB table must have a clustered index. The InnoDB table uses a clustered index for optimizing the speed of most common lookups and DML (Data Manipulation Language) operations like INSERT, UPDATE, and DELETE command.

When the primary key is defined in an InnoDB table, MySQL always uses it as a clustered index named PRIMARY. If the table does not contain a primary key column, MySQL searches for the **unique key**. In the unique key, all columns are **NOT NULL** and use it as a clustered index. Sometimes, the table does not have a primary key nor unique key, then MySQL internally creates hidden clustered index **GEN_CLUST_INDEX** that contains the values of row id. Thus, there is only one clustered index in the InnoDB table.

The indexes other than the PRIMARY Indexes (clustered indexes) are known as a secondary index or non-clustered indexes. In the MySQL InnoDB tables, every record of the non-clustered index has primary key columns for both row and columns. MySQL uses this primary key value for searching a row in the clustered index or secondary index.

Difference between MySQL Clustered and Non-Clustered Index

The difference between clustered and non-clustered index is the most famous question in the database related interviews. Both indexes have the same physical structure and are stored as a BTREE structure in the MySQL server database. In this section, we are going to explain the most popular differences between them.

Indexing in MySQL is a process that helps us to return the requested data from the table very fast. If the table does not have an index, it scans the whole table for the requested data. MySQL

allows two different types of Indexing:

1. Clustered Index
2. Non-Clustered Index

Let us first discuss clustered and non-clustered indexing in brief.

What is a Clustered Index?

A clustered index is a table where the data for the rows are stored. It defines the order of the table data based on the key values that can be sorted in only one direction. In the database, each table can contains only one clustered index. In a relational database, if the table column contains a primary key or unique key, MySQL allows you to create a clustered index named **PRIMARY** based on that specific column.

```
CREATE TABLE Student
( post_id INT NOT NULL AUTO_INCREMENT, user_id INT NOT NULL,
 CONSTRAINT Post_PK
PRIMARY KEY (user_id, post_id),    //clustered index  CONSTRAINT post_id_UQ
   UNIQUE (post_id)
) ENGINE = InnoDB ;
```

Characteristics

Following are the essential characteristics of a clustered index:

- It enables us to store data and indexes together.
- It stores data in only one way based on the key values.
- Key lookup.
- It support index scan and index seek data operations.
- Clustered index always use one or more column for creating an index.

What is a Non-Clustered Index?

The indexes other than PRIMARY indexes (clustered indexes) called a non-clustered index. The non-clustered indexes are also known as secondary indexes. The non-clustered index and table data are both stored in different places. It is not able to sort (ordering) the table data. The non-clustered indexing is the same as a book where the content is written in one place, and the index is at a different place. MySQL allows a table to store one or more than one non-clustered index. The non-clustered indexing improves the performance of the queries which uses keys without assigning primary key.

Example

```
//It will create non-clustered index
CREATE NonClustered INDEX index_name ON table_name (column_name ASC);
```

Characteristics

Following are the essential characteristics of a non-clustered index:

- It stores only key values.
- It allows accessing secondary data that has pointers to the physical rows.
- It helps in the operation of an index scan and seeks.

- A table can contain one or more than one non-clustered index.
- The non-clustered index row stores the value of a non-clustered key and row locator.

Clustered VS Non-Clustered Index

Let us see some of the popular differences between clustered and non-clustered indexes through the tabular form:

Parameter	Clustered Index	Non-Clustered Index
Definition	A clustered index is a table where the data for the rows are stored. In a relational database, if the table column contains a primary key, MySQL automatically creates a clustered index named **PRIMARY**.	The indexes other than PRIMARY indexes (clustered indexes) called a non-clustered index. The non-clustered indexes are also known as secondary indexes.
Use for	It can be used to sort the record and store the index in physical memory.	It creates a logical ordering of data rows and uses pointers for accessing the physical data files.
Size	Its size is large.	Its size is small in comparison to a clustered index.
Data Accessing	It accesses the data very fast.	It has slower accessing power in comparison to the clustered index.
Storing Method	It stores records in the leaf node of an index.	It does not store records in the leaf node of an index that means it takes extra space for data.
Additional Disk Space	It does not require additional reports.	It requires an additional space to store the index separately.

Type of Key	It uses the primary key as a clustered index.	It can work with unique constraints that act as a composite key.
Contains in Table	A table can only one clustered index.	A table can contain one or more than a non-clustered index.
Index Id	A clustered index always contains an index id of 0.	A non-clustered index always contains an index id>0.

```
create table PAYMENT_4 as select * from PAYMENT_3;


CREATE TABLE `PAYMENT_4` (
  `PAYMENT_ID` int(11) NOT NULL,
  `staff_id` int(11) DEFAULT NULL,
  `rental_id` int(11) DEFAULT NULL,
  `amount` double DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
-- NO INDEXING
SELECT * FROM PAYMENT_4 WHERE rental_id=16049; --  4.578 sec / 0.000 sec
-- RENTALID--196
SELECT * FROM PAYMENT_4 WHERE STAFF_ID=2; -- 0.062 SEC
-- >50000
SELECT * FROM PAYMENT_4 WHERE rental_id=16049 AND STAFF_ID=2; -- 4.609 sec / 0.000 sec
```

- ADD INDIVIDAUAL INDEXING

```
ALTER TABLE PAYMENT_4 ADD INDEX(rental_id);
ALTER TABLE PAYMENT_4 ADD INDEX(STAFF_ID);
SELECT * FROM PAYMENT_4 WHERE rental_id=16049 AND STAFF_ID=2; -- 1.906 sec / 0.000 sec
```

EXPLAiN SELECT * FROM PAYMENT_4 WHERE rental_id=16049 AND STAFF_ID=2;

- ADD COMPOSITE INDEX

ALTER TABLE PAYMENT_4 ADD INDEX(rental_id,STAFF_ID);

SELECT * FROM PAYMENT_4 WHERE rental_id=16049 AND STAFF_ID=2; -- 1.985 sec / 0.000 sec

EXPLAiN SELECT * FROM PAYMENT_4 WHERE rental_id=16049 AND STAFF_ID=2;

- DROP COMPOSITE INDEX

ALTER TABLE PAYMENT_4 DROP INDEX rental_id;

ALTER TABLE PAYMENT_4 DROP INDEX staff_id;

SELECT * FROM PAYMENT_4 WHERE rental_id=16049 AND STAFF_ID=2; -- 1.329 sec / 0.000 sec

EXPLAiN SELECT * FROM PAYMENT_4 WHERE rental_id=16049 AND STAFF_ID=2;

SELECT * FROM ORDERS O

JOIN CUSTOMERS C ON C.CUSTOMERID=O.c

JOIN EMPLOYEES E ON E.EMPLOYEEID=O.EMPLOYEEID; -- 0.203 sec

EXPLAIN SELECT * FROM ORDERS O

JOIN CUSTOMERS C ON C.CUSTOMERID=O.CUSTOMERID

JOIN EMPLOYEES E ON E.EMPLOYEEID=O.EMPLOYEEID

JOIN shippers S ON S.ShipperID=O.ShipperID

where E.FirstName=10;

alter table EMPLOYEES add index idx_FirstName(FirstName);

```
-- optimizer --

alter table orders drop   key fk_CustomerID_Customers;

alter table orders drop   key fk_EmployeeID_Employees;

alter table orders add   index IDX_CustomerID_EmployeeID(CUSTOMERID,EmployeeID);

alter table orders drop   key fk_EmployeeID_Employees;



-- WHERE CLUASE COLUMNS

-- JOIN CLAUSE

SELECT * FROM training.customers ;

-- single

update customers set Country='japan' where CustomerID=1;

-- DML with joins- insert, update, delete

-- UPDATE WITH JOINS

SELECT C.CUSTOMERID,C.CustomerName,C.ContactName,O.OrderID

FROM customers C JOIN

ORDERS O ON O.CUSTOMERID=C.CustomerID;

-- UPDATE FOR OrderID 10400, UPDATE CustomerName TO 'JON' FROM ''Eastern Connection'

UPDATE Customers C

JOIN
```

```
ORDERS O ON O.CUSTOMERID=C.CustomerID

AND OrderID='10400'

SET C.CustomerName='JON';


-- DELETE WITH JOINS


DELETE C FROM Customers C

JOIN

ORDERS O ON O.CUSTOMERID=C.CustomerID

AND OrderID='10400';

select * from orders where OrderID=10400;


-- create table with  select

create table test123 as

SELECT C.CUSTOMERID,C.CustomerName,C.ContactName,O.OrderID

FROM customers C JOIN

ORDERS O ON O.CUSTOMERID=C.CustomerID;


select * from test123;


CREATE TABLE `test123` (

  `CUSTOMERID` int(11) NOT NULL,

  `CustomerName` varchar(255) DEFAULT NULL,

  `ContactName` varchar(255) DEFAULT NULL,

  `OrderID` int(11) NOT NULL

) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

```
-- taking of table

create table test12345 as select * from test123;
select * from test12345;

-- tacking backup of table
-- customer table
delete from customers where CustomerID=2;
delete from customers;
create table customers_bkp as select * from customers;
select * from customers_bkp;
select * from customers;

CREATE TABLE `customers` (
  `CustomerID` int(11) NOT NULL,
  `CustomerName` varchar(255) DEFAULT NULL,
  `ContactName` varchar(255) DEFAULT NULL,
  `Address` varchar(255) DEFAULT NULL,
  `City` varchar(255) DEFAULT NULL,
  `PostalCode` varchar(255) DEFAULT NULL,
  `Country` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`CustomerID`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

CREATE TABLE `customers_bkp` (
  `CustomerID` int(11) NOT NULL,
  `CustomerName` varchar(255) DEFAULT NULL,
```

```
  `ContactName` varchar(255) DEFAULT NULL,
  `Address` varchar(255) DEFAULT NULL,
  `City` varchar(255) DEFAULT NULL,
  `PostalCode` varchar(255) DEFAULT NULL,
  `Country` varchar(255) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

create table test12345 select * from test123;

create table customers_empty as select * from customers where customerid=0;
select * from customers_empty;

-- insert into table from another select

insert into customers_empty select * from customers;
select * from customers_empty;

create table test12334534 as
SELECT C.CUSTOMERID,C.CustomerName,C.ContactName,O.OrderID
FROM customers C JOIN
ORDERS O ON O.CUSTOMERID=C.CustomerID and C.CustomerID=0;

insert test12334534 SELECT C.CUSTOMERID,C.CustomerName,C.ContactName,O.OrderID
FROM customers C JOIN
ORDERS O ON O.CUSTOMERID=C.CustomerID;

-- renaming table
```

```
rename table customers_bkp to customers_bkp2;


-- subquery

select * from customers; -- RRRRR

select * from orders WHERE CustomerID

IN (SELECT CustomerID FROM CustomerS WHERE CustomerNAME='Franchi S.p.A.');


-- NOT TO RECOMMEND TO USE SUBQUERY

SELECT O.* FROM CustomerS C JOIN ORDERS O ON

O.CUSTOMERID=C.CUSTOMERID WHERE C.CUSTOMERNAME='Franchi S.p.A.';
```