Project 2 Documentation

Project Overview

In this project, we focused on improving the performance of five key Relational Algebra (RA) operators in a Java 22 environment by using indexing techniques like BpTreeMap or B+Tree Index. The goal was to make these operators faster by using indices, similar to how a real database system works. The operators were implemented within a class structure, allowing us to simulate basic database operations without needing a full database management system.

Key Features

Operators

- Select
    - Implemented an indexed select to quickly find specific data. Instead of searching through the entire table, the index helps locate the data quickly by directly pointing to where the desired values are stored.
- Project
    - Used the index to be more efficient by using an index to check for duplicate rows. If a row (or tuple) already exists, it is skipped, and only unique rows are included in the final result, saving time and resources.
- Union
    - Used the index to efficiently check if a tuple is a duplicate and if so, don't include it in the returned table
- Minus
    - Used index to check for each tuple if it is in the original table, and if so don't return
- Join
    - Implemented indexed join to make the process of combining rows faster

Indexing Techniques

We used BpTreeMap, which helped us organize and access data faster. These indices helped speed up the relational algebra operations by reducing the need to scan through all the data.

BPTree Implementation

- AddI Method

    - This method adds a node to the tree and handles internal node overflows by invoking a **split** function. It is similar to the standard add method, but with a key difference: instead of performing a regular split, the node is split at the internal level, ensuring that the tree remains balanced as new nodes are added.
- Insert Method
    - The insert method begins by checking whether the node is null. If the node is a leaf, it is added in the usual manner. However, if the node is internal, the method recursively descends to ensure that the leaf nodes are correctly added to existing nodes.
    - Once the leaf nodes are handled, the method checks if the children of the current node are not null and if they correspond to the root at the internal level. If this condition is met, the root is updated with the value located at the midpoint of the current node, or with the new root value, to maintain the tree's structure.

Unique and Non-Unique Indexes

- Unique Index: The primary key of the table is made a unique index, by default. This means that no two rows in the table can have the same value for the primary key. The unique index ensures that every row is different and helps to quickly find or update rows based on this key. It also prevents duplicate entries for the primary key, ensuring data consistency.
- Non-Unique Index: Individual columns are made non-unique indexes for faster select queries. By organizing the data in a way that can be searched quickly, it speeds up operations when you're looking for rows based on the values in these columns, even if the values are repeated.

Create_index Method

The **create_index** method creates an **index** – **unique or non-unique -** for specified column(s) in a table, helping to speed up queries by making searches and lookups faster.

- It currently uses a **B+ Tree** structure, which organizes the data in a way that allows quick retrieval based on the values in the column. This speeds up operations like **SELECT** and **JOIN**.

- If it creates a **non-unique index**, duplicate values are allowed in the indexed column, meaning multiple rows can have the same value while still benefiting from faster query performance.
- If it creates a **unique index**, duplicate values are not allowed for the indexed column(s), meaning key values for unique indexes are never repeated.

Performance Testing

We tested the operators with and without indexing to see how much faster they performed with indices. The tests showed significant improvements, especially with larger datasets, by reducing the time needed to execute each operation.

Conclusion

This project successfully demonstrates the importance of optimizing relational algebra operators through the use of indexing techniques such as B+ Tree and BpTreeMap. By implementing and managing both unique and non-unique indexes, we improved the performance of essential database operations like **Select**, **Project**, **Union**, **Minus**, and **Join**. These optimizations not only enhanced query speed but also ensured the accurate and efficient handling of data. Overall, this project provides a solid foundation for understanding core database functions and lays the groundwork for exploring more advanced concepts in database management systems