DSA FINAL PROJECT REPORT

Session 2023 – 2027

Submitted by:

M Ahmed Butt	**2023-CS-18**
Abdul Rehman	**2023-CS-20**
Hamid Riaz	**2023-CS-10**

Submitted To:

Sir Nazeef-Ul-Haq

Department of Computer Science

University of Engineering and Technology

Lahore Pakistan

Table of Contents:

Table of Figures

1. Introduction

The **Database Project** is a lightweight, command-line database engine implemented in **C**. Designed to provide hands-on exposure to database concepts such as data storage, indexing, and query execution, this project demonstrates the fundamental principles of database management systems (DBMS). Drawing inspiration from SQLite, the implementation focuses on applying the **B$^+$-Tree** data structure for efficient data organization and retrieval. This project is specifically targeted at beginners to help them understand how databases work at a basic level.

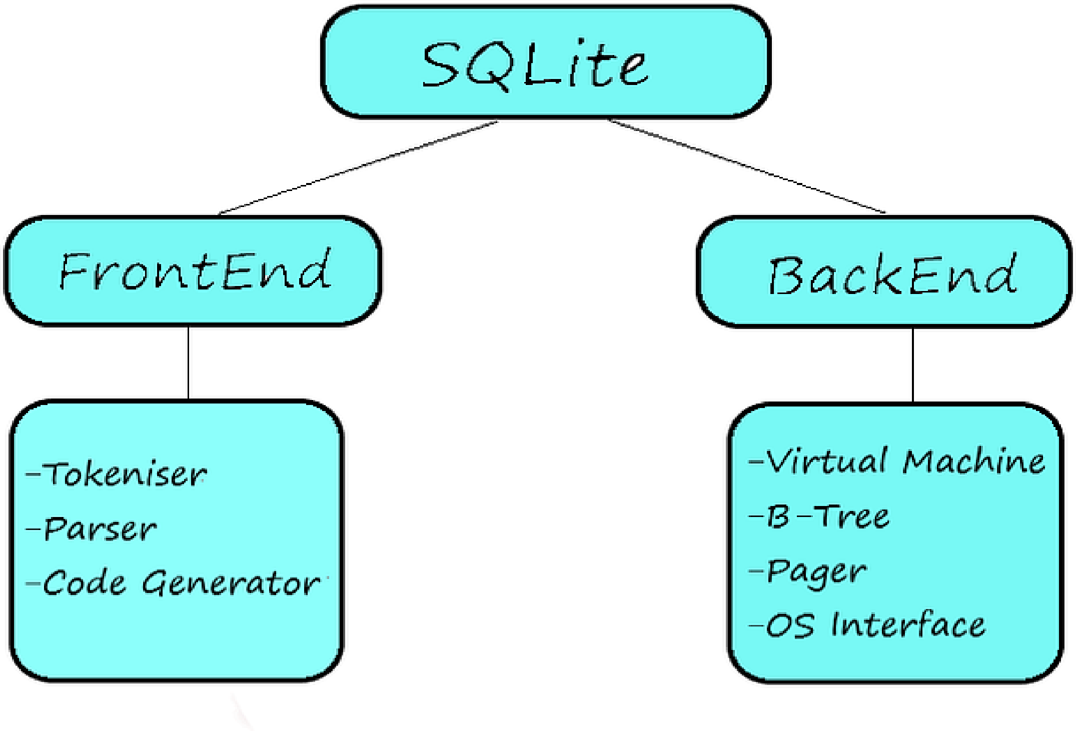

Figure 1: Project Structure

Beginner Notes:

- A **database management system (DBMS)** helps store, manage, and retrieve data efficiently.
- The **B$^+$-Tree** is used to keep data sorted and enables quick look ups.
- This project offers an opportunity to learn by building a mini version of a real-world database system.

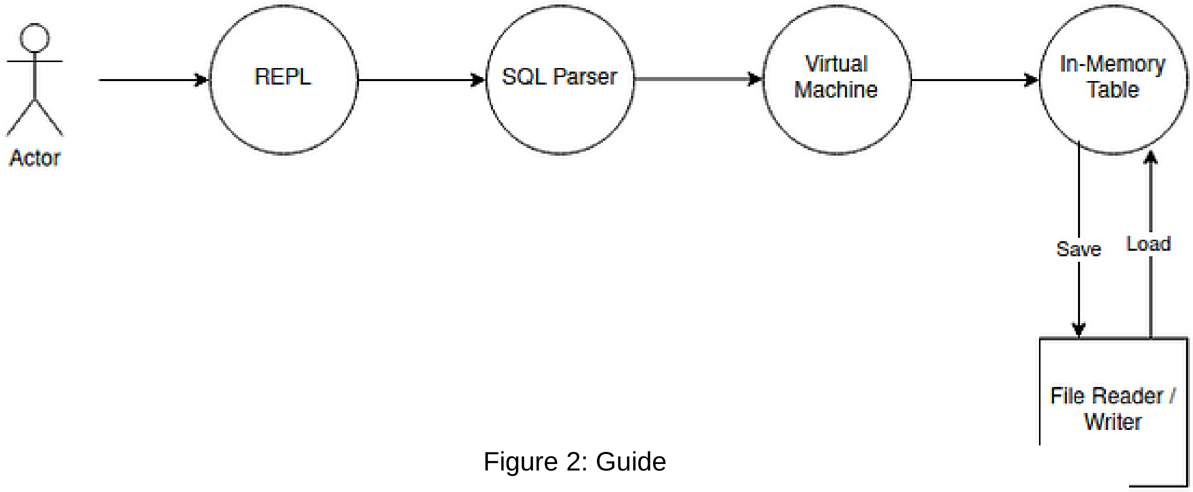

Figure 2: Guide

2. Project Objectives

2.1 Primary Objectives

- **Understand Core Concepts**: Learn how databases store and retrieve data.
- **Efficient Data Handling**: Implement data operations (insert, delete, update, and select) using optimized methods.
- **Indexing with B-Trees**: Use a data structure that balances speed and memory usage.

2.2 Secondary Objectives

- **Build an Interactive Shell**: Allow users to enter commands in real time.
- **Implement Persistent Storage**: Save data permanently using files.
- **Debugging Tools**: Add commands to inspect the database structure during development.

2.3 Beginner Notes

- Think of "objectives" as the main goals you aim to achieve by completing this project.
- By focusing on these, you'll understand how key database operations are designed.

3. Data Structures:

3.1. B+-Tree Data Structure

The **B$^+$-Tree** is a balanced tree data structure that maintains sorted data and allows for efficient insertion, deletion, and search operations, making it highly suitable for use in databases and file systems.

Why Use B-Trees?

- **Keeps Data Sorted:** The B+-Tree ensures that data is always stored in sorted order, which is essential for performing efficient range queries and searches.
- **Fast Search, Insert, and Delete Operations:** B+-Trees maintain balance, which allows for efficient searches, insertions, and deletions. The height of the tree is kept small, ensuring that operations remain logarithmic in complexity (O(log N)).
- **Scalable for Large Datasets**: B+-Trees handle large datasets efficiently. Even as the database grows, the tree remains balanced, allowing for consistent performance even with vast amounts of data.
- **Efficient Use of Disk I/O:** Since B+-Trees are designed to minimize the height of the tree, they optimize disk reads by storing multiple keys in each node. This is particularly useful in databases, where nodes are typically stored in disk blocks.
- **Automatic Rebalancing:** The B+-Tree automatically balances itself as data is inserted or deleted. When a node overflows, it splits; when it underflows, it merges. This ensures the tree remains balanced, minimizing search path lengths.
- Balances itself automatically during inserts and deletes.

Key Properties

Nodes Structure:

- **Internal Nodes:** These nodes contain only keys, which help in navigating the tree. Internal nodes do not store data but store references (pointers) to child nodes.

- **Leaf Nodes:** Leaf nodes contain both keys and actual data pointers (or data). In a B+-Tree, all data is stored in the leaf nodes, and the internal nodes only store keys to guide the search.
- **Sorted Keys:** Keys in each node are stored in sorted order. This sorting allows for efficient searching using binary search within each node.

Balanced Structure:

- **Self-Balancing:** A B+-Tree is a balanced tree, meaning all leaf nodes are at the same level. This ensures that all operations (search, insert, delete) are logarithmic in time complexity, which is key to performance.
- **Splitting and Merging:** As elements are inserted or deleted, nodes may split or merge to maintain balance:
- **Node Split:** When a node exceeds its maximum capacity, it splits into two, and a new key is inserted into the parent node.
- **Node Merge:** When a node becomes too sparse, it merges with its neighboring node, and the parent is adjusted accordingly.

Multi-Level Indexing:

- In a B+-Tree, there can be multiple levels of internal nodes that allow for efficient multi-level indexing. This results in fewer disk reads since multiple keys are stored in each node, reducing the number of levels the search algorithm must traverse.
- **Efficient Range Queries:**

- ➢ The B+-Tree structure allows for efficient range queries. Since all data is stored in the leaf nodes, traversing the leaf nodes sequentially allows easy retrieval of data within a range.

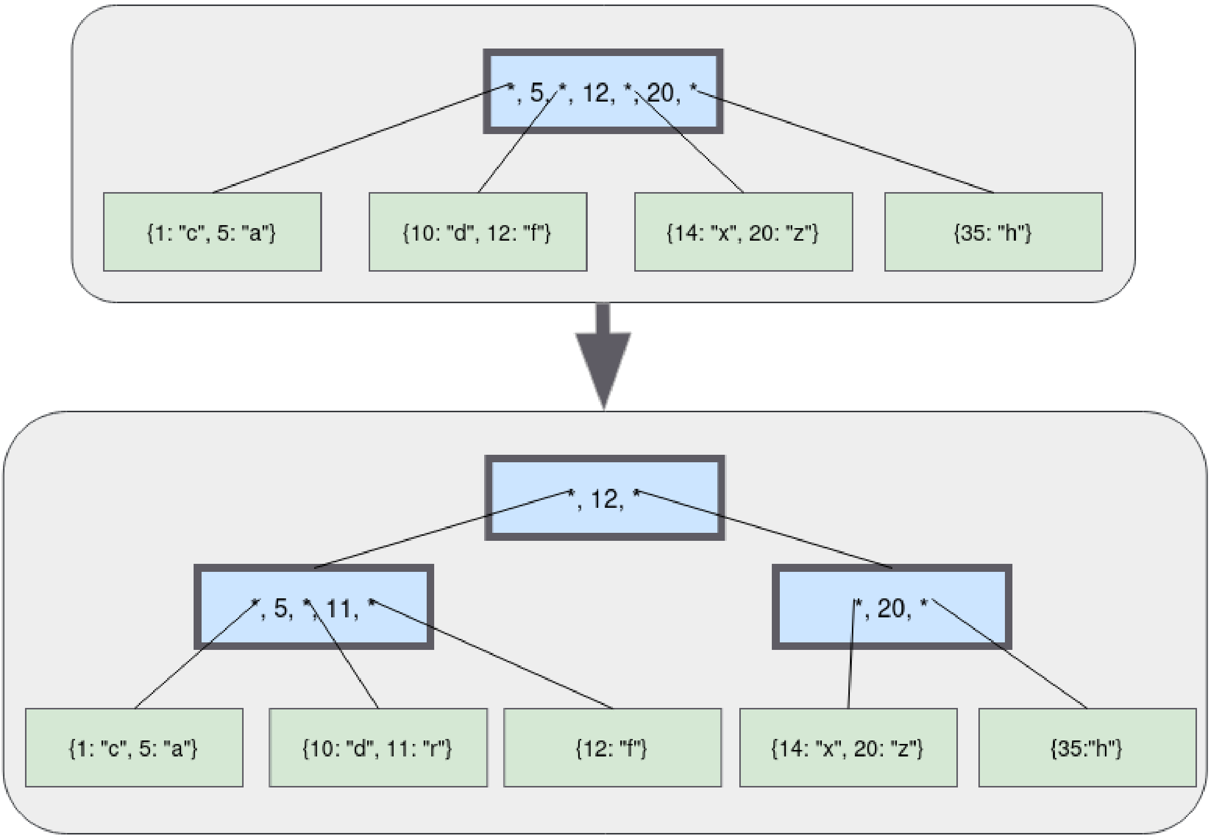

Figure 3: B+ Tree Working

3.1.2 B+ Tree Visualization:

3.1.2.1 Empty B+ Tree:

An empty B-tree has a single node: the root node. The root node starts as a leaf node with zero key/value pairs:

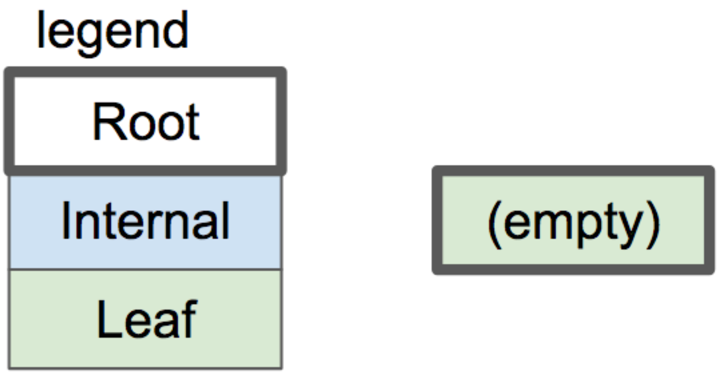

Figure 4: Empty B+ Tree

3.1.2.2 One Node B+ Tree:

One Node B^+ Tree has only root which contains both keys and values but does not contain any pointer

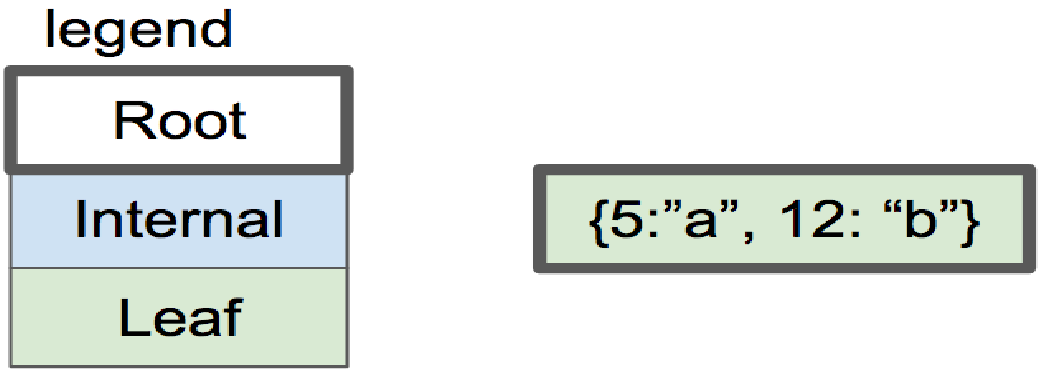

Figure 5: One Node B+ Tree

3.1.2.3 Two Level B+- tree:

- **Initial Structure**

➢ **Leaf Nodes:**

Each leaf node can hold a maximum of 2 key-value pairs. A key-value pair consists of a key and its associated data (or pointer to the data).

➢ **Internal Nodes:**

Internal nodes store keys that guide the search. They don’t store data directly, but point to child nodes.

- **Inserting Keys in Leaf Nodes**

➢ When inserting a new key-value pair, the insertion is straightforward as long as the leaf node is not full.

➢ If the leaf node is full (i.e., it already contains 2 key-value pairs), the node splits.

- **Splitting the Leaf Node**

➢ Upon splitting, the leaf node is divided into two separate leaf nodes, each holding 1 key-value pair.

➢ The middle key from the original leaf node is promoted to an internal node. This key will now act as a separator and will guide future searches.

➢ A new internal node is created to store the middle key, and it will point to the two newly split leaf nodes.

- **New Root Creation**

➢ If the tree didn’t have a root, the new internal node created from the split will become the root node of the tree.

- If there was already a root, the new internal node will be added as a child to the existing root.
- This results in the tree having two levels: the root pointing to an internal node, which in turn points to the leaf nodes.

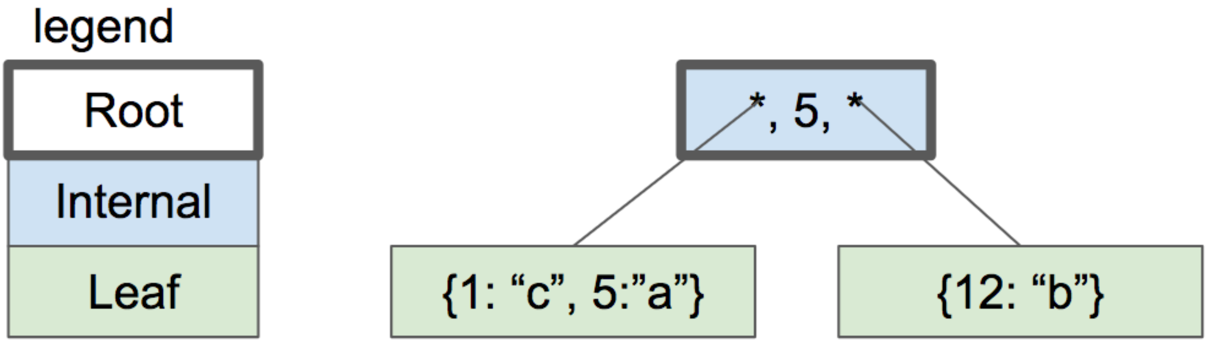

Figure 6: New Node Creation

3.2.4 Three Level B+ tree:

A Three-Level B+ Tree consists of three types of nodes: leaf nodes, internal nodes, and a root node. Each type of node plays a critical role in maintaining the structure and ensuring efficient insertion, deletion, and search operations.

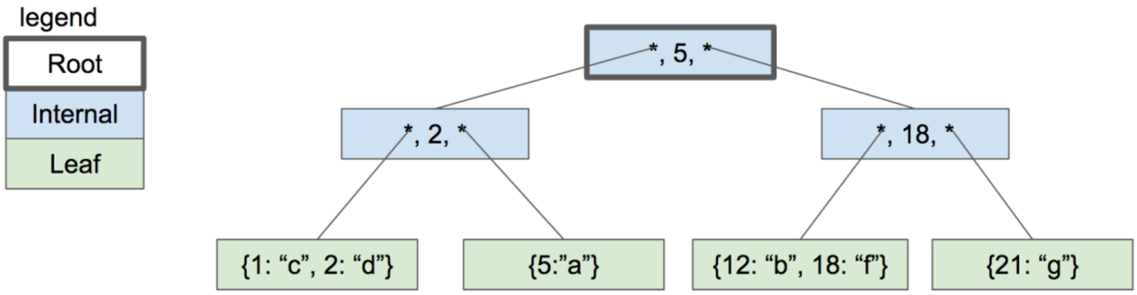

Figure 7: Tree Level B+ tree

Practical Steps:

1. **Understand Nodes**: A node is like a folder that contains sorted keys and pointers to other nodes.
2. **Learn Traversal**: Start at the root node, move down the tree by comparing keys.
3. **Implement Split Logic**: When a node is too full, split it into two and adjust the tree.

3.2 Stack:

Description:

- A **Stack** is a linear data structure that follows the **Last In, First out (LIFO)** principle. The last element added is the first one to be removed.
- Common applications: recursion (function calls), expression evaluation, and tree/graph traversal.

Role in the Project:

- **Iterative Tree Traversal:** The stack is useful for **iterative traversal** of the **B+-Tree**, especially for depth-first traversal. This avoids deep recursion, which could cause stack overflow for large trees.
- **Avoiding Recursion:** Instead of using recursive function calls, the stack maintains the state of the traversal manually.

Implementation:

The stack is implemented as a linked list of Stack Node structures, where each node contains:

- A pointer to the **data** (node in the B+-Tree).
- The **page number** to keep track of the location in the tree.
- The **level** of the node.
- A pointer to the **next node** in the stack.

3.3 Queue:

Description:

- A **Queue** is a linear data structure that follows the **First In, First out (FIFO)** principle. The first element added is the first one to be removed.
- Common applications: scheduling tasks, managing data buffers, and breadth-first search (BFS) in trees or graphs.

Role in the Project:

- **Level-Order Traversal:** The queue is well-suited for **level-order traversal** of the **B+-Tree**. This traversal processes nodes level by level, making it ideal for operations such as printing or displaying the tree structure.
- The queue ensures that nodes are processed in the correct order, which is essential for handling operations that require sequential processing.

Implementation:

- The queue is implemented using a linked list of Queue Node structures, where each node contains:

- A pointer to the **data** (node in the B+-Tree).
- The **page number** of the node for locating it in memory.
- The **level** of the node.
- A pointer to the **next node** in the queue.

3.4 Vectors:

Description:

- An **Input Buffer** is a temporary storage for user input. It holds the data entered by the user and allows the program to process it in chunks.
- This structure is commonly used in command-line interfaces and text-processing applications.

Role in the Project:

- **Command Processing:** The input buffer is used to read user commands from the command line interface. It ensures that user input is correctly managed before it's processed by the command processor**.**
- **Command Management:** It helps handle different commands and store them in a way that the program can later interpret or execute, such as adding data to the tree or performing a search operation.

Implementation:

The input buffer is implemented using the Input Buffer structure, which contains:

- A pointer to the buffer where the input string is stored.
- The **length** of the buffer to ensure it can handle larger inputs.
- The **input length** to track the actual length of user input.
- Functions can be provided to:
- Create and initialize a new input buffer.
- Read input from the user into the buffer.
- Free the buffer when it is no longer needed.

4. Technical Overview

This project integrates basic database functionalities with efficient data structures. Key components include:

1. Tokenizer:

The SQL statements are first sent to the tokenizer which as its name breaks the string into tokens which are passed to the parser for further action.

2. Parser:

The tokens are given meanings based on their contexts in the parser. It then generates a syntax tree which represents the parsed statements. This tree is then analyzed by the code generator.

3. Code Generator:

This is the component where the magic happens! The code generator analyzes the input representation of SQL text which is compiled by the back-end.

4. Virtual Machine:

This truly acts as the heart of this database. This component runs a program in virtual machine language that searches, reads, or modifies the database. The program begins when instruction is 0 and continues until any error or stop instruction is provided. After completion of execution of the program, all open database cursors are closed, memory is freed.

5. Each storage unit is a page of fixed length. A page can be a reference to another page by using a page number. At the beginning of the page, page meta details (such as the rightmost child page number, first free cell offset, and first cell offset) are stored. We'll learn more about 'B-Tree' and 'Pages' later when we implement them in our code..

6. Command Processing:

Interprets user inputs to perform database operations.

7. File-Based Storage:

Saves data permanently in a binary format for reuse. This helps to provide portability across operating systems. This layer differs depending on the operating system SQLite was compiled for.

Beginner Notes:

- Indexing is like creating a table of contents for a book—it helps find what you need quickly.

- Command processing is the part where the database "understands" what you're asking it to do.
- File-based storage ensures that your data isn’t lost when you close the program.

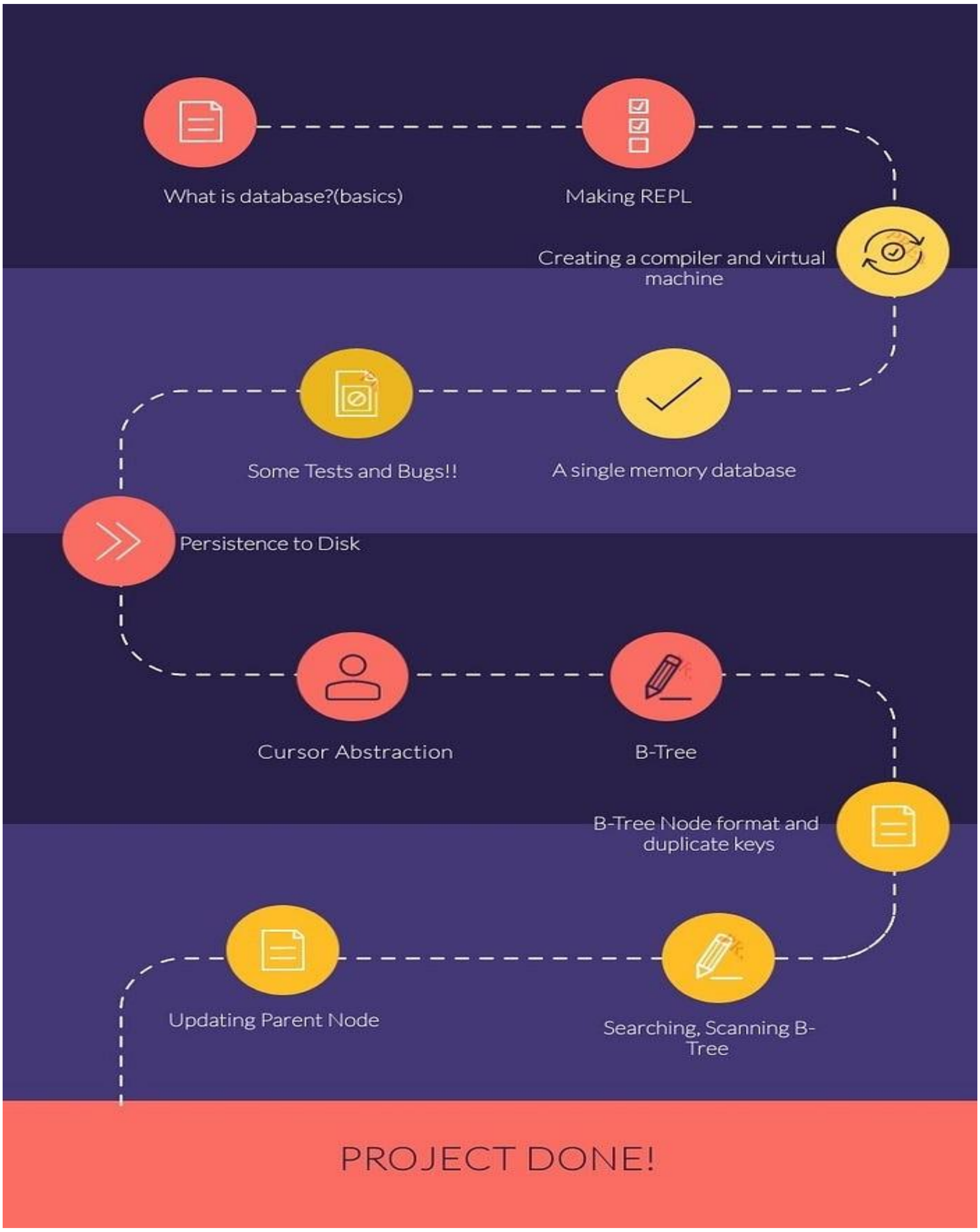

Figure 8:Road Map

5. System Design and Architecture

5.1 Command Processing

Processes user inputs into actions:

1. **Input Parsing**: Splits the command into keywords (e.g., INSERT, SELECT).
2. **Validation**: Checks whether the command is valid.
3. **Execution**: Performs the requested operation (e.g., inserting data).

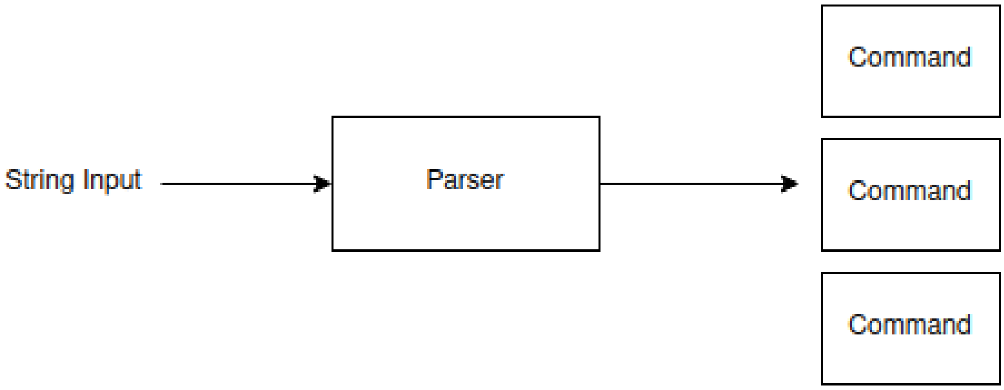

Figure 9: Parsing

Beginner Notes:

- Parsing is breaking the command into pieces the program can understand.
- Validation ensures that your command is "correct" before executing it.

5.2 File Storage Management

Manages saving and loading data to and from a file.

How It Works:

1. Data is stored in rows of fixed size.
2. Metadata, like the total number of records, is saved for quick reference.
3. Binary format ensures that storage is compact and efficient.

Beginner Notes:

- Imagine writing data to a notebook where each page is of the same size—it keeps things organized.
- The metadata acts as an index to quickly find specific pages.

6. Features and Functionalities

6.1 SQL Command Support

6.1.1 Supported operations:

- **INSERT**: Add a new row.
- **SELECT**: Fetch rows based on conditions.
- **UPDATE**: Modify existing rows.
- **DELETE**: Remove rows.

6.1.2 Beginner Notes:

- SQL is like a language to "talk" to databases.
- Each command does a specific job, such as adding or finding data.

6.2 Meta-Commands

Special commands to help developers:

- .exit: Exit the program.
- .btree: Display the structure of the B-Tree.
- .constants: Show internal constants like row size.

6.3 Interactive Shell

Provides a real-time interface where you can type commands and see results immediately.

7. Project Implementation

The project is focused on implementing a database using a B+ Tree as the main data structure to store and retrieve data efficiently. The implementation is broken down into smaller, manageable modules that handle different aspects of the database system.

7.1 Programming Environment

Language:

The project is implemented in C. C is used because of its low-level control over memory and efficient handling of data structures like B+ Trees.

Tools:

- **GCC (GNU Compiler Collection):** Used for compiling the C code. GCC is a robust, widely-used compiler that supports various C standards and provides optimizations for better performance.
- **Python:** Used for testing the implementation. Python scripts are written to simulate user interactions and test SQL-like commands, ensuring that the database behaves as expected.

7.2 Directory Structure

The project files are organized into a directory structure that helps keep the code modular and maintainable. Below is an example of how the project is structured:

```
Database/
├── include/      # Header files
├── src/          # Source files
├── bin/          # Executable
├── test_db.py         # Python test script
└── Makefile      # For build automation
```

Explanation of Directories:

- **include/:** This folder contains the header files (.h) that declare the interfaces (functions, constants, structures) used in the program. Header files ensure modularity by separating function declarations from their implementations in source files.

- **src/:** This folder contains the actual C source files (.c) that implement the core functionality of the project. These files define the logic for managing the B+ Tree, handling input, processing commands, and interacting with the storage files.

- **bin/**: After compiling the source files using GCC, the executables are stored in this directory. These executables are used to run the program from the command line.

- **test_db.py:** This is a Python test script that simulate the operations of the database. These script help verify that the B+

Tree functions as expected and that commands are processed correctly.

- **Makefile:** This is a configuration file used for build automation. It defines how to compile the C code and link the object files to produce the final executable. The Makefile simplifies the build process by allowing the user to run a single command to compile the entire project.

7.3 Key Modules

The project is divided into key modules, each responsible for specific functionality. Below is a detailed description of each module:

btree.c:

This module contains the core logic for managing the B+ Tree. The primary responsibility of this file is to handle operations such as:

- Insertion of key-value pairs into the tree.
- Searching for a key.
- Deleting key-value pairs from the tree.
- Splitting and merging nodes during insertions and deletions.
- This module implements the logic for maintaining the tree's balance and ensuring that the B+ Tree remains sorted and efficient for operations.

command_processor.c:

- This module handles the parsing and processing of SQL-like commands. It interprets the commands from the user (such as INSERT, SELECT, DELETE) and translates them into appropriate function calls in the other modules. This module is essential for:

- Parsing user input.
- Managing queries on the B+ Tree structure.
- Calling the relevant functions to perform operations on the data.

input_handling.c:

- This module is responsible for processing user input. It reads input from the user, validates it, and formats it correctly before passing it to the command_processor.c module. This module handles:
- Reading input from the command line or text files.
- Handling invalid or malformed input.
- Passing the validated input to the command processor for further action.

table.c:

- The table module connects the B+ Tree with the storage file. It manages the process of writing and reading data from the storage file, ensuring that the data is stored persistently. This module:
- Handles the file I/O operations.
- Stores the B+ Tree in a file, allowing data to persist between program executions.
- Loads the data from the file into memory when the program starts.

Beginner Notes:

For beginners, it’s essential to break down the project into smaller, more manageable tasks. Here are some tips to help with the implementation:

Modular Approach:

Divide the project into separate modules, where each module has a clear responsibility. This helps in maintaining the code and debugging individual components.

Start Simple:

Begin with basic functionality and then expand gradually. For example, start by implementing the logic for inserting key-value pairs in the B+ Tree before handling more complex operations like deletions and balancing.

Test Early:

Write test cases early in the development process. This helps in catching errors early and ensures that the different parts of the project are working as expected. Python scripts can simulate database operations, and simple unit tests can help verify individual functions.

Documentation:

Keep your code well-documented. This will help both in the development process and for future reference, especially if you need to revisit the project.

---

8. Detailed Features Analysis

The following sections provide a detailed analysis of key operations that are supported by the B+ Tree implementation, focusing on Insert, Select, Update, and Delete operations.

8.1 Insert Operation

The insert operation allows for the addition of new key-value pairs into the B+ Tree. The steps involved in inserting data are as follows:

Steps:

Parse the Command to Extract Data:

The first step involves parsing the command to extract the key-value data that needs to be inserted into the tree.

Navigate the B+ Tree to Find the Correct Spot:

The B+ Tree is traversed to locate the appropriate leaf node where the new key should be inserted. This search is efficient due to the tree's sorted nature and balanced structure.

Add Data and Split Nodes if Necessary:

If the leaf node is not full (i.e., it has space for more key-value pairs), the new key is inserted directly into the node. If the leaf node is full, a split occurs, where the node is divided into two, and the middle key is promoted to the parent internal node. This may cascade if the internal node also becomes full, requiring further splitting up to the root.

8.1 Select Operation

The select operation is used to retrieve data from the B+ Tree based on certain filters or conditions, typically resembling SQL SELECT queries.

Steps:

Parse the Command to Determine Filters (e.g., WHERE clauses):

The first step is to parse the input command and identify any filters or conditions (such as WHERE clauses) that restrict the data retrieval.

Traverse the B+ Tree to Find Matching Rows:

The B+ Tree is then traversed from the root to the appropriate leaf node. Each key in the tree is checked against the conditions specified in the query. Once a matching row is found, it is returned to the user.

8.2 Update Operation

Steps:

Locate Rows Based on the Condition:

The first step in an update operation is to locate the row(s) that match the specified condition (e.g., a WHERE clause). This requires searching through the B+ Tree to find the leaf node(s) containing the relevant key-value pair.

Modify the Fields Without Disrupting the Tree Structure:

Once the correct row is located, the value associated with the key is updated. The B+ Tree structure is preserved during the update, meaning no node splits or rebalancing is necessary unless the update leads to a key being inserted or deleted.

8.3 Delete Operation

The delete operation is used to remove key-value pairs from the B+ Tree.

Steps:

Locate the Row:

To delete a key-value pair, the row containing the key must first be located. This is achieved by traversing the tree until the corresponding leaf node is found.

Remove It and Re-balance the B+ Tree if Needed:

Once the key-value pair is located, it is removed from the leaf node. If the removal causes a node to become underfilled (i.e., it contains fewer than the minimum number of allowed key-value pairs), the tree is rebalanced by merging nodes or redistributing keys between sibling nodes. This ensures that the tree remains balanced and maintains its efficiency for future operations.

9. System Workflow

9.1 Command Execution Flow

Parse the Command:

The first step is to parse the incoming command to extract the necessary data, operations, and conditions.

Validate the Syntax:

Once the command is parsed, the system checks the syntax for correctness. If the command is malformed or contains errors, the system will return an error message.

Execute the Operation:

If the syntax is valid, the corresponding operation (such as Insert, Select, Update, or Delete) is executed on the B+ Tree or other relevant data structures.

Display Results:

Finally, the results of the operation are displayed to the user, whether it's a success message, the result of a query, or an error if something went wrong.

9.2 B-Tree Indexing

Binary Search within Nodes for Efficiency:

Each node in the B+ Tree uses binary search to quickly locate the appropriate position for keys. This enables faster operations compared to linear search.

Leaf Nodes Contain the Actual Data:

In a B+ Tree, leaf nodes store the actual data or pointers to the data. Internal nodes, on the other hand, only store keys that guide the search process. This structure allows for efficient searching and easy access to the data in the leaf nodes.

10. Code Analysis

10.1 Key Algorithms

- Insert: Balances nodes dynamically.
- Search: Quickly finds keys using binary search.

11. Testing and Validation

Testing and validation are critical for ensuring the correctness and robustness of the database system. In this project, various test cases have been written to verify the functionality of the commands, especially focusing on B+ tree operations such as insert, select, update, and delete.

11.1 Test Cases Overview

Test cases are written for different scenarios to validate whether the system behaves as expected, including error handling, boundary conditions, and regular operations.

Test Cases:

- **Insert and Retrieve a Row:**

Verifies that a row can be inserted and retrieved correctly.

- **Table is Full (Error Handling):**

Tests if the system correctly handles the situation when trying to insert more rows than the table's capacity.

- **Inserting Strings of Maximum Length:**

Ensures that the system can handle strings that are at the maximum allowed length.

- **Error for Strings Too Long:**

Checks if the system correctly prevents inserting strings that exceed the allowed length.

- **Negative ID Handling:**

Tests whether the system throws an error when a negative ID is inserted.

- **Data Persistence after Closing Connection:**

Ensures data is stored persistently after closing and reopening the database.

- **Printing Constants:**

Verifies that constants like ROW_SIZE and LEAF_NODE_MAX_CELLS are correctly printed.

- **Printing the Structure of a Single Node B+ Tree:**

Verifies that the structure of a B+ tree with a single node can be printed correctly.

- **Duplicate ID Handling:**

Tests if the system properly handles attempts to insert duplicate IDs.

- **Structure of a 3-Leaf Node B+ Tree:**

Verifies the structure of a B+ tree with multiple leaf nodes.

- **All Rows in Multi-Level Tree:**

Verifies that the system prints all rows in a multi-level B+ tree.

- **Select with Where Clause:**

Ensures the select where id = X command works as expected.

- **Update Username by ID:**

Verifies the functionality of updating a row's username.

- **Update Email by ID:**

Ensures that the email field can be updated properly.

- **Delete by ID:**

Tests whether the deletion of a record based on its ID works correctly.

Test Summary:

Automated Testing:

The use of automated test scripts ensures that each feature of the database works as expected and handles edge cases appropriately

Continuous Testing:

You can run these test cases regularly to validate updates and avoid regressions in the system.

12. Challenges and Solutions

Example Challenge:

Memory Management:

One of the major challenges encountered in this project was memory management, especially in relation to storing and accessing large amounts of data efficiently. Since the database system was designed to handle rows of fixed sizes, this posed a risk of memory inefficiency and fragmentation.

Solution: To overcome this challenge, we used fixed-size rows and controlled allocation techniques to ensure memory usage remains optimal. This strategy reduces overhead and prevents memory fragmentation, ensuring that the database can handle large amounts of data without performance degradation.

Other Challenges:

Concurrency Control:

Ensuring that multiple operations on the database don't interfere with each other can be tricky.

Solution:

Implementing locks for critical sections helped maintain the integrity of the database when multiple operations are performed simultaneously.

Error Handling:

Handling user input errors, like trying to insert a duplicate entry or an invalid value, presented a challenge.

Solution:

Clear error messages and validation checks were added to improve user experience and prevent invalid data from being stored.

13. Future Improvements

13.1 Add Advanced SQL Features:

The current implementation supports basic SQL commands, but it could be expanded to include more advanced features like joins, sub queries, and group operations, which would make it more versatile for complex queries.

13.2 Support for Transactions:

Adding support for transactions would allow for batch processing of operations, providing users with the ability to commit or rollback changes, enhancing the database's reliability and integrity.

13.3 Optimization:

The database's query execution and data retrieval mechanisms could be optimized further. Techniques like indexing and query caching could be implemented to speed up query performance.

.

14. Conclusion

This project offers a practical understanding of database systems by implementing core features such as indexing, data manipulation, and basic SQL commands from scratch. Through the challenges faced, like memory management and user input validation, and the solutions implemented, this project serves as an excellent foundation for learning how databases work at a low level. Future improvements like adding advanced SQL features and supporting transactions will further enhance the functionality and reliability of the database system.

---

15. References

For any questions, suggestions, or feedback, feel free to reach out to the project maintainers:

- **Ahmed8881**: GitHub Profile
- **hamidriaz1998**: GitHub Profile
- **abdulrehmansafdar**: GitHub Profile

Additionally, the following references were helpful in understanding and implementing key concepts in this project:

- **Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C.** (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
- This book provided the foundational knowledge for data structures such as B+ trees and their implementation in databases.
- **Date, C. J.** (2004). *An Introduction to Database Systems* (8th ed.). Pearson Education.
- A comprehensive resource on database theory and SQL, which guided the development of the database features implemented in this project.