Let's Build a Simple Database

Writing a sqlite clone from scratch in C

Overview

View on GitHub (pull requests welcome)

Part 3 - An In-Memory, Append-Only, Single-Table Database

< Part 2 - World's Simplest SQL Compiler and Virtual Machine

Part 4 - Our First Tests (and Bugs) >

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:

column	type
id	integer
username	varchar(32)
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
```

That means we need to upgrade our `prepare_statement` function to parse arguments

```
   if (strncmp(input_buffer->buffer, "insert", 6) == 0) {
     statement->type = STATEMENT_INSERT;
+    int args_assigned = sscanf(
+        input_buffer->buffer, "insert %d %s %s", &(statement->r
+        statement->row_to_insert.username, statement->row_to_ir
+    if (args_assigned < 3) {
+      return PREPARE_SYNTAX_ERROR;
+    }
     return PREPARE_SUCCESS;
   }
   if (strcmp(input_buffer->buffer, "select") == 0) {
```

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

```
+#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:

- Store rows in blocks of memory called pages

- Each page stores as many rows as it can fit
- Rows are serialized into a compact representation with each page
- Pages are only allocated as needed
- 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*)(
+
+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 means the layout of a serialized row will look like this:

column	size (bytes)	offset
id	4	0
username	32	4
email	255	36
total	291	

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

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

```
+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;
```

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;
+}
```

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* ta
   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);
+}
```

```
 int main(int argc, char* argv[]) {
+  Table* table = new_table();
   InputBuffer* input_buffer = new_input_buffer();
   while (true) {
     print_prompt();
@@ -105,13 +203,22 @@ int main(int argc, char* argv[]) {
     switch (prepare_statement(input_buffer, &statement)) {
       case (PREPARE_SUCCESS):
         break;
+      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;
     }
-    execute_statement(&statement);
-    printf("Executed.\n");
+    switch (execute_statement(&statement, table)) {
+      case (EXECUTE_SUCCESS):
+        printf("Executed.\n");
+        break;
+      case (EXECUTE_TABLE_FULL):
+        printf("Error: Table full.\n");
+        break;
+    }
   }
 }
```

With those changes we can actually save data in our database!

```
~ ./db
db > insert 1 cstack foo@bar.com
Executed.
db > insert 2 bob bob@example.com
Executed.
db > select
(1, cstack, foo@bar.com)
(2, bob, bob@example.com)
Executed.
db > insert foo bar 1
Syntax error. Could not parse statement.
db > .exit
~
```

Now would be a great time to write some tests, for a couple reasons:

- We're planning to dramatically change the data structure storing our table, and tests would catch regressions.
- There are a couple edge cases we haven't tested manually (e.g. filling up the table)

We'll address those issues in the next part. For now, here's the complete diff from this part:

```
@@ -2,6 +2,7 @@
 #include <stdio.h>
 #include <stdlib.h>
 #include <string.h>
+#include <stdint.h>

 typedef struct {
   char* buffer;
@@ -10,6 +11,105 @@ typedef struct {
 } InputBuffer;

+typedef enum { EXECUTE_SUCCESS, EXECUTE_TABLE_FULL } ExecuteRes
+
+typedef enum {
+  META_COMMAND_SUCCESS,
+  META_COMMAND_UNRECOGNIZED_COMMAND
+} MetaCommandResult;
+
+typedef enum {
+  PREPARE_SUCCESS,
+  PREPARE_SYNTAX_ERROR,
+  PREPARE_UNRECOGNIZED_STATEMENT
+ } PrepareResult;
+
+typedef enum { STATEMENT_INSERT, STATEMENT_SELECT } StatementTy
+
+#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;
+
+#define size_of_attribute(Struct, Attribute) sizeof(((Struct*)(
+
+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
+const uint32_t ROWS_PER_PAGE = PAGE_SIZE / ROW_SIZE;
+const uint32_t TABLE_MAX_ROWS = ROWS_PER_PAGE * TABLE_MAX_PAGE
+
+typedef struct {
+  uint32_t num_rows;
+  void* pages[TABLE_MAX_PAGES];
+} Table;
+
+void print_row(Row* row) {
+  printf("(%d, %s, %s)\n", row->id, row->username, row->email)
+}
+
+void serialize_row(Row* source, void* destination) {
+  memcpy(destination + ID_OFFSET, &(source->id), ID_SIZE);
+  memcpy(destination + USERNAME_OFFSET, &(source->username), U
+  memcpy(destination + EMAIL_OFFSET, &(source->email), EMAIL_S
+}
+
+void deserialize_row(void *source, Row* destination) {
+  memcpy(&(destination->id), source + ID_OFFSET, ID_SIZE);
+  memcpy(&(destination->username), source + USERNAME_OFFSET, U
+  memcpy(&(destination->email), source + EMAIL_OFFSET, EMAIL_S
+}
+
+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;
+}
+
+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);
+}
+
 InputBuffer* new_input_buffer() {
   InputBuffer* input_buffer = (InputBuffer*)malloc(sizeof(Input
   input_buffer->buffer = NULL;
@@ -40,17 +140,105 @@ void close_input_buffer(InputBuffer* input
     free(input_buffer);
 }

+MetaCommandResult do_meta_command(InputBuffer* input_buffer, Ta
+   if (strcmp(input_buffer->buffer, ".exit") == 0) {
+     close_input_buffer(input_buffer);
+     free_table(table);
+     exit(EXIT_SUCCESS);
+   } else {
+     return META_COMMAND_UNRECOGNIZED_COMMAND;
+   }
+}
```

```
+
+PrepareResult prepare_statement(InputBuffer* input_buffer,
+                                Statement* statement) {
+  if (strncmp(input_buffer->buffer, "insert", 6) == 0) {
+    statement->type = STATEMENT_INSERT;
+    int args_assigned = sscanf(
+       input_buffer->buffer, "insert %d %s %s", &(statement->ro
+       statement->row_to_insert.username, statement->row_to_ins
+       );
+    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;
+  }
+
+  return PREPARE_UNRECOGNIZED_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 *ta
+  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[]) {
+  Table* table = new_table();
   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->buff
+    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->b
+          continue;
+      }
+    }
+
+    Statement statement;
+    switch (prepare_statement(input_buffer, &statement)) {
+      case (PREPARE_SUCCESS):
+        break;
+      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_TABLE_FULL):
+           printf("Error: Table full.\n");
+           break;
      }
    }
  }
```

< Part 2 - World's Simplest SQL Compiler and Virtual Machine

Part 4 - Our First Tests (and Bugs) >