Final Project Proposal

Custom Database Management System

Session: 2023 – 2027

Submitted by:

Mian Saad Tahir 2023-CS-62

Muhammad Talha 2023-CS-70

Supervised by:

Dr. Aatif Hussain

Course:

CSC-207L Advanced Database Management Systems

Department of Computer Science

University of Engineering and Technology

Lahore Pakistan

Table of Contents

Table of Figures

Introduction:

- CUSTOM-DB is an extremely well-implemented Relational Database Management System (RDBMS) and was written as an undergraduate semester project to demonstrate fundamental database design and implementation principles.
- Constructed from scratch in C++, providing a strong and efficient base.
- Allows SQL-like operations such as creating tables and databases, querying and inserting data, updating, and executing complex functionalities such as joins, indexing, and grouping.
- Supports page-based storage engine, B+ tree indexing for fast data access, and buffer pool for high performance.
- Supports data consistency through the use of locking techniques to control transactions.
- Offers role-based access control for secure user authentication.
- Facilitates database backup and recovery to provide data reliability.
- Provides an interactive console-based interface embedded in main.cpp to experience a seamless interaction with a collection of various commands.
- Designed with modularity and extensibility, to be used as a teaching tool to acquire database concepts.
- Offers an operational and functional RDBMS for structured data management, enhancing learning and application.

Objective:

- Create and implement a light RDBMS (CUSTOM-DB) to show database principles with C++.
- Provide SQL-like operations (e.g., create, query, update, join, index) with efficient storage by using page-based engine, B+ tree, and buffer pool.
- Maintain data consistency by way of transaction management and locking.
- Provide safe user authentication with role-based access control.
- Offer stable backup and restore with a recovery system.
- Use an intuitive console-based UI via main.cpp for easy interactions.
- Encourage learning through modular extensible design.
- Deliver an operational RDBMS for effective management of structured data.

Technology Stack:

Frontend:

- C++ Standard Library: Used in development of the user interface of the console program in main.cpp, for input/output operations (e.g., iostream, string, vector) for user input.
- Command-Line Interface (CLI): Implemented through main.cpp to provide a user interface where one can provide input commands and get output.

Backend:

- C++ Programming Language: Main language used to develop the entire system, with control and efficiency on lower-level operations.
- C++ Standard Library: Applied to data structures (e.g., map, vector) and file I/O (e.g., file I/O for storage).

Features and Functionalities:

- **User Authentication**: Offers secure logon with access control based on roles (Admin and Guest roles) through AccessControl.
- **Command Processing:** Has extensive support for SQL-like commands (e.g., CREATE_TABLE, INSERT, SELECT, JOIN) processed through Parser and executed through Executor.
- **Database Management:** Supports creation, modification, and querying of tables and databases through Database.
- **Storage Efficiency:** Leverages a page-based StorageEngine with FileManager as persistent storage, BufferPool for caching, and BPlusTree for indexing.
- **Transaction Handling:** Maintains data consistency with TransactionManager and LockManager for controlling concurrency.
- **Data Recovery:** Provides backup and restoration capability through RecoveryManager for data integrity.
- **User Interface:** Provides a simple console-based interface in main.cpp with command help (HELP) and exit (EXIT) options.
- **Error Handling:** Employs effective error detection and reporting in all components using exceptions.
- **Modular Design:** Designed for extensibility with distinct components for every functionality, allowing for future improvements.

Core Components, Files, and Functions:

Part 1: User-Facing Components

This part includes components responsible for user interaction, authentication, and command processing. It encompasses main.cpp, access_control.hpp/cpp, parser.hpp/cpp, and executor.hpp/cpp.

1.1 main.cpp (User Interface and Command Loop)

- **Purpose**: Acts as the entry point, providing a console-based interface for users to interact with CUSTOM-DB. It handles user authentication, displays available commands, and delegates command execution.
- **Detailed Functions**:
    - **showCommands()**:
        - **Purpose**: Displays a list of supported commands to guide the user.
        - **Details**: Outputs a formatted list of commands (e.g., CREATE_TABLE, INSERT, SELECT). It's called at startup and when the user enters HELP.

        - **Implementation**: Uses cout to print a static string of commands, ensuring user-friendliness.
    - **main()**:
        - **Purpose**: Initializes the system, manages user login, and processes commands in a loop.
        - **Details**:
            - Sets up AccessControl with default users (admin/admin123, guest/guest123).
            - Prompts for username and password, authenticating via AccessControl.
            - Initializes core components (Database, Executor, RecoveryManager).
            - Enters a loop to accept user input, tokenizing commands and handling special cases (HELP, EXIT, BACKUP, RESTORE) directly, while delegating others to Executor.
            - Includes error handling using try-catch for robustness.

1.2 AccessControl (access_control.hpp, access_control.cpp)

- **Purpose**: Manages user authentication and role-based access control.
- **Detailed Functions** (Assumed based on usage in main.cpp and test_access_control.cpp):
    - **addUser(string username, string password, Role role)**:
        - **Purpose**: Registers a new user with credentials and a role.
        - **Details**: Stores user data in a structure (likely a map<string, pair<string, Role>>), associating usernames with passwords and roles.
    - **authenticate(string username, string password)**:
        - **Purpose**: Verifies user credentials.
        - **Details**: Checks if the username exists and if the password matches the stored value, returning true on success.
    - **getUserRole(string username)**:
        - **Purpose**: Retrieves the role of an authenticated user.
        - **Details**: Returns the role (Role::ADMIN or Role::GUEST) for access control decisions.
- **Code Structure**

```
class AccessControl {
private:
    map<string, pair<string, Role>> users;
public:
    void addUser(string username, string password, Role role) {
        users[username] = {password, role};
    }
    bool authenticate(string username, string password) {
        auto it = users.find(username);
```

```
    if (it != users.end() && it->second.first == password) return true;
    return false;
  }
  Role getUserRole(string username) {
    return users[username].second;
  }
};
```

1.3 Parser (parser.hpp, parser.cpp)

- **Purpose**: Parses user commands into a structured format for execution.
- **Detailed Functions** (Assumed based on test_parser.cpp):
  - **parseFullCommand(const string &input)**:
    - **Purpose**: Converts a command string into a ParsedCommand object.
    - **Details**: Tokenizes the input (similar to main.cpp), validates syntax, and identifies the command type (e.g., INSERT, SELECT) and arguments (e.g., table name, values).
  - **Code**:

```
struct ParsedCommand {
  string commandType;
  vector<string> arguments;
};
class Parser {
public:
  ParsedCommand parseFullCommand(const string &input) {
    ParsedCommand result;
    istringstream iss(input);
    string token;
    iss >> result.commandType;
    while (iss >> token) result.arguments.push_back(token);
    for (auto &c : result.commandType) c = toupper(c);
    return result;
  }
};
```

1.4 Executor (executor.hpp, executor.cpp)

- **Purpose**: Executes parsed commands by interacting with the Database.
- **Detailed Functions** (Assumed based on test_executor.cpp and usage in main.cpp):

- **executeCommand(const string &commandLine)**:
  - **Purpose**: Processes a command and performs the requested operation.
  - **Details**:
    - Uses Parser to parse the command.
    - Maps the command type to a Database operation (e.g., INSERT → insertIntoTable()).
    - Handles errors by throwing exceptions (caught in main.cpp).
- **Code**:

```
class Executor {
private:
  Database &database;
  Parser parser;
public:
  Executor(Database &db) : database(db) {}
  void executeCommand(const string &commandLine) {
    ParsedCommand cmd = parser.parseFullCommand(commandLine);
    if (cmd.commandType == "CREATE_TABLE") {
      vector<pair<string, string>> columns;
      for (size_t i = 2; i < cmd.arguments.size(); i++) {
        size_t colon = cmd.arguments[i].find(':');
        string name = cmd.arguments[i].substr(0, colon);
        string type = cmd.arguments[i].substr(colon + 1);
        columns.emplace_back(name, type);
      }
      database.createTable(cmd.arguments[1], columns);
    } else if (cmd.commandType == "INSERT") {
      vector<string> values(cmd.arguments.begin() + 2, cmd.arguments.end());
      database.insertIntoTable(cmd.arguments[1], values);
    } else {
      throw runtime_error("Unsupported command: " + cmd.commandType);
    }
  }
};
```

Part 2: Backend Components

This part includes components responsible for data management, storage, transactions, and recovery. It encompasses database.hpp/cpp,

storage_engine.hpp/cpp, file_manager.hpp/cpp, buffer_pool.hpp/cpp, b_plus_tree.hpp/cpp, page.hpp/cpp, transaction.hpp/cpp, lock_manager.hpp/cpp, and recovery.hpp/cpp.

2.1 Database (database.hpp, database.cpp)

- **Purpose**: Manages in-memory database operations, such as table creation and data manipulation.
- **Detailed Functions** (Assumed based on test_database.cpp and test_table.cpp):
    - **createDatabase(const string &name)**:
        - **Purpose**: Initializes a new database instance.
        - **Details**: Sets up a database context, possibly storing metadata.
    - **createTable(const string &table, const vector<pair<string, string>> &columns)**:
        - **Purpose**: Creates a table with specified columns and types.
        - **Details**: Defines the schema, including column names and data types (e.g., INT, STRING).
    - **insertIntoTable(const string &table, const vector<string> &values)**:
        - **Purpose**: Inserts a row into a table.
        - **Details**: Validates values against the schema and stores the row, possibly delegating to StorageEngine.
    - **Code**:

```
class Database {
private:
  map<string, Table> tables;
  StorageEngine storage;
public:
  void createDatabase(const string &name) {
    // Initialize database metadata
  }
  void createTable(const string &tableName, const vector<pair<string, string>> &columns) {
    tables[tableName] = Table(tableName, columns);
  }
  void insertIntoTable(const string &tableName, const vector<string> &values) {
    if (tables.find(tableName) == tables.end())
      throw runtime_error("Table not found: " + tableName);
    tables[tableName].insertRow(values);
    storage.storeRow(tableName, values);
  }
};
```

2.2 StorageEngine and Related Components

- **Purpose**: Manages persistent storage, caching, and indexing.

- **Files**:
  - **file_manager.hpp/cpp**:
    - **test_create_and_open()**, **test_write_and_read()** (From test_file_manager.cpp):
      - **Purpose**: Tests file creation, opening, and page-level read/write operations.
      - **Details**: Provides low-level file I/O for storing pages.
  - **buffer_pool.hpp/cpp**:
    - **LRU-based caching methods** (Assumed):
      - **Purpose**: Caches pages in memory to reduce disk I/O.
      - **Details**: Implements a least-recently-used (LRU) eviction policy.
  - **b_plus_tree.hpp/cpp**:
    - **testInsert()**, **testRemove()**, **testSearchByCondition()**, **testPrintTree()** (From test_b_plus_tree.cpp):
      - **Purpose**: Tests insertion, removal, and searching in a B+ tree.
      - **Details**: Provides efficient indexing for key-based lookups.
  - **page.hpp/cpp**:
    - **Write/Read methods** (Assumed):
      - **Purpose**: Manages raw data operations on pages.
      - **Details**: Handles serialization and deserialization of data.
- **Code** (StorageEngine):

```
class StorageEngine {
private:
  FileManager fileManager;
  BufferPool bufferPool;
  BPlusTree index;
public:
  void storeRow(const string &tableName, const vector<string> &values) {
    Page page = bufferPool.getPage(tableName);
    page.writeRow(values);
    fileManager.writePage(page);
    index.insert(values[0], page.getPageId()); // Index on first column
  }
};
```

2.3 TransactionManager and LockManager (transaction.hpp/cpp, lock_manager.hpp/cpp)

- **Purpose**: Manages concurrency with locking.
- **Detailed Functions** (Assumed based on test_transaction.cpp):
  - **Lock acquisition/release methods**:
    - **Purpose**: Ensures safe concurrent access to data.

    - **Details**: Implements table-level locking to prevent conflicts during transactions.
- **Code**:

```
class LockManager {
private:
  map<string, bool> locks;
public:
  void acquireLock(const string &tableName) {
    if (locks[tableName]) throw runtime_error("Table locked");
    locks[tableName] = true;
  }
  void releaseLock(const string &tableName) {
    locks[tableName] = false;
  }
};
class TransactionManager {
private:
  LockManager lockManager;
public:
  void beginTransaction() {}
  void commitTransaction() {}
};
```

2.4 RecoveryManager (recovery.hpp, recovery.cpp)

- **Purpose**: Handles database backup and restoration.
- **Detailed Functions** (From main.cpp and test_recovery.cpp):
    - **createBackup(string backupName)**:
        - **Purpose**: Creates a backup of the database file.
        - **Details**: Copies database.txt to a file named backupName.
    - **restoreBackup(string backupName)**:
        - **Purpose**: Restores the database from a backup.
        - **Details**: Overwrites database.txt with the contents of backupName.

- **Code**:

```
class RecoveryManager {
private:
```

```
    string databaseFileName;
public:
    RecoveryManager(const string &fileName) : databaseFileName(fileName) {}
    bool createBackup(const string &backupName) {
        ifstream src(databaseFileName, ios::binary);
        ofstream dst(backupName, ios::binary);
        if (!src || !dst) return false;
        dst << src.rdbuf();
        return true;
    }
    bool restoreBackup(const string &backupName) {
        ifstream src(backupName, ios::binary);
        ofstream dst(databaseFileName, ios::binary);
        if (!src || !dst) return false;
        dst << src.rdbuf();
        return true;
    }
};
```

Summary of Division

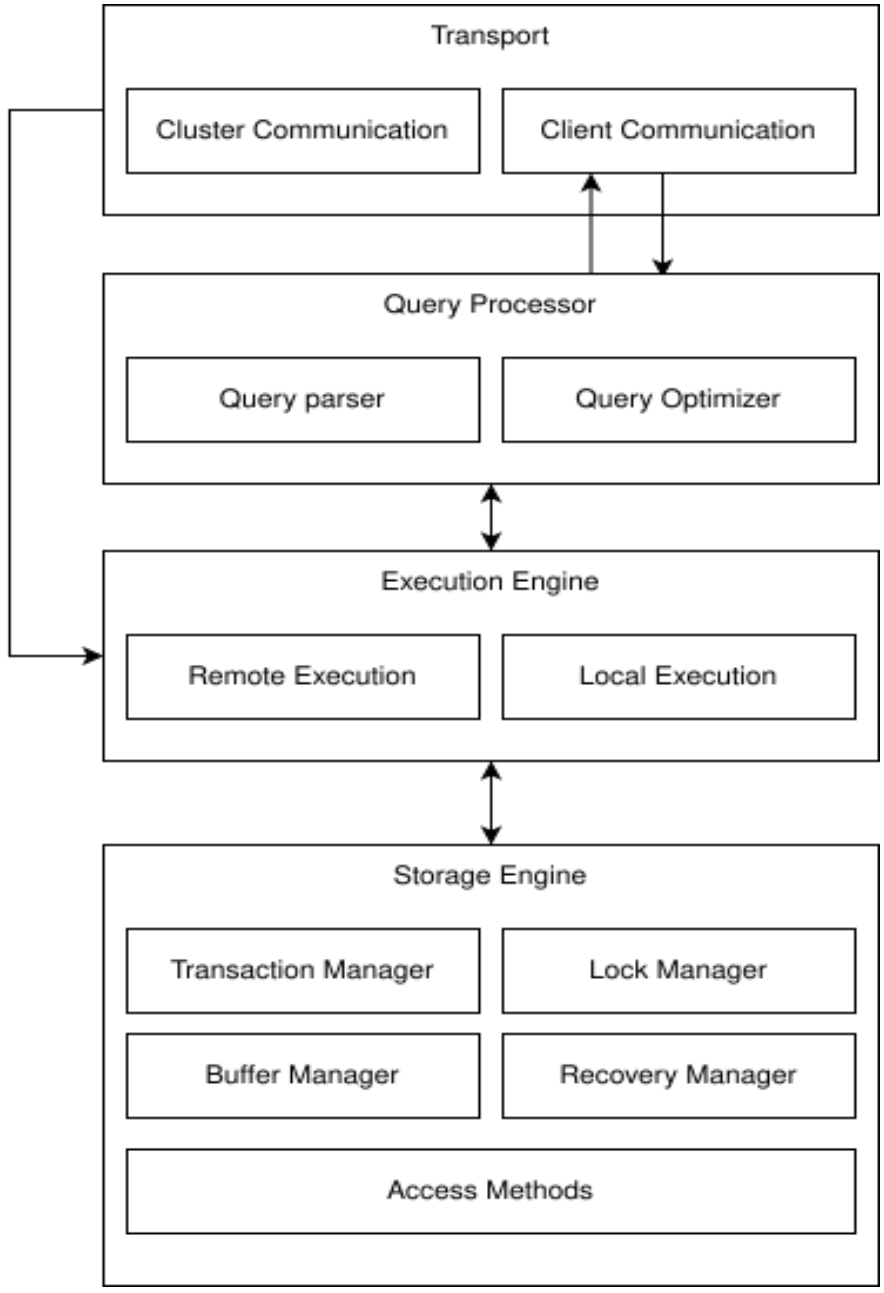
- **Part 1 (User-Facing Components)**: Focuses on user interaction (main.cpp), authentication (AccessControl), and command processing (Parser, Executor). These components ensure users can securely log in, input commands, and receive results.
- **Part 2 (Backend Components)**: Handles data management (Database), storage (StorageEngine, FileManager, BufferPool, BPlusTree, Page), concurrency (TransactionManager, LockManager), and recovery (RecoveryManager). These components ensure data persistence, efficiency, and reliability.

System Requirements:

- **Purpose**: Specify the hardware and software requirements needed to run CUSTOM-DB.
- **Details to Include**:
    - **Hardware**: Minimal requirements (e.g., 1 GB RAM, 100 MB disk space for storage).
    - **Software**: Operating system (e.g., Windows, Linux, macOS), C++ compiler (e.g., g++ with C++11 support or later).
    - **Dependencies**: Note that CUSTOM-DB relies solely on the C++ Standard Library, with no external libraries.

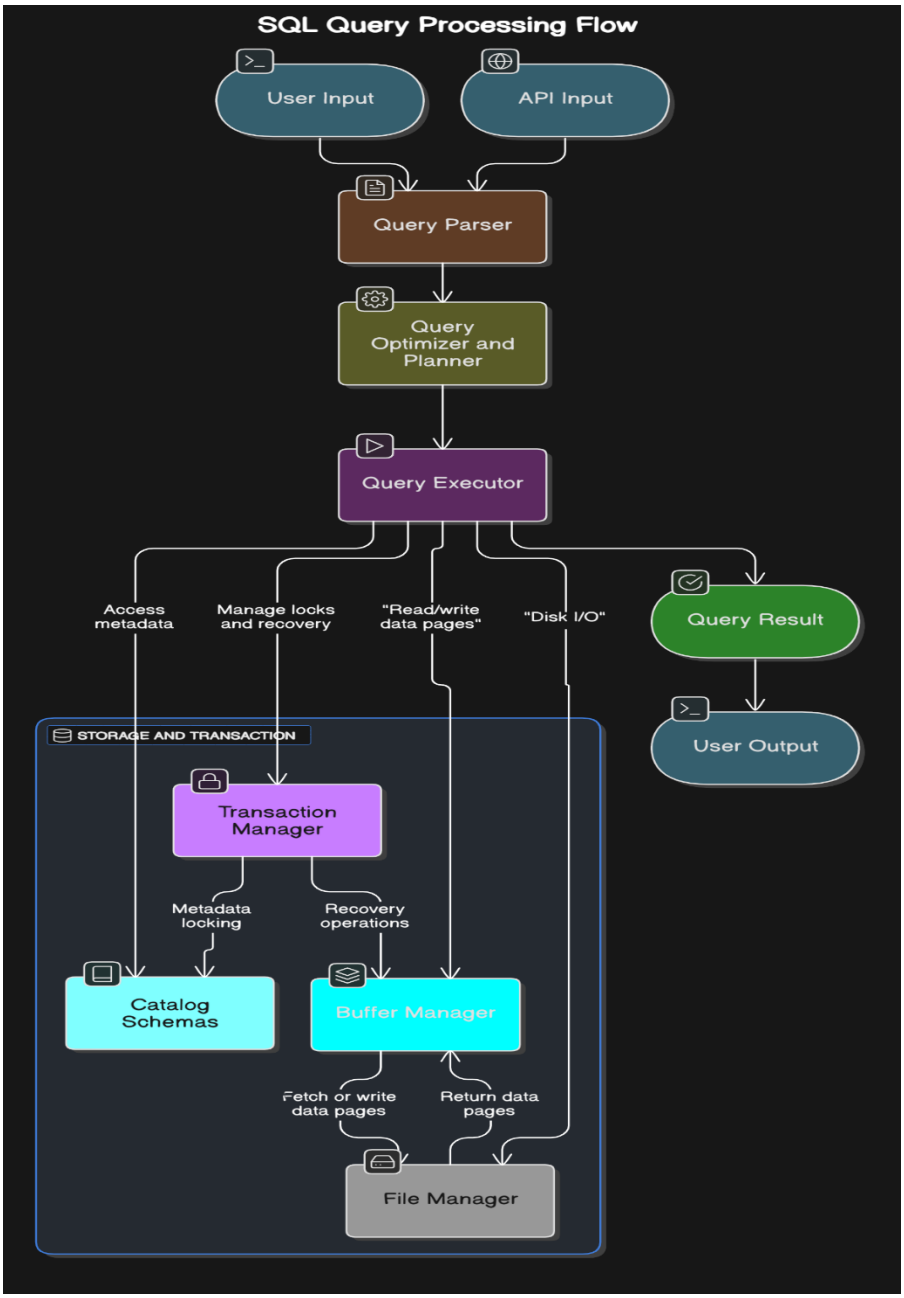
Architecture Diagram:

The Transport layer manages communication with clients and clusters, providing input to the Query Processor, which consists of a Query Parser for syntax checking and a Query Optimizer for optimal query planning. The executed queries are processed by the Execution Engine, providing support for Remote Execution as well as Local Execution. The Storage Engine is responsible for data persistence, with a Transaction Manager for consistency, Lock Manager for concurrency control, Buffer Manager for caching, and Recovery Manager for backup and restoration of data, while Access Methods offer the underlying data retrieval mechanisms.



Data Flow Diagram (DFD):

- Shows SQL Query Processing Flow.
- Inputs: User Input, API Input.
- Query Parser: Analyzes syntax.
- Query Optimizer/Planner: Optimizes execution.
- Query Executor: Processes queries, accesses metadata, manages locks, reads/writes data.
- Output: Query Result to User Output.
- Storage/Transaction: Transaction Manager (locking, recovery), Catalog Schemas, Buffer Manager, File Manager.



UI Components:

Testing each component/module:

test_file_manager:

```
C:\Users\Talha\Desktop\UET\SEMESTER 4\ADBMS\ADBMS Lab\Custom-DB\ADBMS Semester Project>test.exe
Running FileManager tests...
File create/open test passed.
Write/Read page test passed.
All FileManager tests passed.
```

Figure 1-File Manager

test_page:

```
C:\Users\Talha\Desktop\UET\SEMESTER 4\ADBMS\ADBMS Lab\Custom-DB\ADBMS Semester Project>test_page.exe

==== Testing writeData() and readData() ====
Written: id=1,name=Alice,age=30
Read:    id=1,name=Alice,age=30

==== Testing getRawData() ====
Raw data matches expected.

==== Testing loadFromRawData() ====
Loaded Raw: id=2,name=Bob,age=40
Read:       id=2,name=Bob,age=40

==== Testing hasColumn() ====
Column check passed.

==== Testing updateColumn() ====
Updated: id=2,name=Bobby,age=40
Update passed.

All Page tests passed successfully.
```

Figure 2-Page

test_parser:

```
C:\Users\Talha\Desktop\UET\SEMESTER 4\ADBMS\ADBMS Lab\Custom-DB\ADBMS Semester Project>test_parser.exe
All parser tests passed.
```

Figure 3-Parser

test_recovery:

```
C:\Users\Talha\Desktop\UET\SEMESTER 4\ADBMS\ADBMS Lab\Custom-DB\ADBMS Semester Project>test_recovery.exe
[TEST] Creating Dummy Database...
[TEST] Creating Backup...
Backup created successfully.
[TEST] Listing Backups...
- backup1.bak
- backup1.bak
[TEST] Restoring Backup...
Backup restored successfully.
```

Figure 4-Recovery

test_storage_engine:

```
C:\Users\Talha\Desktop\UET\SEMESTER 4\ADBMS\ADBMS Lab\Custom-DB\ADBMS Semester Project>test_storage_engine.exe
Evicted page: students_12
Evicted page: students_10
Evicted page: students_5
5 : 5
6 : 6
7 : 7
10 : 10
12 : 12
17 : 17
20 : 20
30 : 30
Current LRU order:
- students_30
- students_20
- students_17
- students_7
- students_6
Evicted page: students_6
Search 12: Twelve
Search 21: Key not found
```

Figure 5-Storage Engine

test_table:

```
C:\Users\Talha\Desktop\UET\SEMESTER 4\ADBMS\ADBMS Lab\Custom-DB\ADBMS Semester Project>test_table.exe
====================
TESTING TABLE SCHEMA
====================
Column added: id
Column added: name
Column added: balance
Column added: is_active

--- Table Schema ---

--- Table Schema ---
Table: Users
Schema:
- id: INT
- name: STRING
- balance: FLOAT
- is_active: BOOL

--- Inserting Valid Rows ---
Row inserted.
Row inserted.

--- Inserting Invalid Row (should fail) ---
Error: Value count does not match column count.

--- Displaying Rows ---

--- Rows in Table: Users ---
id: 1 | name: Alice | balance: 1500.500000 | is_active: true |
id: 2 | name: Bob | balance: 899.989990 | is_active: false |

All tests passed.
```

Figure 6-Table

test_transaction:

```
C:\Users\Talha\Desktop\UET\SEMESTER 4\ADBMS\ADBMS Lab\Custom-DB\ADBMS Semester Project>test_transaction.exe
[Transaction 1] Trying to acquire lock...
Acquiring new lock on table: users
[Transaction 1] Lock acquired, doing work... (simulated)
[Transaction 1] Releasing lock...
Releasing lock on table: users
Acquiring new lock on table: users
[Transaction 2] Trying to acquire lock...
Table 'users' is already locked!
[Transaction 2] Could not acquire lock. Aborting transaction.
Releasing lock on table: users
[Transaction 3] Trying to acquire lock...
Acquiring new lock on table: users
[Transaction 3] Lock acquired, doing work... (simulated)
[Transaction 3] Releasing lock...
Releasing lock on table: users
```

Figure 7-Transaction

test_executor:

```
C:\Users\Talha\Desktop\UET\SEMESTER 4\ADBMS\ADBMS Lab\Custom-DB\ADBMS Semester Project>test_executor.exe
Inserted: id123 -> John
Inserted: id456 -> Jane
Unknown command: SHOW
Error: Value not found.
Error: Value not found.
Removed: id123 -> John
Unknown command: SHOW
testBasicCommands passed.
Database  destroyed.
Database testdb created and set as active.
Altered to database: prod
Inserted: user1 -> Alice
Error: Value not found.
Database flushed.
Unknown command: SHOW
testDatabaseOps passed.
Database prod destroyed.
Inserted: k1 -> v1
Inserted: k2 -> v2
Inserted: k3 -> v3
Arguments: k1
Usage: CREATE_INDEX <table> <column>
Usage: DISTINCT <table> <column>
Usage: MATCH <table> <column> <pattern>
Usage: ORDER <table> <column> ASC|DESC
Usage: LIMIT <table> <count>
testAdvancedCommands passed.
Database  destroyed.
Inserted: id100 -> Tom
Table <Default> not found.
Update failed or no rows matched condition.
Error: Value not found.
Usage: DELETE <table> WHERE <column> <op> <value>
Error: Value not found.
testUpdateDeleteSelect passed.
Database  destroyed.
All executor tests passed.
```

Figure 8-Executor

test_database:

```
C:\Users\Talha\Desktop\UET\SEMESTER 4\ADBMS\ADBMS Lab\Custom-DB\ADBMS Semester Project>test_database.exe
Running key-value tests...
Inserted: name -> Alice
Removed: name -> Alice
Error: No such key-value pair to remove.
All key-value tests passed!
Database  destroyed.
```

Figure 9-Database

test_b_plus_tree:

```
C:\Users\Talha\Desktop\UET\SEMESTER 4\ADBMS\ADBMS Lab\Custom-DB\ADBMS Semester Project>test.exe
Insert Test Passed!
Remove Test Passed!
Result for values greater than 250: 30 40
Search by Condition Test Passed!
Tree Structure:
5 : 50
15 : 150
25 : 250
```

Figure 10-B+ Tree

test_access_control:

```
C:\Users\Talha\Desktop\UET\SEMESTER 4\ADBMS\ADBMS Lab\Custom-DB\ADBMS Semester Project>test_access_control.exe
[TEST] Adding users...
[TEST] Authenticating users...
Admin correct password: PASS
Admin wrong password: FAIL
[TEST] Checking user roles...
Role for admin: ADMIN
Role for unknown: GUEST
```

Figure 11-Access Control

```
C:\Users\Talha\Desktop\UET\SEMESTER 4\ADBMS\ADBMS Lab\Custom-DB\ADBMS Semester Project>main.exe
==================================
  Welcome to Custom RDBMS Console
==================================

Login
------
Username: admin
Password: admin123
Login successful. Role: Admin
```

```
==================================
  Welcome to Custom RDBMS Console
==================================

Login
------
Username: admin
Password: admin123
Login successful. Role: Admin

Available Commands:
  CREATE_DATABASE <name>
  CREATE_TABLE <table_name> <col1:type1> <col2:type2> ...
  INSERT <table_name> <val1> <val2> ...
  SELECT <columns> FROM <table_name>
  DELETE <table_name> WHERE <column> = <value>
  UPDATE <table_name> SET <column> = <value> WHERE <column> = <value>
  JOIN <table1> <table2> ON <column1> = <column2>
  GROUP_BY <table_name> <column>
  ORDER <table_name> <column> ASC|DESC
  MATCH <table_name> <column> <value>
  LIMIT <table_name> <n>
  DISTINCT <table_name> <column>
  CREATE_INDEX <table_name> <column>
  SHOWDATABASE
  SHOWTABLES
  FLUSH
  PUT <key> <value>
  GET <key>
  REMOVE <key>
  HELP
  EXIT

>>>
```

```
>>> CREATE_DATABASE company
Database company created and set as active.

>>> CREATE_TABLE employees id:pk name:string salary:float activeStatus:bool
Table employees created.

>>> INSERT employees 1 Alice 23000 true 2 Bob 42000 true 3 Jack 72000 false 4 Gotham 86000 1
Inserted into table: employees
Inserted values into table <employees>.

>>> SHOWTABLES

========== Table: employees ==========
id | name | salary | activeStatus
---------------------------------------
1 | Alice | 23000 | true
2 | Bob | 42000 | true
3 | Jack | 72000 | false
4 | Gotham | 86000 | 1
=======================================
```

```
>>> SELECT * FROM employees
========== Selected columns from table: employees ==========
id | name | salary | activeStatus
---------------------------------------
1 | Alice | 23000 | true
2 | Bob | 42000 | true
3 | Jack | 72000 | false
4 | Gotham | 86000 | 1
=======================================

>>> SELECT salary FROM employees WHERE salary > 42000
========== Selected columns from table: employees ==========
salary
---------------------------------------
72000
86000
=======================================

>>> SELECT name FROM employees WHERE activeStatus = true
========== Selected columns from table: employees ==========
name
---------------------------------------
Alice
Bob
=======================================

>>> SELECT activeStatus FROM employees WHERE name = Jack
========== Selected columns from table: employees ==========
activeStatus
---------------------------------------
false
=======================================
```

```
>>> UPDATE employees SET activeStatus = true WHERE name = Gotham
Updated row: 4 Gotham 86000 true

>>> SHOWTABLES

========== Table: employees ==========
id | name | salary | activeStatus
---------------------------------------
1 | Alice | 23000 | true
2 | Bob | 42000 | true
3 | Jack | 72000 | false
4 | Gotham | 86000 | true
=======================================
```

```
>>>  DELETE employees WHERE salary = 72000
Deleted row: 3 Jack 72000 false

>>> SHOWTABLES

========== Table: employees ==========
id | name | salary | activeStatus
-----------------------------------------------
1 | Alice | 23000 | true
2 | Bob | 42000 | true
4 | Gotham | 86000 | 1
=======================================
```

```
>>> JOIN employees designation ON name = name
Joined: employees with designation => employees_designation_join

>>> SHOWTABLES

========== Table: employees_designation_join ==========
id | name | salary | activeStatus | id | name
-----------------------------------------------------------------------
1 | Alice | 23000 | true | 1 | Alice
=====================================

========== Table: designation ==========
id | name
----------------------
1 | Alice
2 | Rober
=====================================

========== Table: employees ==========
id | name | salary | activeStatus
-----------------------------------------------
1 | Alice | 23000 | true
2 | Bob | 42000 | true
4 | Gotham | 86000 | 1
=====================================
```

```
>>> GROUP_BY employees salary
Grouped by: salary -> Created table <employees_grouped_by_salary>

>>> SHOWTABLES

========== Table: employees_designation_join ==========
id | name | salary | activeStatus | id | name
-----------------------------------------------------------------------
1 | Alice | 23000 | true | 1 | Alice
=====================================

========== Table: employees_grouped_by_salary ==========
id | name | salary | activeStatus
-----------------------------------------------
1 | Alice | 23000 | true
2 | Bob | 42000 | true
4 | Gotham | 86000 | 1
=======================================

========== Table: designation ==========
id | name
----------------------
1 | Alice
2 | Rober
=====================================

========== Table: employees ==========
id | name | salary | activeStatus
-----------------------------------------------
1 | Alice | 23000 | true
2 | Bob | 42000 | true
4 | Gotham | 86000 | 1
=======================================
```

```
>>>  MATCH employees name Bob
2 | Bob | 42000 | true
```

```
>>> LIMIT employees 1
1 | Alice | 23000 | true
```

```
>>> DISTINCT employees activeStatus
1 | Alice | 23000 | true
4 | Gotham | 86000 | 1
```

```
>>> ORDER employees salary DESC
Table 'employees' ordered by column 'salary' in DESC order.

>>> SHOWTABLES

========== Table: employees_designation_join ==========
id | name | salary | activeStatus | id | name
-------------------------------------------------------------
1 | Alice | 23000 | true | 1 | Alice
=====================================

========== Table: employees_grouped_by_salary ==========
id | name | salary | activeStatus
-----------------------------------------
1 | Alice | 23000 | true
2 | Bob | 42000 | true
4 | Gotham | 86000 | 1
=====================================

========== Table: designation ==========
id | name
--------------------
1 | Alice
2 | Rober
====================================

========== Table: employees ==========
id | name | salary | activeStatus
-----------------------------------------
4 | Gotham | 86000 | 1
2 | Bob | 42000 | true
1 | Alice | 23000 | true
=====================================
```

```
>>> ORDER employees name ASC
Table 'employees' ordered by column 'name' in ASC order.

>>> SHOWTABLES

========== Table: employees_designation_join ==========
id | name | salary | activeStatus | id | name
-------------------------------------------------------------------
1 | Alice | 23000 | true | 1 | Alice
=====================================

========== Table: employees_grouped_by_salary ==========
id | name | salary | activeStatus
---------------------------------------------
1 | Alice | 23000 | true
2 | Bob | 42000 | true
4 | Gotham | 86000 | 1
=====================================

========== Table: designation ==========
id | name
----------------------
1 | Alice
2 | Rober
=====================================

========== Table: employees ==========
id | name | salary | activeStatus
---------------------------------------------
1 | Alice | 23000 | true
2 | Bob | 42000 | true
4 | Gotham | 86000 | 1
=====================================
```

```
>>> CREATE_INDEX employees id
Arguments: employees id
Created index for column 'id' in table 'employees'.
```

```
>>> PUT username Alice
Inserted: username -> Alice

>>> GET username "Alice"
Error: Value not found.

>>> GET Alice
Usage: GET <column> <value>

>>> GET username Alice
Found: username -> Alice

>>> REMOVE username Alice
Removed: username -> Alice
```

Figure 12-All Main Server

Weaknesses:

- **Limited Concurrency**: LockManager uses table-level locking, causing bottlenecks during concurrent access; lacks row-level locking.

- **In-Memory Constraints**: Database relies on RAM, limiting scalability for large datasets.
- **Basic Indexing**: BPlusTree lacks optimizations (e.g., bulk loading), reducing efficiency for complex queries.
- **No Network Support**: Operates as a standalone console app, missing client-server capabilities.
- **Minimal Recovery**: RecoveryManager supports basic backup/restore but lacks crash recovery or transaction logging.
- **Removed Features**: Excluded api and utils, limiting serialization and external integration.
- **Testing Gaps**: Incomplete test coverage in files like test_transaction.cpp for edge cases and concurrency.

Future Enhancements:

- **Advanced Concurrency**: Implement row-level locking in LockManager to improve performance.
- **Storage Optimization**: Enhance StorageEngine for disk-based storage to handle larger datasets.
- **Better Indexing**: Add bulk loading and adaptive indexing to BPlusTree for efficiency.
- **Network Support**: Introduce client-server architecture for remote access.
- **Robust Recovery**: Add transaction logging and crash recovery to RecoveryManager.
- **Reintroduce Features**: Restore api and utils for better modularity.
- **Comprehensive Testing**: Expand test suites to cover edge cases and performance.
- **GUI Development**: Replace console UI with a graphical interface (e.g., Qt).
- **Query Optimization**: Improve Executor with cost-based query planning.

Conclusion:

Completed by May 14, 2025, CUSTOM-DB showcases RDBMS principles via a C++ implementation, achieving SQL-like query processing, B+ tree indexing, authentication, and basic transaction/recovery features. Despite limitations like concurrency and storage constraints, it offers valuable learning on database internals. Future enhancements can make it a robust, scalable solution for educational and practical use.

Github Link:

https://github.com/MianSaadTahir/Custom-DB