B-TREE IMPLEMENTATION

Relational Database Management Systems
IA-2

Name & Roll no:

Shreya Menon	**16010123324**
Shreyans Tatiya	**16010123325**
Siddhant Raut	**16010123331**

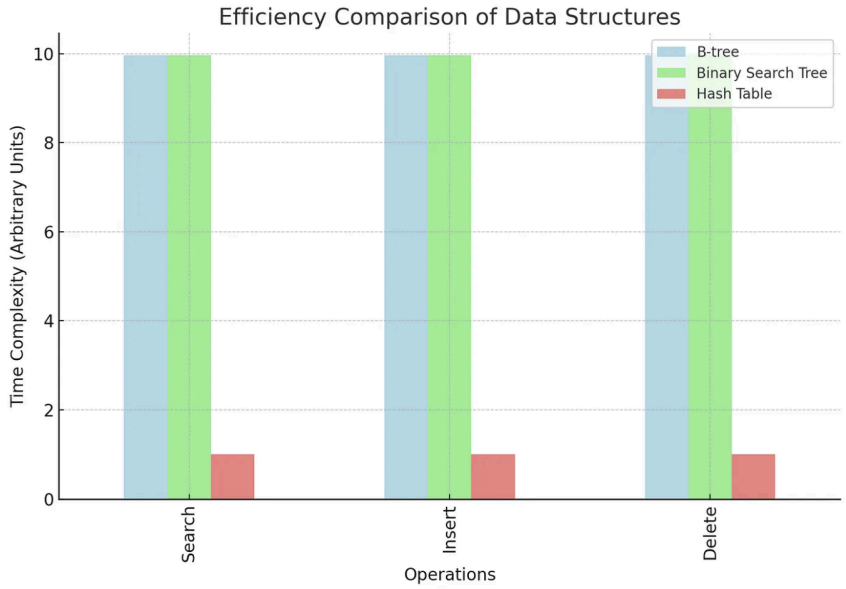
Div: E-2
Date of Submission: 12-04-2025

Introduction

In this study, B-Tree-based indexing algorithms for relational databases are implemented from start to finish and their performance is assessed. For large datasets in particular, indexing is essential to query processing efficiency since it minimizes disk I/O operations and data access times. The main goal of the project is to create a unique B-Tree structure from the ground up and use it as the main index to effectively retrieve data from a database simulation.

Since hashing, indexing, and query optimization are popular DBMS concepts that are crucial for both academic and practical work, this topic was selected.

Applications with high-frequency insertions, deletions, and lookups can benefit from B-Trees' ability to store sorted data and allow logarithmic-time search operations, as demonstrated by the implementation. To measure the advantages of indexing, the project compares query performance of B-Tree-indexed queries against full table scans (SQL-only lookups), assessing response time for a variety of search operations.



Indexing and Querying in DBMS

Indexing

It is a powerful technique used in databases to speed up data retrieval. An index is a separate data structure (like a B-Tree) that helps locate records quickly without scanning the entire table.

Without indexing, a database performs a linear scan checking every row to find matches. With indexing, the data is organized in a way (e.g., sorted or hashed) that allows for faster lookups, usually in O(log n) time.

Example:

Unindexed Employee Table

employee_id	work_hours	hourly_rate
10	12	1.15
14	18	1.31
18	18	1.34
12	12	1.05
18	6	1.34
20	6	1.31

Indexed Table (Sorted on employee_id)

employee_id	work_hours	hourly_rate
10	12	1.15
12	12	1.05
14	18	1.31
18	**18**	**1.34**
18	**6**	**1.34**
20	6	1.31

Querying

It refers to the process of requesting information from a database using a structured language like SQL. Queries can range from simple data retrieval to complex operations involving conditions, aggregations, and joins across multiple tables.

The **Query Processor** in a DBMS is responsible for parsing SQL commands, optimizing execution plans, and fetching results in the most efficient way possible.

One of the key factors that affects query performance is the presence or absence of indexes on the queried columns.

Example:

In the unindexed table, the rows are stored in no particular order. So, when we run a query like:

SELECT * FROM employees_earning WHERE employee_id = 18;

The database must scan each row from top to bottom, checking every employee_id to find all matches. Even though there are only two rows with employee_id = 18, all six rows are checked. This is called a full table scan, and it becomes inefficient as the table size increases.

In contrast, the indexed table is organized (or logically arranged) based on the employee_id. With an index in place, the database can quickly jump to the first occurrence of 18 and stop scanning as soon as the values no longer match. This drastically reduces the number of rows the database has to check, making the query faster and more efficient, especially with larger datasets.

B-Tree

A B-Tree is a specialized data structure designed for the efficient storage and management of extensive data sets. It is referred to as a "tree" due to its resemblance to an inverted tree with branches.

In this structure, each component is known as a node, and each node can possess multiple child nodes. This configuration aids in maintaining the tree's balance, preventing it from becoming excessively tall and narrow or overly wide and short.

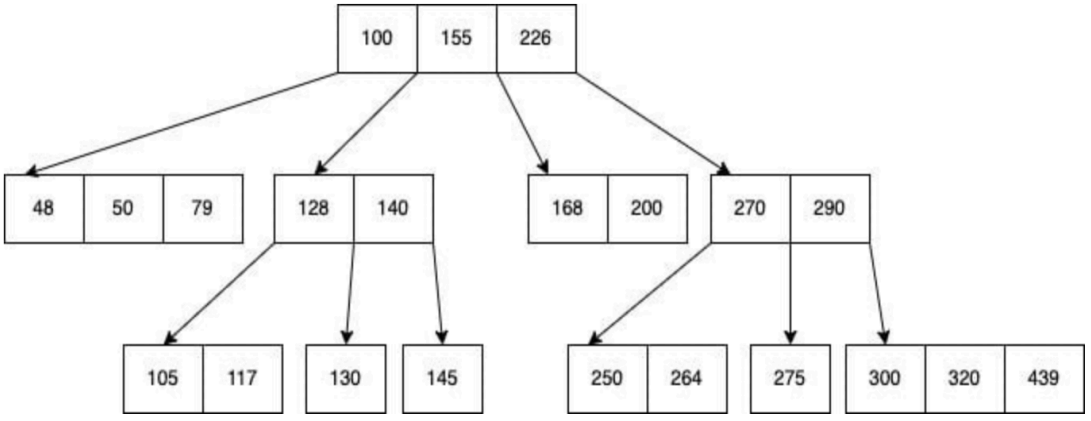
Key characteristics of a B-Tree include:

- Each node can accommodate several values.

- Each node can have numerous children.

- The tree maintains its balance by ensuring that all leaf nodes (the nodes located at the bottom) are aligned at the same level.

The primary purpose of a B-Tree is to organize data in a manner that facilitates quick retrieval, addition, or deletion of information. This efficiency is particularly crucial when handling large volumes of data, such as in databases or file systems.



B-Tree Indexing in DBMS

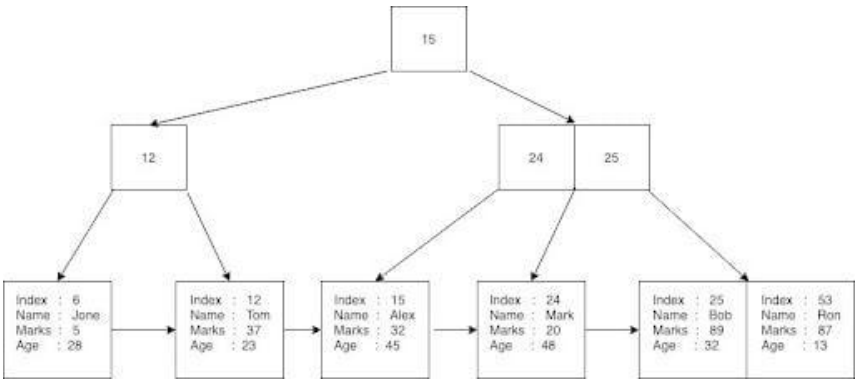
In database systems, query performance efficiency relies to a great degree on disk input/output (I/O) operations. particularly where the quantity of data is too great to fit into available memory. Without indexing, the Database Management System (DBMS) has to execute full table scans, i.e., O(n) disk I/O operations. The use of B-Tree indexing eliminates this issue by storing data in a balanced and sorted manner, thereby making search operations involve only O(log n) I/Os. Every node in a B-Tree can store more than one key in a single block, which means that typically, queries will require only 3 to 4 disk accesses, greatly enhancing efficiency when dealing with large volumes of data.

In database indexing, B-tree data structure is more complicated because, in addition to a key and a value, it also includes the value pertaining to the key. The value is utilized as a pointer for the data record. Key and value are treated as a payload.

If this table needs to be stored in a database:

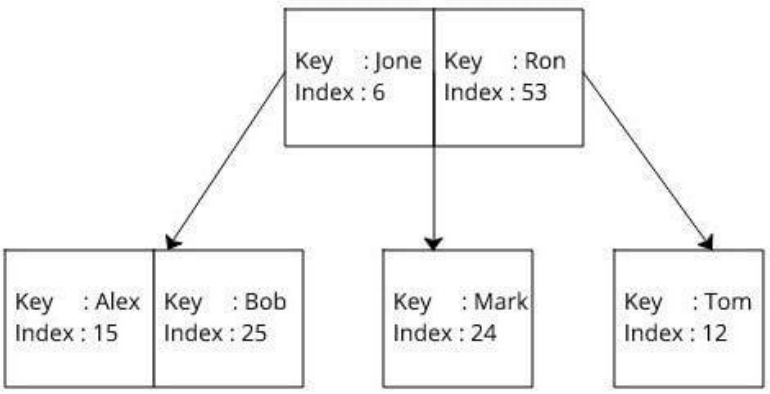
Name	Age	Mark
Tom	23	37
Bob	32	89
Mark	48	20
Jone	28	5
Ron	13	87
Alex	45	32

When a database is initialized, it assigns a distinct randomly generated primary key to each entry. These entries are then transformed into byte streams and stored in a B+ tree structure, with the key paired alongside its corresponding byte stream. This combination is referred to as the payload.



The B+ tree uses these keys as indexing references, organizing and storing all complete data records within its leaf nodes only. Non-leaf (internal) nodes function solely as routing paths and do not hold actual data records. Additionally, the leaf nodes are linked together in sequence to allow efficient traversal.

With indexing in place, the database can either search directly using the B+ tree index or perform a sequential scan by moving through the linked leaf nodes. In contrast, without indexing, the system is forced to examine every record one by one to locate a match, which is far less efficient. Typically, databases maintain multiple B-trees—one per indexed column. In this context, the "key" represents the value used for indexing within the tree, and each index entry points to the location of the actual data row in the table.



Literature Survey

Indexing is very useful for improving the performance of query in the database systems by reducing the disk I/O operations. B+ trees can easily maintain the order of a data set and yield faster insertion, deletion, and sorting operations. Hence, B+ trees are widely used.

B+ trees are well studied and widely applied in today's database management systems. They keep data sorted and permit searches, insertions, and deletions to be performed in logarithmic time—tasks for which one would normally expect linear time in a real-time system. Modern B+ tree implementations have various enhancements to their basic structure that allow even more efficient performance. For example, Graefe (2009) describes several features found in B+ trees used within real DBMSs that enhance their efficiency beyond the logarithmic guarantee.

In a distributed environment, Tadepalli and Cunningham (2009) present the idea of incrementally distributed B+ trees. Their study looks at scaling indexing to multiple nodes, and shows how B+ trees can be replicated and distributed over a grid to serve parallel, high-rate query processing. By using some mixture of the arbitrary replication of root and interior nodes, lazy updates, and lock coupling, they manage to achieve consistency under an optimization regime that makes it possible to maintain a very high degree of concurrency in the large-scale data environment.

B+ trees excel over hash-based indexes for scientific database queries and spatial data. Batory (1983) observed that B+ trees and indexed sequential files are comparable in performance for retrievals and updates. With respect to performance when it comes to the balance of retrievals and updates versus insertions and deletions, B+ trees have a significant edge. While insertions and deletions can be frequent and are often very necessary, the B+ tree has a much steadier, much more reliable, much quieter performance level. That means it can handle more data in a more reliable, quieter way.

Implementation

1. B-Tree Indexing

Python implementation of a minimum degree 3 B-Tree was created using Node and Tree classes. Nodes contain keys in sorted form and support node splitting, insertions, and recursive search. The tree is self-balancing, which makes key lookups possible in O(log n) time. The structure mimics the behavior of a database index to speed up access to records.

2. Database Setup

A local SQLite database (records.db) was established with a table of people(id, name) and filled with 10,000 records, each with a distinct ID and a randomly chosen name.

3. Index Building

All id values from the table were loaded into the B-Tree to mimic a primary index. This in-memory data structure enabled quick existence checks prior to retrieving the actual record from the database.

4. Search and Performance Testing

Two search methods were tested:

B-Tree + DB: Search ID via the B-Tree, then SQL fetch.

SQL Only: Simple SQL query without indexing.

100 random IDs were utilized to test performance. Query times were measured and compared.

5. Visualization

A graph was created to compare search times. Results indicated that the B-Tree method consistently took less time than SQL-only searches, particularly at scale.

Code

```
import sqlite3

import random

import string

import time

import matplotlib.pyplot as plt

class BTreeNode:

   def __init__(self, degree, is_leaf=True):

       self.degree = degree

       self.keys = []

       self.children = []

       self.is_leaf = is_leaf



   def insert_non_full(self, key):

       idx = len(self.keys) - 1

       if self.is_leaf:

           self.keys.append(0)

           while idx >= 0 and self.keys[idx] > key:

               self.keys[idx + 1] = self.keys[idx]

               idx -= 1

           self.keys[idx + 1] = key

       else:

           while idx >= 0 and key < self.keys[idx]:

               idx -= 1

           idx += 1

           if len(self.children[idx].keys) == 2 * self.degree - 1:
```

```
            self.split_child(idx)
            if key > self.keys[idx]:
                idx += 1
        self.children[idx].insert_non_full(key)

def split_child(self, idx):
    degree = self.degree
    y = self.children[idx]
    z = BTreeNode(degree, y.is_leaf)
    z.keys = y.keys[degree:]
    y.keys = y.keys[:degree - 1]
    if not y.is_leaf:
        z.children = y.children[degree:]
        y.children = y.children[:degree]
    self.children.insert(idx + 1, z)
    self.keys.insert(idx, y.keys.pop())

def contains(self, key):
    i = 0
    while i < len(self.keys) and key > self.keys[i]:
        i += 1
    if i < len(self.keys) and self.keys[i] == key:
        return True
    if self.is_leaf:
        return False
    return self.children[i].contains(key)
```

```
class BTree:

    def __init__(self, degree):

        self.root = BTreeNode(degree)

        self.degree = degree



    def insert_key(self, key):

        root_node = self.root

        if len(root_node.keys) == 2 * self.degree - 1:

            new_root = BTreeNode(self.degree, False)

            new_root.children.insert(0, root_node)

            new_root.split_child(0)

            i = 0

            if key > new_root.keys[0]:

                i = 1

            new_root.children[i].insert_non_full(key)

            self.root = new_root

        else:

            root_node.insert_non_full(key)



    def search_key(self, key):

        return self.root.contains(key)



def generate_random_names(count):

    return [''.join(random.choices(string.ascii_lowercase, k=6)) for _ in
range(count)]
```

```
def initialize_database():

    connection = sqlite3.connect("records.db")

    cursor = connection.cursor()

    cursor.execute("DROP TABLE IF EXISTS people")

    cursor.execute("CREATE TABLE people (id INTEGER PRIMARY KEY, name TEXT)")

    names_list = generate_random_names(10000)

    for idx, name in enumerate(names_list, start=1):

        cursor.execute("INSERT INTO people VALUES (?, ?)", (idx, name))

    connection.commit()

    connection.close()

def populate_btree_from_db():

    connection = sqlite3.connect("records.db")

    cursor = connection.cursor()

    cursor.execute("SELECT id FROM people")

    all_ids = [row[0] for row in cursor.fetchall()]

    connection.close()



    btree = BTree(3)

    for record_id in all_ids:

        btree.insert_key(record_id)

    return btree



def search_using_tree_and_db(btree, target_id):

    if btree.search_key(target_id):

        connection = sqlite3.connect("records.db")

        cursor = connection.cursor()
```

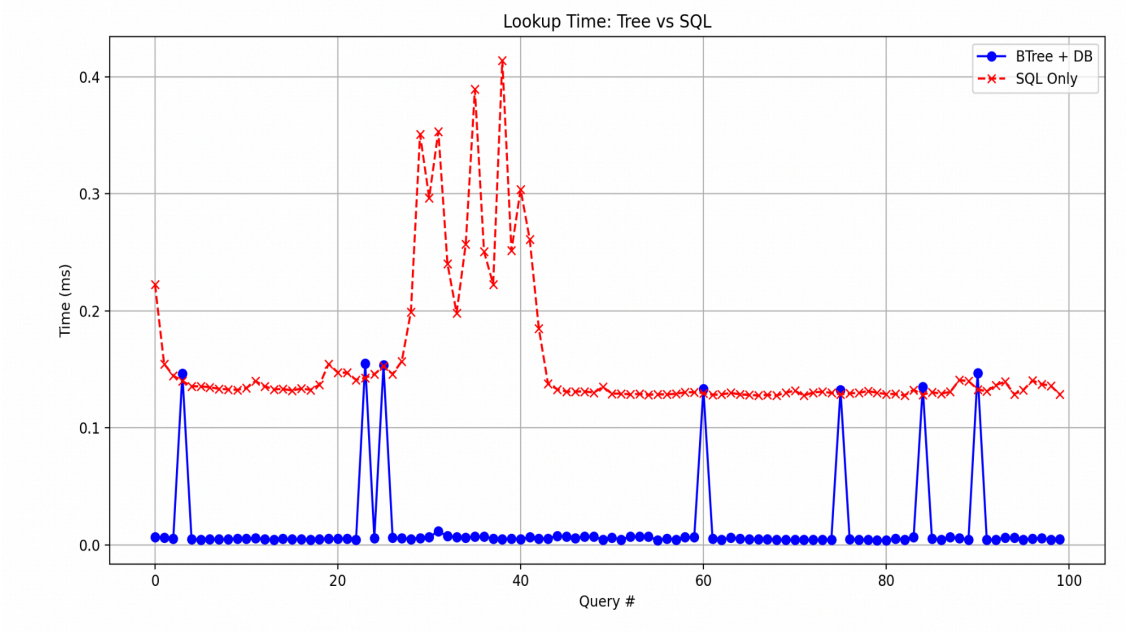
```
        cursor.execute("SELECT * FROM people WHERE id=?", (target_id,))
        result = cursor.fetchone()
        connection.close()
        return result
    return None

def search_using_sql_only(target_id):
    connection = sqlite3.connect("records.db")
    cursor = connection.cursor()
    cursor.execute("SELECT * FROM people WHERE id=?", (target_id,))
    result = cursor.fetchone()
    connection.close()
    return result

if __name__ == "__main__":
    print("Setting up database...")
    initialize_database()

    print("Building B-Tree from database IDs...")
    tree_structure = populate_btree_from_db()

    print("Comparing search performance: B-Tree + DB vs SQL only...")
    query_ids = random.sample(range(1, 10001), 100)
    btree_timings = []
    sql_timings = []
```

```
    for index, value in enumerate(query_ids, start=1):

        if index % 10 == 0:

            print(f"Processed {index} queries...")



        # B-Tree + DB Timing

        start_time = time.perf_counter()

        _ = search_using_tree_and_db(tree_structure, value)

        end_time = time.perf_counter()

        btree_timings.append((end_time - start_time) * 1000)



        # SQL Only Timing

        start_sql = time.perf_counter()

        _ = search_using_sql_only(value)

        end_sql = time.perf_counter()

        sql_timings.append((end_sql - start_sql) * 1000)



    avg_btree_time = sum(btree_timings) / len(btree_timings)

    avg_sql_time = sum(sql_timings) / len(sql_timings)



    print("\nComparison complete.")

    print(f"Average B-Tree + DB Search Time: {avg_btree_time:.4f} ms")

    print(f"Average SQL-only Search Time:    {avg_sql_time:.4f} ms")



    print("Generating performance graph...")

    plt.plot(btree_timings, label='B-Tree + DB', marker='o', linestyle='-',
color='blue')
```

```
plt.plot(sql_timings, label='SQL Only', marker='x', linestyle='--', color='red')

plt.xlabel("Query Index")

plt.ylabel("Time (ms)")

plt.title("Search Time: B-Tree + DB vs SQL Only")

plt.legend()

plt.grid(True)

plt.tight_layout()

plt.savefig("btree_vs_sql_performance.png")

plt.show()

print("Graph saved and displayed.")
```

Output

```
shreya@Shreyas-MacBook-Air-2 rdbms ia % python3 -u "/Users/shreya/Desktop/rdbms ia/btree_analysis.py"
Setting up database...
Building B-Tree from database IDs...
Comparing search performance: B-Tree + DB vs SQL only...
Processed 10 queries...
Processed 20 queries...
Processed 30 queries...
Processed 40 queries...
Processed 50 queries...
Processed 60 queries...
Processed 70 queries...
Processed 80 queries...
Processed 90 queries...
Processed 100 queries...

Comparison complete.
Average B-Tree + DB Search Time: 0.0180 ms
Average SQL-only Search Time:    0.1427 ms
Generating performance graph...
2025-04-12 11:48:29.931 python3[36083:5827093] +[IMKClient subclass]: chose IMKClient_Modern
2025-04-12 11:48:29.932 python3[36083:5827093] +[IMKInputSession subclass]: chose IMKInputSession_Modern
Graph saved and displayed.
shreya@Shreyas-MacBook-Air-2 rdbms ia %
```



The graph illustrates the lookup time for 100 queries against a B-Tree indexed database compared to SQL-only search.

B-Tree + DB (blue): The lookup times are always very low, near 0 ms for the majority of queries, with only a few small spikes. This indicates high efficiency and stability of performance.

SQL Only (red): Times are considerably larger overall, particularly around queries 30–45, where several spikes are as high as 0.42 ms. This implies poor consistency, probably due to higher search complexity as the dataset gets larger

The B-Tree index hence significantly enhances lookup time and consistency, particularly evident as the queries increase in number.

Conclusion

A significant advancement in data structures, B-Trees overcome the drawbacks of binary search trees in managing big datasets, particularly when disc latency is an issue. B-Trees maintain their efficiency and balance in contrast to binary trees, which can become unbalanced. Depending on the tree's order, their nodes, which are m-ary trees, can hold several keys and children.

Key features include: all leaves at the same depth, and internal nodes (except the root) having at least $\lceil m/2 \rceil$ children to maintain balance. Keys are sorted, and searches take $O(\log n)$ time. Insertions and deletions maintain balance via node splitting and merging.

B-Trees' primary goal is to minimise disc I/O and tree height. Even for millions of records, B-Trees remain shallow due to the placement of children between keys and the multiple keys per node, which results in predictable, slow access times. They are therefore perfect for secondary storage devices like SSDs and hard drives, aligning well with data stored in disk blocks or pages. As a result, file systems and database indexing make extensive use of them.

B-Trees are highly scalable, unlike binary trees designed for small in-memory sets. That’s why relational databases (like MySQL, Oracle, PostgreSQL) and many NoSQL databases use B-Trees or B+ Trees for indexing. They also support various traversal methods, useful for range queries and ordered operations.

In short, B-Trees are powerful, practical tools for efficient data organization and retrieval—an essential concept for computer science and modern data management.

References

- **Goetz Graefe (2011), "Modern B-Tree Techniques"**
- **Incrementally distributed B+ trees: approaches and challenges, Pallavi Tadepalli and H. Conrad Cunningham**
- **Indexing Essentials in SQL | Atlassian**
- **10.3.1 How MySQL Uses Indexes**
- **B+ Trees and Indexed Sequential Files: A Performance Comparison, D.S. Batory Computer and Information Sciences Department, University of Florida**
- **Silberschatz, A., Korth, H.F., & Sudarshan, S. (2019). "Database System Concepts" (7th ed.). McGraw-Hill Education.**
- **B-tree indexes for high update rates, Goetz Graefe**
- **Formulae for B tree and B+ tree in DB | Medium**