Let's Build a Simple Database

Writing a sqlite clone from scratch in C

Overview

View on GitHub (pull requests welcome)

Part 4 - Our First Tests (and Bugs)

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

Part 5 - Persistence to Disk >

We've got the ability to insert rows into our database and to print out all rows. Let's take a moment to test what we've got so far.

I'm going to use rspec to write my tests because I'm familiar with it, and the syntax is fairly readable.

I'll define a short helper to send a list of commands to our database program then make assertions about the output:

```
describe 'database' do
  def run_script(commands)
    raw_output = nil
    IO.popen("./db", "r+") do |pipe|
      commands.each do |command|
        pipe.puts command
      end

      pipe.close_write

      # Read entire output
      raw_output = pipe.gets(nil)
    end
    raw_output.split("\n")
```

```
  end

  it 'inserts and retrieves a row' do
    result = run_script([
      "insert 1 user1 person1@example.com",
      "select",
      ".exit",
    ])
    expect(result).to match_array([
      "db > Executed.",
      "db > (1, user1, person1@example.com)",
      "Executed.",
      "db > ",
    ])
  end
end
```

This simple test makes sure we get back what we put in. And indeed it passes:

```
bundle exec rspec
.

Finished in 0.00871 seconds (files took 0.09506 seconds to load)
1 example, 0 failures
```

Now it's feasible to test inserting a large number of rows into the database:

```
it 'prints error message when table is full' do
  script = (1..1401).map do |i|
    "insert #{i} user#{i} person#{i}@example.com"
  end
  script << ".exit"
  result = run_script(script)
  expect(result[-2]).to eq('db > Error: Table full.')
end
```

Running tests again…

```
bundle exec rspec
..

Finished in 0.01553 seconds (files took 0.08156 seconds to load)
2 examples, 0 failures
```

Sweet, it works! Our db can hold 1400 rows right now because we set the maximum number of pages to 100, and 14 rows can fit in a page.

Reading through the code we have so far, I realized we might not handle storing text fields correctly. Easy to test with this example:

```
it 'allows inserting strings that are the maximum length' do
  long_username = "a"*32
  long_email = "a"*255
  script = [
    "insert 1 #{long_username} #{long_email}",
    "select",
    ".exit",
  ]
  result = run_script(script)
  expect(result).to match_array([
    "db > Executed.",
    "db > (1, #{long_username}, #{long_email})",
    "Executed.",
    "db > ",
  ])
end
```

And the test fails!

```
Failures:

  1) database allows inserting strings that are the maximum leng
     Failure/Error: raw_output.split("\n")

     ArgumentError:
       invalid byte sequence in UTF-8
     # ./spec/main_spec.rb:14:in `split'
```

```
      # ./spec/main_spec.rb:14:in `run_script'
      # ./spec/main_spec.rb:48:in `block (2 levels) in <top (requ
```

If we try it ourselves, we'll see that there's some weird characters when we try to print out the row. (I'm abbreviating the long strings):

```
db > insert 1 aaaaa... aaaaa...
Executed.
db > select
(1, aaaaa...aaa\x, aaaaa...aaa\x)
Executed.
db >
```

What's going on? If you take a look at our definition of a Row, we allocate exactly 32 bytes for username and exactly 255 bytes for email. But C strings are supposed to end with a null character, which we didn't allocate space for. The solution is to allocate one additional byte:

```
 const uint32_t COLUMN_EMAIL_SIZE = 255;
 typedef struct {
   uint32_t id;
-  char username[COLUMN_USERNAME_SIZE];
-  char email[COLUMN_EMAIL_SIZE];
+  char username[COLUMN_USERNAME_SIZE + 1];
+  char email[COLUMN_EMAIL_SIZE + 1];
 } Row;
```

And indeed that fixes it:

```
 bundle exec rspec
...

Finished in 0.0188 seconds (files took 0.08516 seconds to load)
3 examples, 0 failures
```

We should not allow inserting usernames or emails that are longer than column size. The spec for that looks like this:

```
it 'prints error message if strings are too long' do
  long_username = "a"*33
  long_email = "a"*256
  script = [
    "insert 1 #{long_username} #{long_email}",
    "select",
    ".exit",
  ]
  result = run_script(script)
  expect(result).to match_array([
    "db > String is too long.",
    "db > Executed.",
    "db > ",
  ])
end
```

In order to do this we need to upgrade our parser. As a reminder, we're currently using scanf():

```
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_
      statement->row_to_insert.username, statement->row_to_inse
  if (args_assigned < 3) {
    return PREPARE_SYNTAX_ERROR;
  }
  return PREPARE_SUCCESS;
}
```

But scanf has some disadvantages. If the string it's reading is larger than the buffer it's reading into, it will cause a buffer overflow and start writing into unexpected places. We want to check the length of each string before we copy it into a `Row` structure. And to do that, we need to divide the input by spaces.

I'm going to use strtok() to do that. I think it's easiest to understand if you see it in action:

```
+PrepareResult prepare_insert(InputBuffer* input_buffer, Stateme
+  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 (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);
-    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;
```

```
    }
```

Calling `strtok` successively on the input buffer breaks it into substrings by inserting a null character whenever it reaches a delimiter (space, in our case). It returns a pointer to the start of the substring.

We can call strlen() on each text value to see if it's too long.

We can handle the error like we do any other error code:

```
 enum PrepareResult_t {
   PREPARE_SUCCESS,
+  PREPARE_STRING_TOO_LONG,
   PREPARE_SYNTAX_ERROR,
   PREPARE_UNRECOGNIZED_STATEMENT
 };
```

```
 switch (prepare_statement(input_buffer, &statement)) {
   case (PREPARE_SUCCESS):
     break;
+  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;
```

Which makes our test pass

```
bundle exec rspec
....

Finished in 0.02284 seconds (files took 0.116 seconds to load)
4 examples, 0 failures
```

While we're here, we might as well handle one more error case:

```
it 'prints an error message if id is negative' do
```

```
  script = [
    "insert -1 cstack foo@bar.com",
    "select",
    ".exit",
  ]
  result = run_script(script)
  expect(result).to match_array([
    "db > ID must be positive.",
    "db > Executed.",
    "db > ",
  ])
end
```

```
 enum PrepareResult_t {
   PREPARE_SUCCESS,
+  PREPARE_NEGATIVE_ID,
   PREPARE_STRING_TOO_LONG,
   PREPARE_SYNTAX_ERROR,
   PREPARE_UNRECOGNIZED_STATEMENT
@@ -148,9 +147,6 @@ PrepareResult prepare_insert(InputBuffer* ir
   }

   int id = atoi(id_string);
+  if (id < 0) {
+    return PREPARE_NEGATIVE_ID;
+  }
   if (strlen(username) > COLUMN_USERNAME_SIZE) {
     return PREPARE_STRING_TOO_LONG;
   }
@@ -230,9 +226,6 @@ int main(int argc, char* argv[]) {
     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;
```

Alright, that's enough testing for now. Next is a very important feature: persistence! We're going to save our database to a file and read it back out again.

It's gonna be great.

Here's the complete diff for this part:

```
@@ -22,6 +22,8 @@

 enum PrepareResult_t {
   PREPARE_SUCCESS,
+  PREPARE_NEGATIVE_ID,
+  PREPARE_STRING_TOO_LONG,
   PREPARE_SYNTAX_ERROR,
   PREPARE_UNRECOGNIZED_STATEMENT
  };
@@ -34,8 +36,8 @@
 #define COLUMN_EMAIL_SIZE 255
 typedef struct {
   uint32_t id;
-  char username[COLUMN_USERNAME_SIZE];
-  char email[COLUMN_EMAIL_SIZE];
+  char username[COLUMN_USERNAME_SIZE + 1];
+  char email[COLUMN_EMAIL_SIZE + 1];
 } Row;

@@ -150,18 +152,40 @@ MetaCommandResult do_meta_command(InputBu
   }
 }

-PrepareResult prepare_statement(InputBuffer* input_buffer,
-                                Statement* statement) {
-  if (strncmp(input_buffer->buffer, "insert", 6) == 0) {
+PrepareResult prepare_insert(InputBuffer* input_buffer, Stateme
   statement->type = STATEMENT_INSERT;
-  int args_assigned = sscanf(
-      input_buffer->buffer, "insert %d %s %s", &(statement->row_
-      statement->row_to_insert.username, statement->row_to_inser
-      );
-  if (args_assigned < 3) {
```

```
+
+  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;
@@ -223,6 +247,12 @@ int main(int argc, char* argv[]) {
     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;
```

And we added tests:

```
+describe 'database' do
+  def run_script(commands)
+    raw_output = nil
+    IO.popen("./db", "r+") do |pipe|
+      commands.each do |command|
+        pipe.puts command
+      end
+
+      pipe.close_write
+
+      # Read entire output
+      raw_output = pipe.gets(nil)
+    end
+    raw_output.split("\n")
+  end
+
+  it 'inserts and retrieves a row' do
+    result = run_script([
+      "insert 1 user1 person1@example.com",
+      "select",
+      ".exit",
+    ])
+    expect(result).to match_array([
+      "db > Executed.",
+      "db > (1, user1, person1@example.com)",
+      "Executed.",
+      "db > ",
+    ])
```

```
+  end
+
+  it 'prints error message when table is full' do
+    script = (1..1401).map do |i|
+      "insert #{i} user#{i} person#{i}@example.com"
+    end
+    script << ".exit"
+    result = run_script(script)
+    expect(result[-2]).to eq('db > Error: Table full.')
+  end
+
+  it 'allows inserting strings that are the maximum length' do
+    long_username = "a"*32
+    long_email = "a"*255
+    script = [
+      "insert 1 #{long_username} #{long_email}",
+      "select",
+      ".exit",
+    ]
+    result = run_script(script)
+    expect(result).to match_array([
+      "db > Executed.",
+      "db > (1, #{long_username}, #{long_email})",
+      "Executed.",
+      "db > ",
+    ])
+  end
+
+  it 'prints error message if strings are too long' do
+    long_username = "a"*33
+    long_email = "a"*256
+    script = [
+      "insert 1 #{long_username} #{long_email}",
+      "select",
+      ".exit",
+    ]
+    result = run_script(script)
+    expect(result).to match_array([
+      "db > String is too long.",
+      "db > Executed.",
```

```
+      "db > ",
+    ])
+  end
+
+  it 'prints an error message if id is negative' do
+    script = [
+      "insert -1 cstack foo@bar.com",
+      "select",
+      ".exit",
+    ]
+    result = run_script(script)
+    expect(result).to match_array([
+      "db > ID must be positive.",
+      "db > Executed.",
+      "db > ",
+    ])
+  end
+end
```

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

Part 5 - Persistence to Disk >
