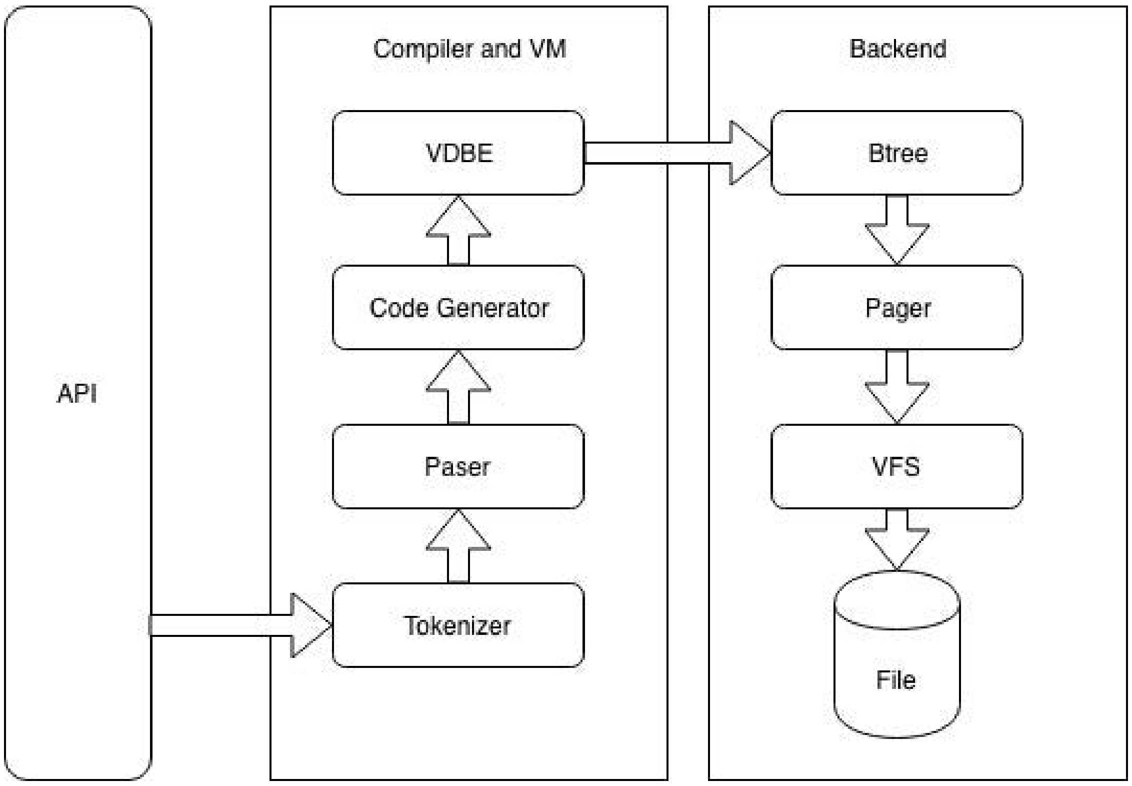
Introduction_and_Setting_up_REPL

July 15, 2024

1 How SQLite Work?

```
[ ]: from IPython.display import Image
     import warnings
     warnings.filterwarnings('ignore')
     Image(".\Images\SQL1.jpg")
```

[ ]:



SQLite is a lightweight, embedded SQL database engine. It's widely used in various applications, from mobile devices to desktop applications. Here's a detailed step-by-step explanation of how SQLite works, covering its architecture and key components:

1.0.1 1. Initialization and Setup

a. Initialization When an SQLite database is opened, it initializes its internal data structures and reads the database schema. The schema is a definition of the database structure, including tables, indexes, and other entities.

b. File Format SQLite databases are stored in a single cross-platform disk file. The file format is stable, meaning that databases created by older versions of SQLite can be read by newer versions and vice versa.

1.0.2 2. Parsing SQL Statements

a. Tokenization The first step in executing an SQL statement is tokenization. The SQL statement is broken down into individual tokens (keywords, identifiers, operators, etc.).

b. Parsing Tokens are then fed into the parser, which is generated by a parser generator tool called Lemon (similar to Yacc or Bison). The parser converts the tokenized SQL statement into a parse tree, which represents the syntactic structure of the statement.

c. Semantic Analysis The parse tree is analyzed to check for semantic correctness. This includes checking if the tables and columns mentioned in the statement exist, verifying data types, and other logical checks.

1.0.3 3. Query Planning and Optimization

a. Query Plan The parse tree is transformed into a query plan. A query plan is a sequence of steps that SQLite will execute to perform the query. This is similar to a program for a virtual machine.

b. Optimization SQLite uses various optimization techniques to make the query execution more efficient. It may reorder operations, use indexes, and apply other transformations to speed up the execution.

1.0.4 4. Bytecode Generation

The query plan is compiled into bytecode, which is a low-level set of instructions for the SQLite virtual machine (VM). The bytecode instructions are designed to be simple and compact, making them easy to interpret and execute.

1.0.5 5. Execution

a. Virtual Machine (VM) SQLite uses a virtual machine to execute the bytecode. The VM iterates through the bytecode instructions and performs the necessary operations to execute the SQL statement.

b. Storage Engine The storage engine is responsible for reading from and writing to the database file. It handles tasks such as managing free space, ensuring data integrity, and performing I/O operations.

1.0.6 6. Data Storage and Retrieval

a. B-Trees SQLite uses B-Trees to store table and index data. B-Trees are balanced tree data structures that allow for efficient insertion, deletion, and lookup operations.

b. Pager Module The pager module is responsible for managing the database file's pages. A page is the smallest unit of storage in an SQLite database. The pager module handles reading pages from disk into memory, writing pages back to disk, and managing the page cache.

c. Database Pages Each page in the database file contains data organized in a specific format. Pages are used to store tables, indexes, and other database structures.

1.0.7 7. Transactions and Concurrency

a. ACID Properties SQLite ensures that all transactions are ACID (Atomic, Consistent, Isolated, Durable). This means that transactions are all-or-nothing, leave the database in a consistent state, are isolated from each other, and are durable even in the event of a crash.

b. Journaling To ensure durability and atomicity, SQLite uses a journaling mechanism. Before making changes to the database file, SQLite writes the changes to a journal. If a crash occurs, SQLite can use the journal to roll back or complete the transaction.

c. Locking SQLite uses file locking to manage concurrent access to the database. It supports multiple readers but only one writer at a time.

1.0.8 8. Finalization

After the SQL statement has been executed, SQLite finalizes the statement, releasing any resources associated with it. This includes deallocating memory and closing any open cursors or file handles.

1.0.9 Detailed Example: SELECT Statement Execution

1. **User Input**: `SELECT * FROM users WHERE id = 1;`
2. **Tokenization**:
   - Tokens: `SELECT`, `*`, `FROM`, `users`, `WHERE`, `id`, `=`, `1`, `;`
3. **Parsing**:
   - Parse tree representing the SQL syntax:

```
SelectStmt
    ResultColumn
         Star
    FromClause
         TableName: users
    WhereClause
         Expression
              ColumnName: id
              Operator: =
                              LiteralValue: 1
```

4. **Semantic Analysis**:
   - Check if table `users` exists.
   - Verify column `id` exists in `users`.
5. **Query Planning and Optimization**:
   - Generate query plan:

     ```
     QUERY PLAN
     |-- SCAN TABLE users
     `-- SEARCH TABLE users USING INTEGER PRIMARY KEY (id=?)
     ```

   - Optimization: Use index on `id` if available.
6. **Bytecode Generation**:
   - Generate bytecode:

     ```
     1: Init
     2: OpenRead
     3: Integer 1
     4: SeekGE
     5: Column
     6: ResultRow
     7: Halt
     ```

7. **Execution**:
   - Virtual machine executes bytecode instructions.
   - Storage engine retrieves data from the database file.
8. **Data Retrieval**:
   - Data read from B-Tree pages.
   - Page cache used for efficient access.
9. **Transaction Handling**:
   - Ensure ACID properties.
   - Use journaling for durability.
10. **Finalization**:
    - Release resources.
    - Close cursors.

By understanding each of these steps in detail, you can gain a deep understanding of how SQLite works internally and apply similar concepts to your own database implementation.

2 What is REPL?

REPL stands for **Read-Eval-Print Loop**. It is an interactive programming environment that takes user inputs, evaluates them, and returns the result to the user. The cycle continues until the user exits the program. REPLs are commonly used for programming languages and command-line interfaces to provide a quick and interactive way to write and test code.

2.0.1 Components of REPL

1. **Read**: The REPL reads the user input, which is usually a single line of code or command.
2. **Eval (Evaluate)**: The input is then evaluated or executed by the interpreter or compiler.
3. **Print**: The result of the evaluation is printed or displayed back to the user.
4. **Loop**: The loop starts again, waiting for the next input from the user.

2.0.2 Example of REPL

Python REPL When you run `python` in your terminal, you start a Python REPL:

```
$ python
Python 3.8.5 (default, Jul 20 2020, 15:54:26)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 1 + 1
2
>>> print("Hello, World!")
Hello, World!
>>>
```

In this example: - **Read**: The REPL reads `1 + 1`. - **Eval**: It evaluates the expression to `2`. - **Print**: It prints `2`. - **Loop**: It waits for the next input.

SQL REPL Example Many SQL databases, including SQLite, provide a REPL environment for executing SQL commands interactively.

```
$ sqlite3 test.db
SQLite version 3.32.3 2020-06-18 14:00:33
Enter ".help" for usage hints.
sqlite> CREATE TABLE users (id INTEGER PRIMARY KEY, name TEXT);
sqlite> INSERT INTO users (name) VALUES ('Alice');
sqlite> SELECT * FROM users;
1|Alice
sqlite>
```

In this example: - **Read**: The REPL reads `CREATE TABLE users (id INTEGER PRIMARY KEY, name TEXT);`. - **Eval**: It evaluates and executes the SQL command. - **Print**: It prints any result (e.g., the output of `SELECT * FROM users;`). - **Loop**: It waits for the next input.

2.0.3 Benefits of REPL

- **Immediate Feedback**: Allows for quick testing and debugging of code.
- **Interactive Learning**: Great for beginners to experiment and learn a new language.
- **Prototyping**: Useful for rapid development and testing of small code snippets or ideas.

2.0.4 Implementation of a Simple REPL in C

Here's a simple example of a REPL in C that reads a line of input and echoes it back to the user:

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

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

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

void process_input(const char *input) {
    if (strncmp(input, "exit", 4) == 0) {
        printf("Goodbye!\n");
        exit(0);
    } else {
        printf("Unrecognized command: %s", input);
    }
}

int main() {
    while (1) {
        print_prompt();
        char *input = read_input();
        process_input(input);
        free(input);
    }
    return 0;
}
```

In this example: - **Read**: The `read_input` function reads a line of input from the user. - **Eval**: The `process_input` function processes the input. - **Print**: If the input is unrecognized, it prints an error message. - **Loop**: The `while (1)` loop repeats the process.

3 Main method():

[ ]:
```c
int main(int argc, char* argv[]) {
  InputBuffer* input_buffer = new_input_buffer();
  while (true) {
    print_prompt();
    read_input(input_buffer);

    if (strcmp(input_buffer->buffer, ".exit") == 0) {
      close_input_buffer(input_buffer);
      exit(EXIT_SUCCESS);
    } else {
      printf("Unrecognized command '%s'.\n", input_buffer->buffer);
    }
  }
}
```

This C code snippet outlines a simple REPL (Read-Eval-Print Loop) structure within a `main` function. It's designed to continuously accept user input, compare it to a predefined command (in this case, ".exit"), and perform actions based on that input. Here's a detailed breakdown:

1. **`int main(int argc, char* argv[]) {`**
   - This is the entry point of any C program. `argc` represents the number of command-line arguments passed to the program, and `argv` is an array of strings (character pointers) representing the arguments themselves.
2. **`InputBuffer* input_buffer = new_input_buffer();`**
   - An `InputBuffer` type pointer named `input_buffer` is declared and initialized. Presumably, `InputBuffer` is a custom data structure designed to hold user input, and `new_input_buffer()` is a function that allocates memory for an `InputBuffer` and returns a pointer to it. This part of the code is not shown but is crucial for understanding how input is managed.
3. **`while (true) {`**
   - This introduces an infinite loop, meaning the code inside this loop will execute repeatedly until the program is explicitly exited (via the `exit` function call) or terminated externally.
4. **`print_prompt();`**
   - A function call to `print_prompt()`, which likely prints a prompt to the console to let the user know the program is ready to accept input. The specifics of what is printed are not shown but could be something simple like `>`.
5. **`read_input(input_buffer);`**
   - Calls the `read_input` function, passing it the `input_buffer` pointer. This function is responsible for reading user input from the console and storing it in the structure pointed to by `input_buffer`. The details of how this is implemented are not provided.
6. **`if (strcmp(input_buffer->buffer, ".exit") == 0) {`**
   - Uses the `strcmp` function to compare the user input stored in `input_buffer->buffer` against the string `".exit"`. If they are equal (`strcmp` returns `0` for equal strings), the condition is true.
7. **`close_input_buffer(input_buffer);`**
   - If the user has entered `.exit`, this line calls `close_input_buffer`, a function presumably designed to free any resources associated with `input_buffer`. This is an example of proper resource management, ensuring no memory leaks occur.
8. **`exit(EXIT_SUCCESS);`**
   - Calls the `exit` function with `EXIT_SUCCESS`, a macro that indicates successful program termination. This exits the program.
9. **`} else {`**

    - If the user input does not match `".exit"`, the `else` block is executed.
10. **`printf("Unrecognized command '%s'.\n", input_buffer->buffer);`**
    - Prints a message to the console indicating that the entered command is unrecognized, echoing back the user's input. This serves as simple feedback to the user.
11. **`}`**
    - Marks the end of the `else` block.
12. **The loop then repeats**, waiting for more user input.

This code is a basic framework for a command-line application that waits for user input, checks for a specific command (`.exit`), and exits if that command is entered. Otherwise, it informs the user that the command is unrecognized. This pattern is foundational for many interactive applications, including shells and other command-line utilities.

4 InputBuffer?

[ ]:
```
typedef struct {
  char* buffer;
  size_t buffer_length;
  ssize_t input_length;
} InputBuffer;

InputBuffer* new_input_buffer() {
  InputBuffer* input_buffer = (InputBuffer*)malloc(sizeof(InputBuffer));
  input_buffer->buffer = NULL;
  input_buffer->buffer_length = 0;
  input_buffer->input_length = 0;

  return input_buffer;
}
```

This C code snippet defines a structure named `InputBuffer` and a function `new_input_buffer()` that initializes and returns a pointer to an `InputBuffer` instance. Here's a detailed breakdown:

4.0.1 `InputBuffer` Structure

- **`typedef struct { ... } InputBuffer;`**
  - This defines a new type `InputBuffer` as a `struct` (structure). Structures in C are used to group related variables under a single name, allowing for more organized and manageable code, especially when dealing with complex data.
- **`char* buffer;`**
  - This member variable is a pointer to a `char` (character), essentially representing a string or an array of characters. It's intended to store the actual user input.
- **`size_t buffer_length;`**
  - A variable of type `size_t` (an unsigned integer type) that stores the length of the `buffer`. This is not the length of the string stored in `buffer` but the total allocated size of the `buffer` array.
- **`ssize_t input_length;`**
  - A signed size type (`ssize_t`) variable that holds the length of the input stored in `buffer`.

The use of a signed type (`ssize_t` vs. `size_t`) allows this variable to potentially store a negative value, which could be used to indicate an error or special condition.

4.0.2 `new_input_buffer()` Function

- **`InputBuffer* new_input_buffer() { ... }`**
  - This function creates a new `InputBuffer` instance, initializes it, and returns a pointer to it. It does not take any parameters.
- **`InputBuffer* input_buffer = (InputBuffer*)malloc(sizeof(InputBuffer));`**
  - Allocates memory on the heap for an `InputBuffer` instance using `malloc`, which takes the size of an `InputBuffer` as an argument. The result of `malloc` is cast to `(InputBuffer*)` to match the type. This line initializes the `input_buffer` variable as a pointer to the allocated `InputBuffer`.
- **`input_buffer->buffer = NULL;`**
  - Initializes the `buffer` member of `input_buffer` to `NULL`, indicating that it currently points to no memory location. This is a safe initial state, preventing accidental dereferencing of an uninitialized pointer.
- **`input_buffer->buffer_length = 0;`**
  - Sets the `buffer_length` member to `0`, reflecting that no memory has been allocated to `buffer` yet.
- **`input_buffer->input_length = 0;`**
  - Initializes `input_length` to `0`, indicating that no user input has been stored in `buffer`.
- **`return input_buffer;`**
  - Returns the pointer to the newly created and initialized `InputBuffer` instance.

This function is typically called at the beginning of a program that requires dynamic input handling. It sets up an `InputBuffer` with no allocated memory for the buffer itself, ready to be allocated as needed when user input is read. This approach allows for flexible handling of input of varying lengths, as the buffer can be resized to accommodate the input.

5 print_prompt():

[ ]:
```
void print_prompt() { printf("db > "); }
```

The `print_prompt` function is a simple utility function in C that prints a prompt to the standard output (typically the console). Here's a breakdown of its components:

- **`void print_prompt() { ... }`**: This defines a function named `print_prompt` that takes no parameters (`void`) and returns no value (`void`). It's a standalone function meant to be called without needing any arguments.

- **`printf("db > ");`**: Inside the function, the `printf` function is called with a single string argument `"db > "`. `printf` is a standard input/output library function that prints the given string to the standard output. In this case, it prints `db >`, which is intended to be a prompt for the user. This prompt suggests that the program is a database application waiting for commands or queries from the user.

The purpose of `print_prompt` is to visually indicate to the user that the program is ready to accept input. It's a common practice in command-line applications and REPLs (Read-Eval-Print Loops) to provide such prompts to improve user experience by making the interface more interactive and user-friendly.

6 getline()?

[ ]:
```
ssize_t getline(char **lineptr, size_t *n, FILE *stream);
```

The `getline` function is a standard library function in C that reads an entire line from a stream, storing the address of the buffer containing the text into `*lineptr`. The buffer is null-terminated and includes the newline character, if one was found. Here's a breakdown of its parameters and behavior:

6.0.1 Parameters

- **`char **lineptr`**: A pointer to a buffer where the read line will be stored. This buffer is dynamically allocated or resized as necessary to hold the line contents, including the null-terminating character. If `*lineptr` is `NULL`, `getline` will allocate a new buffer, making `*lineptr` point to it.
- **`size_t *n`**: A pointer to a variable that stores the size of the buffer pointed to by `*lineptr`. If the buffer is not large enough to hold the line, `getline` will reallocate it with a larger size and update `*n` to reflect the new size.
- **`FILE *stream`**: The input stream from which to read the line. This could be a file, standard input (`stdin`), or any other stream opened with the standard C I/O library.

6.0.2 Return Value

- **`ssize_t`**: The function returns the number of characters read, including the newline character, but not including the terminating null byte. If an error occurs or end of file is reached while no characters have been read, `getline` returns `-1`.

6.0.3 Behavior

- `getline` reads from the specified `stream` until either a newline character (`'\n'`) is encountered or an end-of-file (EOF) condition is reached.
- If `*lineptr` is initially `NULL`, or if the buffer is not large enough to hold the line, `getline` will allocate or reallocate the buffer as needed. This means the caller is responsible for freeing the allocated buffer (using `free()`) when it is no longer needed to avoid memory leaks.
- The newline character, if read, is included in the buffer before the null-terminating character.
- After a successful call, `*lineptr` points to the buffer containing the read line, and `*n` reflects the buffer's size.

6.0.4 Example Usage

[ ]:
```
#include <stdio.h>
#include <stdlib.h>

int main() {
    char *line = NULL;
    size_t len = 0;
    ssize_t read;

    printf("Enter a line: ");
    read = getline(&line, &len, stdin);

    if (read != -1) {
        printf("You entered: %s", line);
    }

    free(line); // Free the allocated buffer
    return 0;
}
```

This function is particularly useful for reading lines of varying lengths, as it handles memory allocation automatically, resizing the buffer as needed to accommodate the line length.

7 read_input()?

[ ]:
```
void read_input(InputBuffer* input_buffer) {
  ssize_t bytes_read =
      getline(&(input_buffer->buffer), &(input_buffer->buffer_length), stdin);

  if (bytes_read <= 0) {
    printf("Error reading input\n");
    exit(EXIT_FAILURE);
  }

  // Ignore trailing newline
  input_buffer->input_length = bytes_read - 1;
  input_buffer->buffer[bytes_read - 1] = 0;
}
```

The `read_input` function is designed to read a line of text from the standard input (`stdin`) and store it in an `InputBuffer` structure. Here's a detailed explanation of its components:

7.0.1 Parameters

- **`InputBuffer* input_buffer`**: A pointer to an `InputBuffer` structure where the read line will be stored. This structure is expected to have at least three members: `buffer` (a pointer to the character array), `buffer_length` (the allocated size of `buffer`), and `input_length` (the length of the input read).

7.0.2 Process

1. **`ssize_t bytes_read = getline(&(input_buffer->buffer), &(input_buffer->buffer_length), stdin);`**

   - This line calls the `getline` function, passing the address of the `buffer` and `buffer_length` members of `input_buffer`, along with `stdin` as the input stream. `getline` will read a line from the standard input, allocate or resize the `buffer` as necessary (updating `buffer_length` accordingly), and return the number of bytes read.
2. **`if (bytes_read <= 0) {`**
   - After reading, it checks if `bytes_read` is less than or equal to `0`. A return value of `0` or less indicates an error or end-of-file (EOF) condition without reading any characters. In such cases, the function prints an error message and exits the program with `EXIT_FAILURE`.
3. **`input_buffer->input_length = bytes_read - 1;`**
   - If the read was successful, this line updates the `input_length` of the `input_buffer` to be one less than `bytes_read`. This adjustment is made to exclude the newline character (`\n`) from the input length, assuming `getline` includes the newline in the count.
4. **`input_buffer->buffer[bytes_read - 1] = 0;`**
   - This line replaces the newline character at the end of the input with a null terminator (`\0`), effectively removing the newline from the stored string. This is important for processing commands or input where the newline is not desired as part of the input.

7.0.3 Summary

The `read_input` function is a utility for reading a line of input into a dynamically managed buffer within an `InputBuffer` structure, handling memory allocation automatically and preparing the input for further processing by removing the trailing newline.

8 close_input_buffer()?

[ ]:
```
void close_input_buffer(InputBuffer* input_buffer) {
    free(input_buffer->buffer);
    free(input_buffer);
}
```

The `close_input_buffer` function is designed to release the memory allocated for an `InputBuffer` structure and its associated `buffer`. Here's a breakdown of its operations:

8.0.1 Parameters

- **`InputBuffer* input_buffer`**: A pointer to an `InputBuffer` structure. This structure is expected to have at least a `buffer` member, which is a dynamically allocated array of characters, and possibly other metadata such as buffer length and input length.

8.0.2 Operations

1. **`free(input_buffer->buffer);`**
   - This line releases the memory allocated for the `buffer` member of the `InputBuffer` structure. The `buffer` is dynamically allocated (likely using `malloc` or a similar function) to hold input data. It's crucial to `free` this memory when it's no longer needed to avoid memory leaks.
2. **`free(input_buffer);`**

- After freeing the `buffer`, this line releases the memory allocated for the `InputBuffer` structure itself. Since `input_buffer` is a pointer to a dynamically allocated `InputBuffer`, failing to free it would also result in a memory leak.

8.0.3 Summary

The `close_input_buffer` function ensures that all memory allocated for an `InputBuffer` and its internal `buffer` is properly released, preventing memory leaks. This function should be called when an `InputBuffer` is no longer needed, typically at the end of its usage or before the program terminates.

9 Wrap Up All Function?

[ ]:
```
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct {
  char* buffer;
  size_t buffer_length;
  ssize_t input_length;
} InputBuffer;

InputBuffer* new_input_buffer() {
  InputBuffer* input_buffer = malloc(sizeof(InputBuffer));
  input_buffer->buffer = NULL;
  input_buffer->buffer_length = 0;
  input_buffer->input_length = 0;

  return input_buffer;
}

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

void read_input(InputBuffer* input_buffer) {
  ssize_t bytes_read =
      getline(&(input_buffer->buffer), &(input_buffer->buffer_length), stdin);

  if (bytes_read <= 0) {
    printf("Error reading input\n");
    exit(EXIT_FAILURE);
  }

  // Ignore trailing newline
  input_buffer->input_length = bytes_read - 1;
  input_buffer->buffer[bytes_read - 1] = 0;
}

void close_input_buffer(InputBuffer* input_buffer) {
    free(input_buffer->buffer);
    free(input_buffer);
}

int main(int argc, char* argv[]) {
  InputBuffer* input_buffer = new_input_buffer();
  while (true) {
    print_prompt();
    read_input(input_buffer);

    if (strcmp(input_buffer->buffer, ".exit") == 0) {
      close_input_buffer(input_buffer);
      exit(EXIT_SUCCESS);
    } else {
      printf("Unrecognized command '%s'.\n", input_buffer->buffer);
    }
  }
}
```

This C program implements a simple Read-Eval-Print Loop (REPL) for a database application. It demonstrates dynamic memory management, string handling, and basic control flow. Here's a step-by-step explanation:

1. **Includes and Definitions**
   - Includes standard libraries for boolean values, input/output operations, memory allocation, and string manipulation.
   - Defines a `struct` named `InputBuffer` to hold user input, including the buffer itself, its length, and the length of the input.
2. **Function `new_input_buffer`**
   - Allocates memory for an `InputBuffer` instance and initializes its members. Returns a pointer to the newly allocated `InputBuffer`.
3. **Function `print_prompt`**
   - Prints a prompt (`"db > "`) to the standard output, indicating the program is ready to accept input.
4. **Function `read_input`**
   - Reads a line of input from the standard input (`stdin`) into the `InputBuffer` passed as an argument. It handles dynamic memory allocation for the input buffer and trims the trailing newline character.
5. **Function `close_input_buffer`**
   - Frees the memory allocated for the `InputBuffer`'s buffer and the `InputBuffer` itself, cleaning up resources before the program exits.
6. **Function `main`**
   - Initializes an `InputBuffer` and enters an infinite loop to:
     - Print the prompt.
     - Read input into the `InputBuffer`.

  - Check if the input is the command `.exit`. If so, it cleans up and exits the program successfully.
  - If the input is not recognized, it prints an error message.

This program serves as a foundational framework for a database application, where commands can be extended beyond `.exit`. The REPL structure allows for interactive command execution, and the dynamic memory management ensures flexibility in handling user input of varying lengths.

10 Thank You!