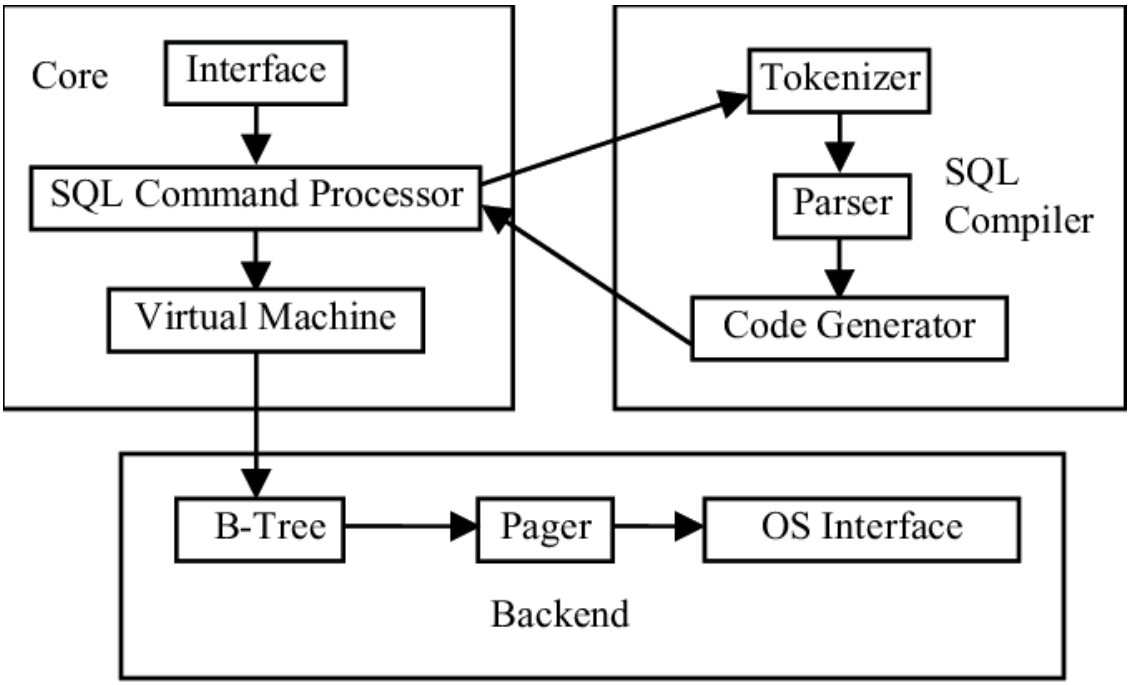
Compiler_and_Virtual_Machine

July 15, 2024

1 SQL Architecture

[ ]:
```
from IPython.display import Image
Image(filename='./Images/sql.png')
```

[ ]:



SQL architecture typically refers to the layered structure of a database management system (DBMS) that interprets and executes SQL queries. While implementations can vary among different SQL-based DBMS (like MySQL, PostgreSQL, Oracle, etc.), the core architecture often follows a similar pattern. Here's a simplified overview of the main components involved in SQL architecture:

1.0.1 1. Client Layer

- **User Interface**: This is where users interact with the database, typically through applications, command-line tools, or graphical interfaces.

- **Application Layer**: Applications communicate with the database by sending SQL queries. This layer often includes drivers or APIs that abstract the details of SQL command execution.

1.0.2 2. Server Layer

- **SQL Parser and Compiler**: When the server receives an SQL query, this component parses the query to ensure it's syntactically correct and then compiles it into a format that the database can execute.
- **Query Optimizer**: This component analyzes different ways to execute a given query and selects the most efficient execution plan. It considers factors like data indexes, the cost of different query paths, and the current database workload.
- **Execution Engine**: Executes the optimized query plan. It interacts with the data storage layer to retrieve or modify data as required by the query.

1.0.3 3. Storage Layer

- **Data Files**: This is where the actual data resides. Data files can be organized in various formats, such as tables, heaps, or indexes.
- **Transaction Manager**: Ensures that all database transactions are processed reliably and adhere to ACID properties (Atomicity, Consistency, Isolation, Durability). It handles transaction logs, locking, and concurrency control mechanisms.
- **Buffer Manager**: Manages the cache of data in memory to reduce the number of disk accesses. Frequently accessed data is kept in memory for faster retrieval.

1.0.4 4. Disk Storage

- **Physical Storage**: The physical layer where data is stored on disk. This includes the organization of data files, indexes, and logs on the storage medium.

1.0.5 Summary

SQL architecture is designed to efficiently process and manage SQL queries and data. It encompasses client interaction, query parsing and optimization, execution, and data storage management. Each layer of the architecture plays a crucial role in ensuring that SQL queries are executed accurately and efficiently, providing users and applications with reliable and fast access to the data stored in the database.

2 What is meta command? How Can I implement it in my main program?

2.0.1 Meta Commands in SQLite

Meta commands (or dot commands) are commands specific to the SQLite command-line interface (CLI) that provide various utility functions, such as `.tables` to list tables, `.schema` to display the schema of a table, and `.exit` to close the session. They are not standard SQL commands but are used to interact with the SQLite environment.

2.0.2 Implementing Meta Commands in Your Program

To implement meta commands in your SQLite-like project, you'll need to:

1. **Identify and Parse Meta Commands**: Check if the input starts with a dot (`.`), indicating a meta command.
2. **Handle Each Meta Command**: Execute the corresponding function for each recognized meta command.

Here's a step-by-step guide to implementing meta commands in your C program:

2.0.3 Step 1: Modify the Main Loop to Recognize Meta Commands

Update the `read_input` and `process_input` functions to recognize and handle meta commands.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Function declarations
void print_prompt();
char* read_input();
void process_input(char *input);

// Meta command handlers
void handle_meta_command(char *input);
void list_tables();
void exit_program();

int main() {
    while (1) {
        print_prompt();
        char *input = read_input();
        if (input[0] == '.') {
            handle_meta_command(input);
        } else {
            process_input(input);
        }
        free(input);
    }
    return 0;
}

void print_prompt() {
    printf("db > ");
}

char* read_input() {
    char *input = NULL;
    size_t buffer_size = 0;
    getline(&input, &buffer_size, stdin);
    return input;
}
```

```c
void process_input(char *input) {
    // Process SQL commands here
    printf("You entered an SQL command: %s", input);
}

void handle_meta_command(char *input) {
    if (strcmp(input, ".tables\n") == 0) {
        list_tables();
    } else if (strcmp(input, ".exit\n") == 0) {
        exit_program();
    } else {
        printf("Unrecognized meta command: %s", input);
    }
}

void list_tables() {
    // List tables in the database (mock implementation)
    printf("List of tables (mock):\n");
    printf("users\n");
    printf("orders\n");
}

void exit_program() {
    printf("Exiting...\n");
    exit(0);
}
```

2.0.4 Step 2: Extend Meta Command Functionality

Add more meta commands as needed. Here are some common ones:

- `.schema [table_name]`: Show the schema of a table.
- `.help`: Show available meta commands.

2.0.5 Step 3: Implement the Meta Command Functions

Extend the `handle_meta_command` function to include additional commands.

```c
void handle_meta_command(char *input) {
    if (strcmp(input, ".tables\n") == 0) {
        list_tables();
    } else if (strcmp(input, ".exit\n") == 0) {
        exit_program();
    } else if (strncmp(input, ".schema", 7) == 0) {
        show_schema(input);
    } else if (strcmp(input, ".help\n") == 0) {
        show_help();
    } else {
```

```
        printf("Unrecognized meta command: %s", input);
    }
}

void show_schema(char *input) {
    // Parse the input to get the table name
    char *table_name = strtok(input + 8, " \n");
    if (table_name) {
        // Show the schema for the given table (mock implementation)
        printf("Schema for table %s (mock):\n", table_name);
        printf("CREATE TABLE %s (\n  id INTEGER PRIMARY KEY,\n  name TEXT\n);\n", table_name);
    } else {
        printf("Usage: .schema [table_name]\n");
    }
}

void show_help() {
    // Display available meta commands
    printf("Available meta commands:\n");
    printf(".tables    - List all tables\n");
    printf(".schema [table_name] - Show schema of a table\n");
    printf(".exit      - Exit the program\n");
    printf(".help      - Show this help message\n");
}
```

2.0.6 Summary

1. **Read Input**: Read the user input.
2. **Check for Meta Commands**: If the input starts with a dot (`.`), treat it as a meta command.
3. **Handle Meta Commands**: Execute the appropriate function based on the meta command.

By following this approach, you can extend your database CLI to include useful meta commands, enhancing its functionality and making it more user-friendly.

3 What are the prepare statements? How can we implements in projects?

3.0.1 Prepared Statements

Prepared statements are a feature used to optimize the execution of SQL statements that are executed multiple times with different parameters. Instead of parsing and compiling the SQL statement each time it's executed, prepared statements allow the statement to be parsed and compiled once, and then executed multiple times with different parameter values. This provides performance benefits and helps to prevent SQL injection attacks.

3.0.2 Benefits of Prepared Statements

1. **Performance**: The SQL statement is parsed and compiled only once, reducing the overhead of repetitive parsing and compiling.

2. **Security**: Using parameterized queries helps to prevent SQL injection attacks.
3. **Convenience**: They provide a way to execute a statement multiple times with different parameters without rewriting the entire SQL statement.

3.0.3 Implementing Prepared Statements in Your Project

To implement prepared statements in your SQLite-like project, you need to follow these steps:

1. **Define the SQL Statement with Placeholders**
2. **Prepare the Statement**
3. **Bind Parameters to the Statement**
4. **Execute the Statement**
5. **Finalize the Statement**

3.0.4 Step-by-Step Implementation

1. Define the SQL Statement with Placeholders Placeholders (e.g., `?`) are used in the SQL statement to indicate where the parameters will be bound.

```
const char *sql = "INSERT INTO users (name, age) VALUES (?, ?)";
```

2. Prepare the Statement Use a function to prepare the SQL statement. This involves parsing and compiling the statement.

```
#include <sqlite3.h>
#include <stdio.h>

sqlite3 *db;
sqlite3_stmt *stmt;
const char *sql = "INSERT INTO users (name, age) VALUES (?, ?)";
int rc;

rc = sqlite3_prepare_v2(db, sql, -1, &stmt, NULL);
if (rc != SQLITE_OK) {
    printf("Failed to prepare statement\n");
    return;
}
```

3. Bind Parameters to the Statement Bind values to the placeholders in the prepared statement. This step allows you to execute the same SQL statement with different values.

```
const char *name = "Alice";
int age = 30;

sqlite3_bind_text(stmt, 1, name, -1, SQLITE_STATIC);
sqlite3_bind_int(stmt, 2, age);
```

In this example, `sqlite3_bind_text` binds a string to the first placeholder, and `sqlite3_bind_int` binds an integer to the second placeholder.

4. Execute the Statement Execute the prepared statement.

```
rc = sqlite3_step(stmt);
if (rc != SQLITE_DONE) {
    printf("Execution failed\n");
} else {
    printf("Execution succeeded\n");
}
```

5. Finalize the Statement Finalize the statement to release the resources associated with it.

```
sqlite3_finalize(stmt);
```

3.0.5 Example: Using Prepared Statements in a Complete Program

Here's a complete example that demonstrates how to use prepared statements in a simple SQLite program:

```
#include <stdio.h>
#include <sqlite3.h>

void insert_user(sqlite3 *db, const char *name, int age) {
    sqlite3_stmt *stmt;
    const char *sql = "INSERT INTO users (name, age) VALUES (?, ?)";
    int rc;

    rc = sqlite3_prepare_v2(db, sql, -1, &stmt, NULL);
    if (rc != SQLITE_OK) {
        printf("Failed to prepare statement: %s\n", sqlite3_errmsg(db));
        return;
    }

    sqlite3_bind_text(stmt, 1, name, -1, SQLITE_STATIC);
    sqlite3_bind_int(stmt, 2, age);

    rc = sqlite3_step(stmt);
    if (rc != SQLITE_DONE) {
        printf("Execution failed: %s\n", sqlite3_errmsg(db));
    } else {
        printf("Execution succeeded\n");
    }

    sqlite3_finalize(stmt);
}

int main() {
    sqlite3 *db;
    int rc;
```

```
    rc = sqlite3_open("test.db", &db);
    if (rc) {
        printf("Can't open database: %s\n", sqlite3_errmsg(db));
        return rc;
    }

    const char *create_table_sql = "CREATE TABLE IF NOT EXISTS users (id INTEGER PRIMARY KEY, n
    rc = sqlite3_exec(db, create_table_sql, 0, 0, 0);
    if (rc != SQLITE_OK) {
        printf("Failed to create table: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);
        return rc;
    }

    insert_user(db, "Alice", 30);
    insert_user(db, "Bob", 25);

    sqlite3_close(db);
    return 0;
}
```

3.0.6 Summary

1. **Define the SQL Statement with Placeholders**: Use `?` as placeholders in the SQL statement.
2. **Prepare the Statement**: Parse and compile the SQL statement once.
3. **Bind Parameters**: Bind different values to the placeholders for each execution.
4. **Execute the Statement**: Execute the prepared statement.
5. **Finalize the Statement**: Release resources associated with the statement after execution.

This approach ensures better performance, security, and convenience when executing repeated SQL operations in your project.

4 Thank You!