International Journal of Research in Engineering, Science and Management
Volume-3, Issue-12, December-2020
https://www.ijresm.com | ISSN (Online): 2581-5792

Implementation of B-Tree based Database using C Programming Language

Pramita Kastha*

Student, Department of Electronics and Communication Engineering, Indian Institute of Technology, Kharagpur, Kharagpur, India

*Corresponding author: coolpro18062001@gmail.com

***Abstract*: A database is crucial to any organization to store, add, modify and retrieve details about the company. Additionally, they form the core of the backend of any imaginable web application that we see around us.**

Thus, a thorough understanding of internal working a database is necessary for creating meaningful software, or for making full use of the DBMS. We attempt to do so by implementing a simple B-Tree based database, following the SQLite architecture.

***Keywords*: B-Tree, C programming language, DBMS, Relational database, SQLite.**

1. Introduction

Databases, or more specifically, relational databases, helps in storing and taking care or large amounts of data, and also to 'relate' different kinds of information based on one or more common properties or attributes they share. Every database has a database engine, which offers a multitude of services such as sorting, saving, changing or serving the information on the database. And finally, a database system is a computer program for managing electronic databases. Databases are the backbone of many businesses currently operational around the world.

The motivation of this project is to gain a better insight into the structure of an RDBMS by building one. This would help one answer a lot of questions like:

- What format is the data saved in? (in memory and on disk)
- When does it move from memory to disk?
- Why can there be only one primary key per table?
- How does rolling back a transaction work?
- How are indexes formatted?
- When and how does a full table scan happen?
- What format is a prepared statement saved in?

We will be using C Programming language for the implementation. The entire database in stored into a single file which must be provided as a command line argument.

2. The SQLite Architecture

A query goes through a chain of components in order to retrieve of modify data. The front end consists of:

- Tokenizer
- Parser
- Code generator

The back end consists of the:

- Virtual machine
- B-tree
- Pager
- OS interface

The virtual machine takes bytecode generated by the front-end as instructions. It can then perform operations on one or more tables or indexes, each of which is stored in a data structure called a B-Tree. The VM is essentially a big switch statement on the type of bytecode instruction.

Each B-tree consists of many nodes. Each node is one page in length. The B-tree can retrieve a page from disk or save it back to disk by issuing commands to the pager.

The pager receives commands to read or write pages of data. It is a responsible for reading/writing at appropriate offsets in the database file. It also keeps a cache of recently-accessed pages in memory, and determines when those pages need to be written back to disk.

The OS interface is the layer that differs depending on which operating system SQLite was compiled for.

3. Implementation

We start with a simple read-execute-print-loop (REPL). For that, our main function will have an infinite loop that prints the prompt, gets a line of input, then processes that line of input.

Next, we add functionality to support meta-commands (Non-SQL commands, like '.exit' are called meta-commands). Also, we display an 'unrecognized command' message to handle the exception where one enters an invalid or unsupported meta-command.

Now, we take turn to implement executable commands like insertion. For, this we impose some restriction and define a fixed schema for only a single table which it supports. The schema is shown in the table 1.

At this stage, we create an array to store the data. Following which the 'select' and 'insert' operations are defined to be able read/write data from/to the array-based table data structure, using an abstraction known as the 'Pager'. Note that, currently, data is available only as long as the program runs. Hence, now we shift our focus on to be able to write the data in a file so that we be able to open it again for modification using our program. Using file handling and the pager abstraction, we implement the functionality for the program to pass a filename as a command line argument to write the data into.

Table 1
Database Schema

Column	Type
id	integer
username	varchar (32)
email	varchar (255)

At this point, we have a fully-functional program which we can use to insert rows into a table, store the data and also view it.

Next, we refactor the code to change the array data structure to store entries to a B-Tree data structure. B-Tree has the advantage of providing the functionality to store large amount of data (many more rows as compared to the array-based database).

For the above task, we begin by coding the 'Cursor' abstraction. The cursor will be used to access the row it is pointing to. Also, one needs to create a cursor at the beginning of the table, and we must be able to advance it to the next row.

The basic idea behind the B-Tree is as follows: a particular node in it can only store a certain amount of data, or rather just a specific no. of key-value pairs. It all starts with a single, root node, and once the limit is reached, new leaf nodes are added. This process occurs recursively with the addition of more rows.

4. The Code

Following is the code for the program:

```
#include <errno.h>
#include <fcntl.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

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

typedef enum {
  EXECUTE_SUCCESS,
  EXECUTE_DUPLICATE_KEY,
} ExecuteResult;

typedef enum {
  META_COMMAND_SUCCESS,
  META_COMMAND_UNRECOGNIZED_COMMAND
} MetaCommandResult;

typedef enum {
  PREPARE_SUCCESS,
  PREPARE_NEGATIVE_ID,
  PREPARE_STRING_TOO_LONG,
  PREPARE_SYNTAX_ERROR,
  PREPARE_UNRECOGNIZED_STATEMENT
} PrepareResult;

typedef enum { STATEMENT_INSERT,
STATEMENT_SELECT } StatementType;

#define COLUMN_USERNAME_SIZE 32
#define COLUMN_EMAIL_SIZE 255
typedef struct {
  uint32_t id;
  char username[COLUMN_USERNAME_SIZE + 1];
  char email[COLUMN_EMAIL_SIZE + 1];
} Row;

typedef struct {
  StatementType type;
  Row row_to_insert;  // only used by insert statement
} Statement;

#define size_of_attribute(Struct, Attribute)
sizeof(((Struct*)0)->Attribute)

const uint32_t ID_SIZE = size_of_attribute(Row, id);
const uint32_t USERNAME_SIZE =
size_of_attribute(Row, username);
const uint32_t EMAIL_SIZE = size_of_attribute(Row,
email);
const uint32_t ID_OFFSET = 0;
const uint32_t USERNAME_OFFSET = ID_OFFSET +
ID_SIZE;
const uint32_t EMAIL_OFFSET = USERNAME_OFFSET
+ USERNAME_SIZE;
const uint32_t ROW_SIZE = ID_SIZE +
USERNAME_SIZE + EMAIL_SIZE;

const uint32_t PAGE_SIZE = 4096;
#define TABLE_MAX_PAGES 100

typedef struct {
  int file_descriptor;
  uint32_t file_length;
  uint32_t num_pages;
  void* pages[TABLE_MAX_PAGES];
} Pager;
```

```
typedef struct {
  Pager* pager;
  uint32_t root_page_num;
} Table;

typedef struct {
  Table* table;
  uint32_t page_num;
  uint32_t cell_num;
  bool end_of_table;  // Indicates a position one past the last
element
} Cursor;

void print_row(Row* row) {
  printf("(%d, %s, %s)\n", row->id, row->username, row-
>email);
}

typedef enum { NODE_INTERNAL, NODE_LEAF }
NodeType;

/*
 * Common Node Header Layout
 */
const uint32_t NODE_TYPE_SIZE = sizeof(uint8_t);
const uint32_t NODE_TYPE_OFFSET = 0;
const uint32_t IS_ROOT_SIZE = sizeof(uint8_t);
const uint32_t IS_ROOT_OFFSET = NODE_TYPE_SIZE;
const uint32_t PARENT_POINTER_SIZE =
sizeof(uint32_t);
const uint32_t PARENT_POINTER_OFFSET =
IS_ROOT_OFFSET + IS_ROOT_SIZE;
const uint8_t COMMON_NODE_HEADER_SIZE =
    NODE_TYPE_SIZE + IS_ROOT_SIZE +
PARENT_POINTER_SIZE;

/*
 * Internal Node Header Layout
 */
const uint32_t INTERNAL_NODE_NUM_KEYS_SIZE =
sizeof(uint32_t);
const uint32_t
INTERNAL_NODE_NUM_KEYS_OFFSET =
COMMON_NODE_HEADER_SIZE;
const uint32_t INTERNAL_NODE_RIGHT_CHILD_SIZE
= sizeof(uint32_t);
const uint32_t
INTERNAL_NODE_RIGHT_CHILD_OFFSET =
    INTERNAL_NODE_NUM_KEYS_OFFSET +
INTERNAL_NODE_NUM_KEYS_SIZE;
const uint32_t INTERNAL_NODE_HEADER_SIZE =
COMMON_NODE_HEADER_SIZE +

INTERNAL_NODE_NUM_KEYS_SIZE +
INTERNAL_NODE_RIGHT_CHILD_SIZE;

/*
 * Internal Node Body Layout
 */
const uint32_t INTERNAL_NODE_KEY_SIZE =
sizeof(uint32_t);
const uint32_t INTERNAL_NODE_CHILD_SIZE =
sizeof(uint32_t);
const uint32_t INTERNAL_NODE_CELL_SIZE =
    INTERNAL_NODE_CHILD_SIZE +
INTERNAL_NODE_KEY_SIZE;
/* Keep this small for testing */
const uint32_t INTERNAL_NODE_MAX_CELLS = 3;

/*
 * Leaf Node Header Layout
 */
const uint32_t LEAF_NODE_NUM_CELLS_SIZE =
sizeof(uint32_t);
const uint32_t LEAF_NODE_NUM_CELLS_OFFSET =
COMMON_NODE_HEADER_SIZE;
const uint32_t LEAF_NODE_NEXT_LEAF_SIZE =
sizeof(uint32_t);
const uint32_t LEAF_NODE_NEXT_LEAF_OFFSET =
    LEAF_NODE_NUM_CELLS_OFFSET +
LEAF_NODE_NUM_CELLS_SIZE;
const uint32_t LEAF_NODE_HEADER_SIZE =
COMMON_NODE_HEADER_SIZE +

LEAF_NODE_NUM_CELLS_SIZE +

LEAF_NODE_NEXT_LEAF_SIZE;

/*
 * Leaf Node Body Layout
 */
const uint32_t LEAF_NODE_KEY_SIZE =
sizeof(uint32_t);
const uint32_t LEAF_NODE_KEY_OFFSET = 0;
const uint32_t LEAF_NODE_VALUE_SIZE =
ROW_SIZE;
const uint32_t LEAF_NODE_VALUE_OFFSET =
    LEAF_NODE_KEY_OFFSET +
LEAF_NODE_KEY_SIZE;
const uint32_t LEAF_NODE_CELL_SIZE =
LEAF_NODE_KEY_SIZE + LEAF_NODE_VALUE_SIZE;
const uint32_t LEAF_NODE_SPACE_FOR_CELLS =
PAGE_SIZE - LEAF_NODE_HEADER_SIZE;
const uint32_t LEAF_NODE_MAX_CELLS =
    LEAF_NODE_SPACE_FOR_CELLS /
LEAF_NODE_CELL_SIZE;
const uint32_t LEAF_NODE_RIGHT_SPLIT_COUNT =
(LEAF_NODE_MAX_CELLS + 1) / 2;
```

```
  const uint32_t LEAF_NODE_LEFT_SPLIT_COUNT =
      (LEAF_NODE_MAX_CELLS + 1) -
LEAF_NODE_RIGHT_SPLIT_COUNT;

  NodeType get_node_type(void* node) {
    uint8_t value = *((uint8_t*)(node +
NODE_TYPE_OFFSET));
    return (NodeType)value;
  }

  void set_node_type(void* node, NodeType type) {
    uint8_t value = type;
    *((uint8_t*)(node + NODE_TYPE_OFFSET)) = value;
  }

  bool is_node_root(void* node) {
    uint8_t value = *((uint8_t*)(node + IS_ROOT_OFFSET));
    return (bool)value;
  }

  void set_node_root(void* node, bool is_root) {
    uint8_t value = is_root;
    *((uint8_t*)(node + IS_ROOT_OFFSET)) = value;
  }

  uint32_t* node_parent(void* node) { return node +
PARENT_POINTER_OFFSET; }

  uint32_t* internal_node_num_keys(void* node) {
    return node +
INTERNAL_NODE_NUM_KEYS_OFFSET;
  }

  uint32_t* internal_node_right_child(void* node) {
    return node +
INTERNAL_NODE_RIGHT_CHILD_OFFSET;
  }

  uint32_t* internal_node_cell(void* node, uint32_t
cell_num) {
    return node + INTERNAL_NODE_HEADER_SIZE +
cell_num * INTERNAL_NODE_CELL_SIZE;
  }

  uint32_t* internal_node_child(void* node, uint32_t
child_num) {
    uint32_t num_keys = *internal_node_num_keys(node);
    if (child_num > num_keys) {
      printf("Tried to access child_num %d > num_keys
%d\n", child_num, num_keys);
      exit(EXIT_FAILURE);
    } else if (child_num == num_keys) {
      return internal_node_right_child(node);
    } else {
      return internal_node_cell(node, child_num);
    }
  }

  uint32_t* internal_node_key(void* node, uint32_t
key_num) {
    return (void*)internal_node_cell(node, key_num) +
INTERNAL_NODE_CHILD_SIZE;
  }

  uint32_t* leaf_node_num_cells(void* node) {
    return node + LEAF_NODE_NUM_CELLS_OFFSET;
  }

  uint32_t* leaf_node_next_leaf(void* node) {
    return node + LEAF_NODE_NEXT_LEAF_OFFSET;
  }

  void* leaf_node_cell(void* node, uint32_t cell_num) {
    return node + LEAF_NODE_HEADER_SIZE + cell_num
* LEAF_NODE_CELL_SIZE;
  }

  uint32_t* leaf_node_key(void* node, uint32_t cell_num) {
    return leaf_node_cell(node, cell_num);
  }

  void* leaf_node_value(void* node, uint32_t cell_num) {
    return leaf_node_cell(node, cell_num) +
LEAF_NODE_KEY_SIZE;
  }

  uint32_t get_node_max_key(void* node) {
    switch (get_node_type(node)) {
      case NODE_INTERNAL:
        return *internal_node_key(node,
*internal_node_num_keys(node) - 1);
      case NODE_LEAF:
        return *leaf_node_key(node,
*leaf_node_num_cells(node) - 1);
    }
  }

  void print_constants() {
    printf("ROW_SIZE: %d\n", ROW_SIZE);
    printf("COMMON_NODE_HEADER_SIZE: %d\n",
COMMON_NODE_HEADER_SIZE);
    printf("LEAF_NODE_HEADER_SIZE: %d\n",
LEAF_NODE_HEADER_SIZE);
    printf("LEAF_NODE_CELL_SIZE: %d\n",
LEAF_NODE_CELL_SIZE);
    printf("LEAF_NODE_SPACE_FOR_CELLS: %d\n",
LEAF_NODE_SPACE_FOR_CELLS);
    printf("LEAF_NODE_MAX_CELLS: %d\n",
LEAF_NODE_MAX_CELLS);
    }

    void* get_page(Pager* pager, uint32_t page_num) {
      if (page_num > TABLE_MAX_PAGES) {
        printf("Tried to fetch page number out of bounds. %d >
%d\n", page_num,
               TABLE_MAX_PAGES);
        exit(EXIT_FAILURE);
      }

      if (pager->pages[page_num] == NULL) {
        // Cache miss. Allocate memory and load from file.
        void* page = malloc(PAGE_SIZE);
        uint32_t num_pages = pager->file_length / PAGE_SIZE;

        // We might save a partial page at the end of the file
        if (pager->file_length % PAGE_SIZE) {
          num_pages += 1;
        }

        if (page_num <= num_pages) {
          lseek(pager->file_descriptor, page_num * PAGE_SIZE,
SEEK_SET);
          ssize_t bytes_read = read(pager->file_descriptor, page,
PAGE_SIZE);
          if (bytes_read == -1) {
            printf("Error reading file: %d\n", errno);
            exit(EXIT_FAILURE);
          }
        }

        pager->pages[page_num] = page;

        if (page_num >= pager->num_pages) {
          pager->num_pages = page_num + 1;
        }
      }

      return pager->pages[page_num];
    }

    void indent(uint32_t level) {
      for (uint32_t i = 0; i < level; i++) {
        printf("  ");
      }
    }

    void print_tree(Pager* pager, uint32_t page_num, uint32_t
indentation_level) {
      void* node = get_page(pager, page_num);
      uint32_t num_keys, child;

      switch (get_node_type(node)) {
        case (NODE_LEAF):
          num_keys = *leaf_node_num_cells(node);
          indent(indentation_level);
          printf("- leaf (size %d)\n", num_keys);
          for (uint32_t i = 0; i < num_keys; i++) {
            indent(indentation_level + 1);
            printf("- %d\n", *leaf_node_key(node, i));
          }
          break;
        case (NODE_INTERNAL):
          num_keys = *internal_node_num_keys(node);
          indent(indentation_level);
          printf("- internal (size %d)\n", num_keys);
          for (uint32_t i = 0; i < num_keys; i++) {
            child = *internal_node_child(node, i);
            print_tree(pager, child, indentation_level + 1);

            indent(indentation_level + 1);
            printf("- key %d\n", *internal_node_key(node, i));
          }
          child = *internal_node_right_child(node);
          print_tree(pager, child, indentation_level + 1);
          break;
      }
    }

    void serialize_row(Row* source, void* destination) {
      memcpy(destination + ID_OFFSET, &(source->id),
ID_SIZE);
      memcpy(destination + USERNAME_OFFSET, &(source-
>username), USERNAME_SIZE);
      memcpy(destination + EMAIL_OFFSET, &(source-
>email), EMAIL_SIZE);
    }

    void deserialize_row(void* source, Row* destination) {
      memcpy(&(destination->id), source + ID_OFFSET,
ID_SIZE);
      memcpy(&(destination->username), source +
USERNAME_OFFSET, USERNAME_SIZE);
      memcpy(&(destination->email), source +
EMAIL_OFFSET, EMAIL_SIZE);
    }

    void initialize_leaf_node(void* node) {
      set_node_type(node, NODE_LEAF);
      set_node_root(node, false);
      *leaf_node_num_cells(node) = 0;
      *leaf_node_next_leaf(node) = 0;  // 0 represents no sibling
    }

    void initialize_internal_node(void* node) {
      set_node_type(node, NODE_INTERNAL);
      set_node_root(node, false);
    *internal_node_num_keys(node) = 0;
  }

  Cursor* leaf_node_find(Table* table, uint32_t page_num,
uint32_t key) {
    void* node = get_page(table->pager, page_num);
    uint32_t num_cells = *leaf_node_num_cells(node);

    Cursor* cursor = malloc(sizeof(Cursor));
    cursor->table = table;
    cursor->page_num = page_num;
    cursor->end_of_table = false;

    // Binary search
    uint32_t min_index = 0;
    uint32_t one_past_max_index = num_cells;
    while (one_past_max_index != min_index) {
      uint32_t index = (min_index + one_past_max_index) / 2;
      uint32_t key_at_index = *leaf_node_key(node, index);
      if (key == key_at_index) {
        cursor->cell_num = index;
        return cursor;
      }
      if (key < key_at_index) {
        one_past_max_index = index;
      } else {
        min_index = index + 1;
      }
    }

    cursor->cell_num = min_index;
    return cursor;
  }

  uint32_t internal_node_find_child(void* node, uint32_t
key) {
    /*
    Return the index of the child which should contain
    the given key.
    */

    uint32_t num_keys = *internal_node_num_keys(node);

    /* Binary search */
    uint32_t min_index = 0;
    uint32_t max_index = num_keys; /* there is one more
child than key */

    while (min_index != max_index) {
      uint32_t index = (min_index + max_index) / 2;
      uint32_t key_to_right = *internal_node_key(node,
index);
      if (key_to_right >= key) {
        max_index = index;
      } else {
        min_index = index + 1;
      }
    }

    return min_index;
  }

  Cursor* internal_node_find(Table* table, uint32_t
page_num, uint32_t key) {
    void* node = get_page(table->pager, page_num);

    uint32_t child_index = internal_node_find_child(node,
key);
    uint32_t child_num = *internal_node_child(node,
child_index);
    void* child = get_page(table->pager, child_num);
    switch (get_node_type(child)) {
      case NODE_LEAF:
        return leaf_node_find(table, child_num, key);
      case NODE_INTERNAL:
        return internal_node_find(table, child_num, key);
    }
  }

  /*
  Return the position of the given key.
  If the key is not present, return the position
  where it should be inserted
  */
  Cursor* table_find(Table* table, uint32_t key) {
    uint32_t root_page_num = table->root_page_num;
    void* root_node = get_page(table->pager,
root_page_num);

    if (get_node_type(root_node) == NODE_LEAF) {
      return leaf_node_find(table, root_page_num, key);
    } else {
      return internal_node_find(table, root_page_num, key);
    }
  }

  Cursor* table_start(Table* table) {
    Cursor* cursor = table_find(table, 0);

    void* node = get_page(table->pager, cursor->page_num);
    uint32_t num_cells = *leaf_node_num_cells(node);
    cursor->end_of_table = (num_cells == 0);

    return cursor;
  }

  void* cursor_value(Cursor* cursor) {
    uint32_t page_num = cursor->page_num;
  void* page = get_page(cursor->table->pager, page_num);
  return leaf_node_value(page, cursor->cell_num);
}

void cursor_advance(Cursor* cursor) {
  uint32_t page_num = cursor->page_num;
  void* node = get_page(cursor->table->pager, page_num);

  cursor->cell_num += 1;
  if (cursor->cell_num >= (*leaf_node_num_cells(node))) {
    /* Advance to next leaf node */
    uint32_t next_page_num = *leaf_node_next_leaf(node);
    if (next_page_num == 0) {
      /* This was rightmost leaf */
      cursor->end_of_table = true;
    } else {
      cursor->page_num = next_page_num;
      cursor->cell_num = 0;
    }
  }
}

Pager* pager_open(const char* filename) {
  int fd = open(filename,
              O_RDWR |      // Read/Write mode
                  O_CREAT  // Create file if it does not exist
                // User read permission
              );

  if (fd == -1) {
    printf("Unable to open file\n");
    exit(EXIT_FAILURE);
  }

  off_t file_length = lseek(fd, 0, SEEK_END);

  Pager* pager = malloc(sizeof(Pager));
  pager->file_descriptor = fd;
  pager->file_length = file_length;
  pager->num_pages = (file_length / PAGE_SIZE);

  if (file_length % PAGE_SIZE != 0) {
    printf("Db file is not a whole number of pages. Corrupt
file.\n");
    exit(EXIT_FAILURE);
  }

  for (uint32_t i = 0; i < TABLE_MAX_PAGES; i++) {
    pager->pages[i] = NULL;
  }

  return pager;
}

Table* db_open(const char* filename) {
  Pager* pager = pager_open(filename);

  Table* table = malloc(sizeof(Table));
  table->pager = pager;
  table->root_page_num = 0;

  if (pager->num_pages == 0) {
    // New database file. Initialize page 0 as leaf node.
    void* root_node = get_page(pager, 0);
    initialize_leaf_node(root_node);
    set_node_root(root_node, true);
  }

  return table;
}

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);
}

void pager_flush(Pager* pager, uint32_t page_num) {
  if (pager->pages[page_num] == NULL) {
    printf("Tried to flush null page\n");
    exit(EXIT_FAILURE);
  }
```

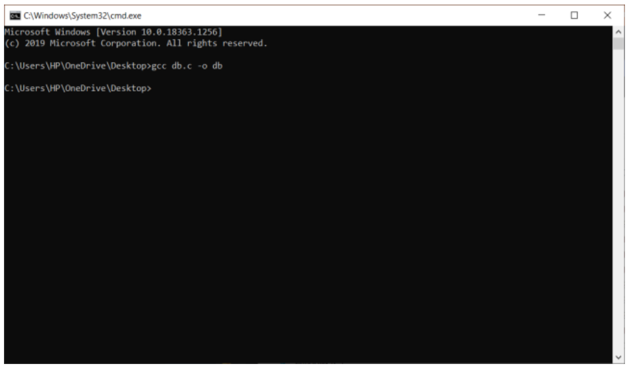
```
   off_t offset = lseek(pager->file_descriptor, page_num *
PAGE_SIZE, SEEK_SET);

   if (offset == -1) {
     printf("Error seeking: %d\n", errno);
     exit(EXIT_FAILURE);
   }

   ssize_t bytes_written =
       write(pager->file_descriptor, pager->pages[page_num],
PAGE_SIZE);

   if (bytes_written == -1) {
     printf("Error writing: %d\n", errno);
     exit(EXIT_FAILURE);
   }
 }

 void db_close(Table* table) {
   Pager* pager = table->pager;

   for (uint32_t i = 0; i < pager->num_pages; i++) {
     if (pager->pages[i] == NULL) {
       continue;
     }
     pager_flush(pager, i);
     free(pager->pages[i]);
     pager->pages[i] = NULL;
   }

   int result = close(pager->file_descriptor);
   if (result == -1) {
     printf("Error closing db file.\n");
     exit(EXIT_FAILURE);
   }
   for (uint32_t i = 0; i < TABLE_MAX_PAGES; i++) {
     void* page = pager->pages[i];
     if (page) {
       free(page);
       pager->pages[i] = NULL;
     }
   }
   free(pager);
   free(table);
 }

 MetaCommandResult do_meta_command(InputBuffer*
input_buffer, Table* table) {
   if (strcmp(input_buffer->buffer, ".exit") == 0) {
     close_input_buffer(input_buffer);
     db_close(table);
     exit(EXIT_SUCCESS);
   } else if (strcmp(input_buffer->buffer, ".btree") == 0) {
     printf("Tree:\n");
     print_tree(table->pager, 0, 0);
     return META_COMMAND_SUCCESS;
   } else if (strcmp(input_buffer->buffer, ".constants") == 0)
{
     printf("Constants:\n");
     print_constants();
     return META_COMMAND_SUCCESS;
   } else {
     return
META_COMMAND_UNRECOGNIZED_COMMAND;
   }
 }

 PrepareResult prepare_insert(InputBuffer* input_buffer,
Statement* statement) {
   statement->type = STATEMENT_INSERT;

   char* keyword = strtok(input_buffer->buffer, " ");
   char* id_string = strtok(NULL, " ");
   char* username = strtok(NULL, " ");
   char* email = strtok(NULL, " ");

   if (id_string == NULL || username == NULL || email ==
NULL) {
     return PREPARE_SYNTAX_ERROR;
   }

   int id = atoi(id_string);
   if (id < 0) {
     return PREPARE_NEGATIVE_ID;
   }
   if (strlen(username) > COLUMN_USERNAME_SIZE) {
     return PREPARE_STRING_TOO_LONG;
   }
   if (strlen(email) > COLUMN_EMAIL_SIZE) {
     return PREPARE_STRING_TOO_LONG;
   }

   statement->row_to_insert.id = id;
   strcpy(statement->row_to_insert.username, username);
   strcpy(statement->row_to_insert.email, email);

   return PREPARE_SUCCESS;
 }

 PrepareResult prepare_statement(InputBuffer*
input_buffer,
                                 Statement* statement) {
   if (strncmp(input_buffer->buffer, "insert", 6) == 0) {
     return prepare_insert(input_buffer, statement);
   }
   if (strcmp(input_buffer->buffer, "select") == 0) {
     statement->type = STATEMENT_SELECT;
     return PREPARE_SUCCESS;
```

```
    }

    return PREPARE_UNRECOGNIZED_STATEMENT;
  }

  /*
  Until we start recycling free pages, new pages will always
  go onto the end of the database file
  */
  uint32_t get_unused_page_num(Pager* pager) { return
pager->num_pages; }

  void create_new_root(Table* table, uint32_t
right_child_page_num) {
    /*
    Handle splitting the root.
    Old root copied to new page, becomes left child.
    Address of right child passed in.
    Re-initialize root page to contain the new root node.
    New root node points to two children.
    */

    void* root = get_page(table->pager, table-
>root_page_num);
    void* right_child = get_page(table->pager,
right_child_page_num);
    uint32_t left_child_page_num =
get_unused_page_num(table->pager);
    void* left_child = get_page(table->pager,
left_child_page_num);

    /* Left child has data copied from old root */
    memcpy(left_child, root, PAGE_SIZE);
    set_node_root(left_child, false);

    /* Root node is a new internal node with one key and two
children */
    initialize_internal_node(root);
    set_node_root(root, true);
    *internal_node_num_keys(root) = 1;
    *internal_node_child(root, 0) = left_child_page_num;
    uint32_t left_child_max_key =
get_node_max_key(left_child);
    *internal_node_key(root, 0) = left_child_max_key;
    *internal_node_right_child(root) = right_child_page_num;
    *node_parent(left_child) = table->root_page_num;
    *node_parent(right_child) = table->root_page_num;
  }

  void internal_node_insert(Table* table, uint32_t
parent_page_num,
                          uint32_t child_page_num) {
    /*
    Add a new child/key pair to parent that corresponds to
child
    */

    void* parent = get_page(table->pager, parent_page_num);
    void* child = get_page(table->pager, child_page_num);
    uint32_t child_max_key = get_node_max_key(child);
    uint32_t index = internal_node_find_child(parent,
child_max_key);

    uint32_t original_num_keys =
*internal_node_num_keys(parent);
    *internal_node_num_keys(parent) = original_num_keys +
1;

    if (original_num_keys >=
INTERNAL_NODE_MAX_CELLS) {
      printf("Need to implement splitting internal node\n");
      exit(EXIT_FAILURE);
    }

    uint32_t right_child_page_num =
*internal_node_right_child(parent);
    void* right_child = get_page(table->pager,
right_child_page_num);

    if (child_max_key > get_node_max_key(right_child)) {
      /* Replace right child */
      *internal_node_child(parent, original_num_keys) =
right_child_page_num;
      *internal_node_key(parent, original_num_keys) =
        get_node_max_key(right_child);
      *internal_node_right_child(parent) = child_page_num;
    } else {
      /* Make room for the new cell */
      for (uint32_t i = original_num_keys; i > index; i--) {
        void* destination = internal_node_cell(parent, i);
        void* source = internal_node_cell(parent, i - 1);
        memcpy(destination, source,
INTERNAL_NODE_CELL_SIZE);
      }
      *internal_node_child(parent, index) = child_page_num;
      *internal_node_key(parent, index) = child_max_key;
    }
  }

  void update_internal_node_key(void* node, uint32_t
old_key, uint32_t new_key) {
    uint32_t old_child_index =
internal_node_find_child(node, old_key);
    *internal_node_key(node, old_child_index) = new_key;
  }

  void leaf_node_split_and_insert(Cursor* cursor, uint32_t
key, Row* value) {
    /*
    Create a new node and move half the cells over.
    Insert the new value in one of the two nodes.
    Update parent or create a new parent.
    */

    void* old_node = get_page(cursor->table->pager, cursor->page_num);
    uint32_t old_max = get_node_max_key(old_node);
    uint32_t new_page_num = get_unused_page_num(cursor->table->pager);
    void* new_node = get_page(cursor->table->pager, new_page_num);
    initialize_leaf_node(new_node);
    *node_parent(new_node) = *node_parent(old_node);
    *leaf_node_next_leaf(new_node) = *leaf_node_next_leaf(old_node);
    *leaf_node_next_leaf(old_node) = new_page_num;

    /*
    All existing keys plus new key should should be divided
    evenly between old (left) and new (right) nodes.
    Starting from the right, move each key to correct position.
    */
    for (int32_t i = LEAF_NODE_MAX_CELLS; i >= 0; i--) {
      void* destination_node;
      if (i >= LEAF_NODE_LEFT_SPLIT_COUNT) {
        destination_node = new_node;
      } else {
        destination_node = old_node;
      }
      uint32_t index_within_node = i % LEAF_NODE_LEFT_SPLIT_COUNT;
      void* destination = leaf_node_cell(destination_node, index_within_node);

      if (i == cursor->cell_num) {
        serialize_row(value,
                      leaf_node_value(destination_node, index_within_node));
        *leaf_node_key(destination_node, index_within_node) = key;
      } else if (i > cursor->cell_num) {
        memcpy(destination, leaf_node_cell(old_node, i - 1), LEAF_NODE_CELL_SIZE);
      } else {
        memcpy(destination, leaf_node_cell(old_node, i), LEAF_NODE_CELL_SIZE);
      }
    }

    /* Update cell count on both leaf nodes */
    *(leaf_node_num_cells(old_node)) = LEAF_NODE_LEFT_SPLIT_COUNT;
    *(leaf_node_num_cells(new_node)) = LEAF_NODE_RIGHT_SPLIT_COUNT;

    if (is_node_root(old_node)) {
      return create_new_root(cursor->table, new_page_num);
    } else {
      uint32_t parent_page_num = *node_parent(old_node);
      uint32_t new_max = get_node_max_key(old_node);
      void* parent = get_page(cursor->table->pager, parent_page_num);

      update_internal_node_key(parent, old_max, new_max);
      internal_node_insert(cursor->table, parent_page_num, new_page_num);
      return;
    }
  }

  void leaf_node_insert(Cursor* cursor, uint32_t key, Row* value) {
    void* node = get_page(cursor->table->pager, cursor->page_num);

    uint32_t num_cells = *leaf_node_num_cells(node);
    if (num_cells >= LEAF_NODE_MAX_CELLS) {
      // Node full
      leaf_node_split_and_insert(cursor, key, value);
      return;
    }

    if (cursor->cell_num < num_cells) {
      // Make room for new cell
      for (uint32_t i = num_cells; i > cursor->cell_num; i--) {
        memcpy(leaf_node_cell(node, i), leaf_node_cell(node, i - 1),
               LEAF_NODE_CELL_SIZE);
      }
    }

    *(leaf_node_num_cells(node)) += 1;
    *(leaf_node_key(node, cursor->cell_num)) = key;
    serialize_row(value, leaf_node_value(node, cursor->cell_num));
  }

  ExecuteResult execute_insert(Statement* statement, Table* table) {
    Row* row_to_insert = &(statement->row_to_insert);
    uint32_t key_to_insert = row_to_insert->id;
    Cursor* cursor = table_find(table, key_to_insert);

    void* node = get_page(table->pager, cursor->page_num);
    uint32_t num_cells = *leaf_node_num_cells(node);
    if (cursor->cell_num < num_cells) {
      uint32_t key_at_index = *leaf_node_key(node, cursor->cell_num);
      if (key_at_index == key_to_insert) {
        return EXECUTE_DUPLICATE_KEY;
      }
    }

    leaf_node_insert(cursor, row_to_insert->id, row_to_insert);

    free(cursor);

    return EXECUTE_SUCCESS;
  }

  ExecuteResult execute_select(Statement* statement, Table* table) {
    Cursor* cursor = table_start(table);

    Row row;
    while (!(cursor->end_of_table)) {
      deserialize_row(cursor_value(cursor), &row);
      print_row(&row);
      cursor_advance(cursor);
    }

    free(cursor);

    return EXECUTE_SUCCESS;
  }

  ExecuteResult execute_statement(Statement* statement, Table* table) {
    switch (statement->type) {
      case (STATEMENT_INSERT):
        return execute_insert(statement, table);
      case (STATEMENT_SELECT):
        return execute_select(statement, table);
    }
  }

  int main(int argc, char* argv[]) {
    if (argc < 2) {
      printf("Must supply a database filename.\n");
      exit(EXIT_FAILURE);
    }

    char* filename = argv[1];
    Table* table = db_open(filename);

    InputBuffer* input_buffer = new_input_buffer();
    while (true) {
      print_prompt();
      read_input(input_buffer);

      if (input_buffer->buffer[0] == '.') {
        switch (do_meta_command(input_buffer, table)) {
          case (META_COMMAND_SUCCESS):
            continue;
          case (META_COMMAND_UNRECOGNIZED_COMMAND):
            printf("Unrecognized command '%s'\n", input_buffer->buffer);
            continue;
        }
      }

      Statement statement;
      switch (prepare_statement(input_buffer, &statement)) {
        case (PREPARE_SUCCESS):
          break;
        case (PREPARE_NEGATIVE_ID):
          printf("ID must be positive.\n");
          continue;
        case (PREPARE_STRING_TOO_LONG):
          printf("String is too long.\n");
          continue;
        case (PREPARE_SYNTAX_ERROR):
          printf("Syntax error. Could not parse statement.\n");
          continue;
        case (PREPARE_UNRECOGNIZED_STATEMENT):
          printf("Unrecognized keyword at start of '%s'.\n",
                 input_buffer->buffer);
          continue;
      }

      switch (execute_statement(&statement, table)) {
        case (EXECUTE_SUCCESS):
          printf("Executed.\n");
          break;
        case (EXECUTE_DUPLICATE_KEY):
          printf("Error: Duplicate key.\n");
          break;
      }
    }
  }
```

5. Result

First, we compile the program using gcc and generate an executable. In our case, the program is “db.c”, and the name of the executable is also “db”. The compilation process is done on a Windows machine, so the file generated is “db.exe”

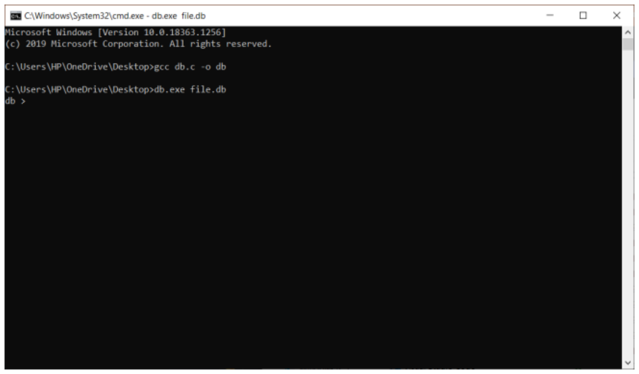
International Journal of Research in Engineering, Science and Management
Volume-3, Issue-12, December-2020
https://www.ijresm.com | ISSN (Online): 2581-5792



```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.18363.1256]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\HP\OneDrive\Desktop>gcc db.c -o db

C:\Users\HP\OneDrive\Desktop>
```

Fig. 1. Compiling the program

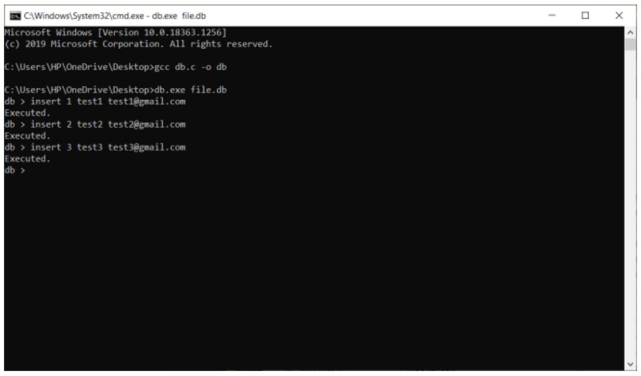
Next, we run the program, providing the filename we wish to write on. In our case, it is “file.db”.



```
C:\Windows\System32\cmd.exe - db.exe file.db
Microsoft Windows [Version 10.0.18363.1256]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\HP\OneDrive\Desktop>gcc db.c -o db

C:\Users\HP\OneDrive\Desktop>db.exe file.db
db >
```

Fig. 2. Running the program

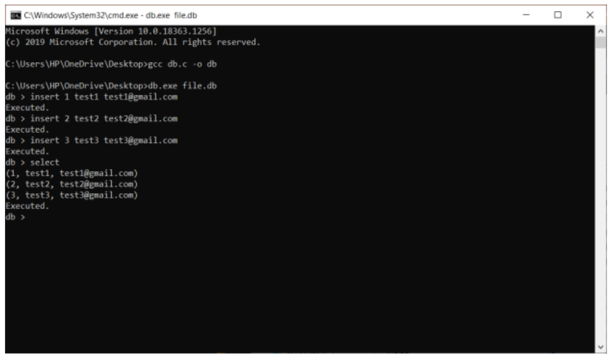
Now, we begin by inserting three rows.



```
C:\Windows\System32\cmd.exe - db.exe file.db
Microsoft Windows [Version 10.0.18363.1256]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\HP\OneDrive\Desktop>gcc db.c -o db

C:\Users\HP\OneDrive\Desktop>db.exe file.db
db > insert 1 test1 test1@gmail.com
Executed.
db > insert 2 test2 test2@gmail.com
Executed.
db > insert 3 test3 test3@gmail.com
Executed.
db >
```

Fig. 3. Performing the insertion operation

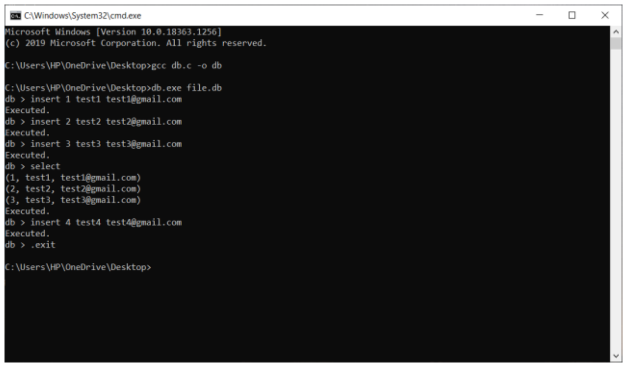
Now, we type “select”, and this will fetch all the data from our table and display it.



```
C:\Windows\System32\cmd.exe - db.exe file.db
Microsoft Windows [Version 10.0.18363.1256]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\HP\OneDrive\Desktop>gcc db.c -o db

C:\Users\HP\OneDrive\Desktop>db.exe file.db
db > insert 1 test1 test1@gmail.com
Executed.
db > insert 2 test2 test2@gmail.com
Executed.
db > insert 3 test3 test3@gmail.com
Executed.
db > select
(1, test1, test1@gmail.com)
(2, test2, test2@gmail.com)
(3, test3, test3@gmail.com)
Executed.
db >
```

Fig. 4. Retrieving data

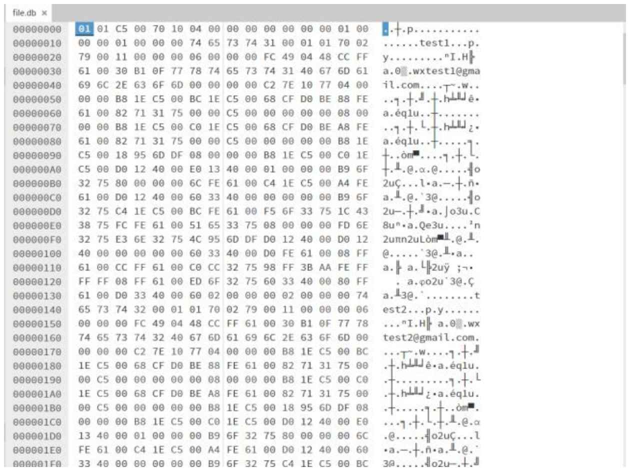
Finally, we insert one more row and type the exit meta command and close the shell.



```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.18363.1256]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\HP\OneDrive\Desktop>gcc db.c -o db

C:\Users\HP\OneDrive\Desktop>db.exe file.db
db > insert 1 test1 test1@gmail.com
Executed.
db > insert 2 test2 test2@gmail.com
Executed.
db > insert 3 test3 test3@gmail.com
Executed.
db > select
(1, test1, test1@gmail.com)
(2, test2, test2@gmail.com)
(3, test3, test3@gmail.com)
Executed.
db > insert 4 test4 test4@gmail.com
Executed.
db > .exit

C:\Users\HP\OneDrive\Desktop>
```

Fig. 5. Executing the exit meta command

The file is now opened using a hex editor and given a closer look.



```
file.db ×
00000000  01 01 C5 00 70 10 04 00 00 00 00 00 00 00 01 00  ..┼.p...........
00000010  00 00 01 00 00 00 74 65 73 74 31 00 01 01 70 02  ......test1...p.
00000020  79 00 11 00 00 00 06 00 00 00 FC 49 04 48 CC FF  y.........ⁿI.H╠
00000030  61 00 30 B1 0F 77 78 74 65 73 74 31 40 67 6D 61  a.0▒.wxtest1@gma
00000040  69 6C 2E 63 6F 6D 00 00 00 00 C2 7E 10 77 04 00  il.com....┬~.w..
00000050  00 00 B8 1E C5 00 BC 1E C5 00 68 CF D0 BE 88 FE  ..╕.┼.╝.┼.h╧╨╛ê■
00000060  61 00 82 71 31 75 00 00 C5 00 00 00 00 00 08 00  a.éq1u..┼.......
00000070  00 00 B8 1E C5 00 C0 1E C5 00 68 CF D0 BE A8 FE  ..╕.┼.└.┼.h╧╨╛¿■
00000080  61 00 82 71 31 75 00 00 C5 00 00 00 00 00 B8 1E  a.éq1u..┼.....╕.
00000090  C5 00 18 95 6D DF 08 00 00 00 B8 1E C5 00 C0 1E  ┼..òm▀....╕.┼.└.
000000A0  C5 00 D0 12 40 00 E0 13 40 00 01 00 00 00 B9 6F  ┼.╨.@.α.@.....╣o
000000B0  32 75 80 00 00 00 6C FE 61 00 C4 1E C5 00 A4 FE  2uÇ...l■a.─.┼.ñ■
000000C0  61 00 D0 12 40 00 60 33 40 00 00 00 00 00 B9 6F  a.╨.@.`3@.....╣o
000000D0  32 75 C4 1E C5 00 BC FE 61 00 F5 6F 33 75 1C 43  2u─.┼.╝■a.⌡o3u.C
000000E0  38 75 FC FE 61 00 51 65 33 75 08 00 00 00 FD 6E  8uⁿ■a.Qe3u....²n
000000F0  32 75 E3 6E 32 75 4C 95 6D DF D0 12 40 00 D0 12  2uπn2uLòm▀╨.@.╨.
00000100  40 00 00 00 00 00 60 33 40 00 D0 FE 61 00 08 FF  @.....`3@.╨■a..
00000110  61 00 CC FF 61 00 C0 CC 32 75 98 FF 3B AA FE FF  a.╠ a.└╠2uÿ ;¬■
00000120  FF FF 08 FF 61 00 ED 6F 32 75 60 33 40 00 80 FF     . a.φo2u`3@.Ç
00000130  61 00 D0 33 40 00 60 02 00 00 00 02 00 00 00 74  a.╨3@.`........t
00000140  65 73 74 32 00 01 01 70 02 79 00 11 00 00 00 06  est2...p.y......
00000150  00 00 00 FC 49 04 48 CC FF 61 00 30 B1 0F 77 78  ...ⁿI.H╠ a.0▒.wx
00000160  74 65 73 74 32 40 67 6D 61 69 6C 2E 63 6F 6D 00  test2@gmail.com.
00000170  00 00 00 C2 7E 10 77 04 00 00 00 B8 1E C5 00 BC  ...┬~.w....╕.┼.╝
00000180  1E C5 00 68 CF D0 BE 88 FE 61 00 82 71 31 75 00  .┼.h╧╨╛ê■a.éq1u.
00000190  00 C5 00 00 00 00 00 08 00 00 00 B8 1E C5 00 C0  .┼.........╕.┼.└
000001A0  1E C5 00 68 CF D0 BE A8 FE 61 00 82 71 31 75 00  .┼.h╧╨╛¿■a.éq1u.
000001B0  00 C5 00 00 00 00 00 B8 1E C5 00 18 95 6D DF 08  .┼.....╕.┼..òm▀.
000001C0  00 00 00 B8 1E C5 00 C0 1E C5 00 D0 12 40 00 E0  ...╕.┼.└.┼.╨.@.α
000001D0  13 40 00 01 00 00 00 B9 6F 32 75 80 00 00 00 6C  .@.....╣o2uÇ...l
000001E0  FE 61 00 C4 1E C5 00 A4 FE 61 00 D0 12 40 00 60  ■a.─.┼.ñ■a.╨.@.`
000001F0  33 40 00 00 00 00 00 B9 6F 32 75 C4 1E C5 00 BC  3@.....╣o2u─.┼.╝
```

Fig. 6. Opening ‘file.db’ using a hex editor

6. Conclusion

We have successfully implemented a simple database which follows the SQLite architecture using C programming language. Data can conveniently be written into a database file by inserting rows, and can be viewed by executing the select statement, or by using a hex editor to open the file and then parse its contents. The ability to write the data into a file that can be saved into the disk ensures information is not lost, and using a B-Tree to store and insert new rows supports the handling of large amount of data.

7. Future Scope

Currently, our program can only support the insert and the select statement. We would also like to implement the ability to delete the row which is pointed by the cursor, and to modify the row pointed by the cursor.

References

[1] https://www.sqlite.org/arch.html.
[2] https://dzone.com/articles/how-sqlite-database-works
[3] https://datacarpentry.org/sql-socialsci/08-sqlite-command-line/index.
[4] https://cstack.github.io/db_tutorial/