Memory_Single_Table_Database

July 16, 2024

1 Implementing `Insert` Statement...

We're going to start small by putting a lot of limitations on our database. For now, it will:

support two operations: inserting a row and printing all rows reside only in memory (no persistence to disk) support a single, hard-coded table Our hard-coded table is going to store users and look like this:

1. column : type
2. id: integer
3. username: varchar(32)
4. email: varchar(255) This is a simple schema, but it gets us to support multiple data types and multiple sizes of text data types.

insert statements are now going to look like this:

insert 1 cstack foo@bar.com

[ ]:
```c
if (strncmp(input_buffer->buffer, "insert", 6) == 0) {
     statement->type = STATEMENT_INSERT;
    int args_assigned = sscanf(
       input_buffer->buffer, "insert %d %s %s", &(statement->row_to_insert.id),
        statement->row_to_insert.username, statement->row_to_insert.email);
    if (args_assigned < 3) {
      return PREPARE_SYNTAX_ERROR;
    }
     return PREPARE_SUCCESS;
   }
   if (strcmp(input_buffer->buffer, "select") == 0) {
     statement->type = STATEMENT_SELECT;
     return PREPARE_SUCCESS;
   }
```

We store those parsed arguments into a new Row data structure inside the statement object:

[ ]:
```c
#define COLUMN_USERNAME_SIZE 32
#define COLUMN_EMAIL_SIZE 255
typedef struct {
  uint32_t id;
  char username[COLUMN_USERNAME_SIZE];
```

```
  char email[COLUMN_EMAIL_SIZE];
} Row;

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

Now we need to copy that data into some data structure representing the table. SQLite uses a B-tree for fast lookups, inserts and deletes. We'll start with something simpler. Like a B-tree, it will group rows into pages, but instead of arranging those pages as a tree it will arrange them as an array.

Here's my plan:

1. Store rows in blocks of memory called pages
2. Each page stores as many rows as it can fit
3. Rows are serialized into a compact representation with each page
4. Pages are only allocated as needed
5. Keep a fixed-size array of pointers to pages

First we'll define the compact representation of a row:

[ ]:
```
#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;
```

This code defines and calculates the sizes and offsets of attributes in a `Row` struct. Let's break it down step by step:

1. The `size_of_attribute` macro is defined. This macro takes two arguments: `Struct` and `Attribute`. It calculates the size of the `Attribute` within the `Struct` using the `sizeof` operator. The `sizeof` operator returns the size in bytes of its operand.
2. The `ID_SIZE` constant is defined using the `size_of_attribute` macro. It calculates the size of the `id` attribute within the `Row` struct.
3. The `USERNAME_SIZE` constant is defined using the `size_of_attribute` macro. It calculates the size of the `username` attribute within the `Row` struct.
4. The `EMAIL_SIZE` constant is defined using the `size_of_attribute` macro. It calculates the size of the `email` attribute within the `Row` struct.
5. The `ID_OFFSET` constant is defined as 0. This represents the starting offset of the `id` attribute within a `Row` struct.

6. The `USERNAME_OFFSET` constant is defined as the sum of `ID_OFFSET` and `ID_SIZE`. This represents the starting offset of the `username` attribute within a `Row` struct.
7. The `EMAIL_OFFSET` constant is defined as the sum of `USERNAME_OFFSET` and `USERNAME_SIZE`. This represents the starting offset of the `email` attribute within a `Row` struct.
8. The `ROW_SIZE` constant is defined as the sum of `ID_SIZE`, `USERNAME_SIZE`, and `EMAIL_SIZE`. This represents the total size of a `Row` struct, taking into account the sizes of all its attributes.

Overall, this code is used to calculate the sizes and offsets of attributes within a `Row` struct. These values can be useful for various purposes, such as memory allocation or serialization/deserialization operations.

We also need code to convert to and from the compact representation.

[ ]:
```
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);
}
```

The provided code selection consists of two functions: `serialize_row` and `deserialize_row`. These functions are used to convert a `Row` object into a byte array (serialization) and vice versa (deserialization). This can be useful when storing or transmitting data in a format that can be easily written to disk or sent over a network.

The `serialize_row` function takes a `Row` object called `source` and a pointer to the destination byte array. It uses the `memcpy` function to copy the individual fields of the `Row` object into the appropriate positions in the byte array. The `memcpy` function takes the source address, the destination address, and the size of the data to be copied. In this case, the `memcpy` calls are used to copy the `id`, `username`, and `email` fields of the `Row` object into the byte array.

The `deserialize_row` function does the opposite of `serialize_row`. It takes a pointer to the source byte array and a `Row` object called `destination`. It uses `memcpy` to copy the data from the byte array into the corresponding fields of the `destination` `Row` object. The `memcpy` calls in this function are reversed compared to `serialize_row`, with the source and destination addresses swapped.

These functions assume that the byte array has enough space to hold the serialized data and that the `Row` object has been properly initialized before calling `deserialize_row`. It's important to note that these functions only handle the data copying part of serialization and deserialization. Any additional logic, such as handling endianness or converting data types, would need to be implemented separately if required.

Overall, these functions provide a convenient way to convert `Row` objects to byte arrays and back, allowing for efficient storage and transmission of data.

Next, a Table structure that points to pages of rows and keeps track of how many rows there are:

[ ]:
```c
const uint32_t PAGE_SIZE = 4096;
#define TABLE_MAX_PAGES 100
const uint32_t ROWS_PER_PAGE = PAGE_SIZE / ROW_SIZE;
const uint32_t TABLE_MAX_ROWS = ROWS_PER_PAGE * TABLE_MAX_PAGES;

typedef struct {
  uint32_t num_rows;
  void* pages[TABLE_MAX_PAGES];
} Table;
```

The provided code selection is written in C and defines a data structure called `Table`. This structure represents a table in a memory-based single-table database. Let's break down the code and understand its components.

First, we have the definition of a constant variable `PAGE_SIZE` with a value of 4096. This constant represents the size of a page in the database. In memory, data is typically organized into pages, and each page has a fixed size. In this case, the page size is set to 4096 bytes.

Next, we have a preprocessor directive `#define` that defines another constant variable `TABLE_MAX_PAGES` with a value of 100. This constant represents the maximum number of pages that can be stored in the table. It is common to limit the number of pages to prevent excessive memory usage.

Following that, we have another constant variable `ROWS_PER_PAGE` which is calculated by dividing the `PAGE_SIZE` by the size of a row in the table. The size of a row is not explicitly defined in the provided code, but it is assumed to be defined elsewhere. This constant represents the maximum number of rows that can be stored in a single page.

Finally, we have the `Table` structure definition. It consists of two members: `num_rows` and `pages`. The `num_rows` member is an unsigned 32-bit integer that represents the total number of rows in the table. The `pages` member is an array of `void` pointers with a size of `TABLE_MAX_PAGES`. This array is used to store the pointers to the pages of the table. Each page is represented by a `void` pointer, allowing flexibility in storing different types of data.

Overall, this code selection defines the necessary constants and data structure to represent a table in a memory-based single-table database. It provides a foundation for managing and manipulating data within the table.

I'm making our page size 4 kilobytes because it's the same size as a page used in the virtual memory systems of most computer architectures. This means one page in our database corresponds to one page used by the operating system. The operating system will move pages in and out of memory as whole units instead of breaking them up.

I'm setting an arbitrary limit of 100 pages that we will allocate. When we switch to a tree structure, our database's maximum size will only be limited by the maximum size of a file. (Although we'll still limit how many pages we keep in memory at once)

Rows should not cross page boundaries. Since pages probably won't exist next to each other in memory, this assumption makes it easier to read/write rows.

Speaking of which, here is how we figure out where to read/write in memory for a particular row:

[ ]:
```
void* row_slot(Table* table, uint32_t row_num) {
  uint32_t page_num = row_num / ROWS_PER_PAGE;
  void* page = table->pages[page_num];
  if (page == NULL) {
    // Allocate memory only when we try to access page
    page = table->pages[page_num] = malloc(PAGE_SIZE);
  }
  uint32_t row_offset = row_num % ROWS_PER_PAGE;
  uint32_t byte_offset = row_offset * ROW_SIZE;
  return page + byte_offset;
}
```

The provided code snippet is a function called `row_slot` that is part of a larger program for a memory-based single-table database. This function is responsible for returning a pointer to the memory location where a specific row of data is stored.

Let's break down the code step by step to understand how it works.

The function takes two parameters: a pointer to a `Table` struct called `table` and an unsigned 32-bit integer called `row_num`, which represents the row number we want to access.

The first line of the function calculates the page number where the desired row is located. It does this by dividing the `row_num` by the constant `ROWS_PER_PAGE`. This constant represents the number of rows that can fit in a single page of memory.

Next, the function retrieves the pointer to the page where the row is stored. It does this by accessing the `pages` array within the `table` struct using the calculated `page_num` as the index. If the page pointer is `NULL`, it means that the page has not been allocated yet. In this case, the function allocates memory for the page using the `malloc` function and assigns the allocated memory to the corresponding index in the `pages` array.

After obtaining the page pointer, the function calculates the offset of the desired row within the page. It does this by taking the remainder of the division of `row_num` by `ROWS_PER_PAGE`. This gives us the row offset within the page.

The next line calculates the byte offset of the row within the page. It multiplies the row offset by the constant `ROW_SIZE`, which represents the size of each row in bytes.

Finally, the function returns a pointer to the memory location where the desired row is stored. It does this by adding the byte offset to the page pointer, effectively pointing to the correct memory location within the page.

In summary, the `row_slot` function calculates the page and byte offsets for a given row number and returns a pointer to the memory location where that row is stored in the database. This function ensures that the necessary memory is allocated for the page if it hasn't been accessed before.

Now we can make execute_statement read/write from our table structure:

[ ]:
```
void execute_statement(Statement* statement) {
ExecuteResult execute_insert(Statement* statement, Table* table) {
```

```
  if (table->num_rows >= TABLE_MAX_ROWS) {
    return EXECUTE_TABLE_FULL;
  }

  Row* row_to_insert = &(statement->row_to_insert);

  serialize_row(row_to_insert, row_slot(table, table->num_rows));
  table->num_rows += 1;

  return EXECUTE_SUCCESS;
}

ExecuteResult execute_select(Statement* statement, Table* table) {
  Row row;
  for (uint32_t i = 0; i < table->num_rows; i++) {
    deserialize_row(row_slot(table, i), &row);
    print_row(&row);
  }
  return EXECUTE_SUCCESS;
}

ExecuteResult execute_statement(Statement* statement, Table* table) {
   switch (statement->type) {
     case (STATEMENT_INSERT):
      printf("This is where we would do an insert.\n");
      break;
      return execute_insert(statement, table);
     case (STATEMENT_SELECT):
      printf("This is where we would do a select.\n");
      break;
      return execute_select(statement, table);
   }
}
```

Lastly, we need to initialize the table, create the respective memory release function and handle a few more error cases:

[ ]:
```
 Table* new_table() {
  Table* table = (Table*)malloc(sizeof(Table));
  table->num_rows = 0;
  for (uint32_t i = 0; i < TABLE_MAX_PAGES; i++) {
     table->pages[i] = NULL;
  }
  return table;
}

void free_table(Table* table) {
```

```
    for (int i = 0; table->pages[i]; i++) {
        free(table->pages[i]);
    }
    free(table);
}
```