Let's Build a Simple Database

Writing a sqlite clone from scratch in C

Overview

View on GitHub (pull requests welcome)

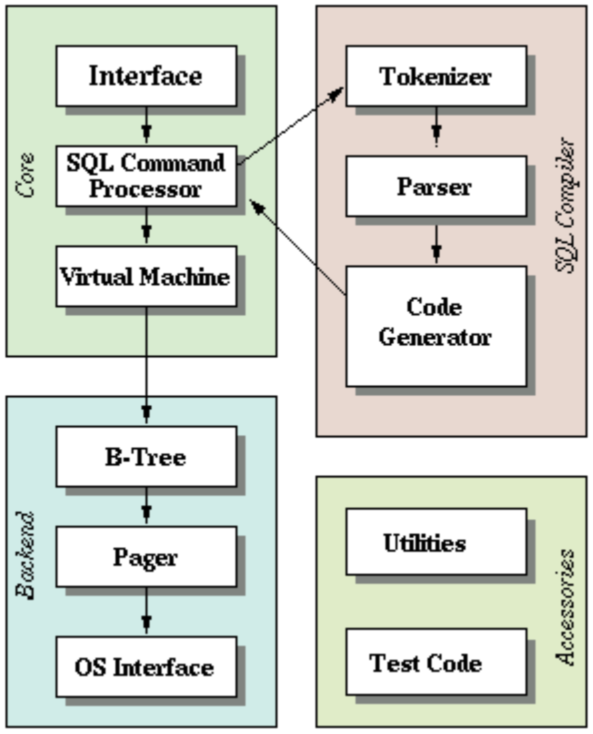
Part 2 - World's Simplest SQL Compiler and Virtual Machine

< Part 1 - Introduction and Setting up the REPL

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

We’re making a clone of sqlite. The “front-end” of sqlite is a SQL compiler that parses a string and outputs an internal representation called bytecode.

This bytecode is passed to the virtual machine, which executes it.



SQLite Architecture (https://www.sqlite.org/arch.html)

Breaking things into two steps like this has a couple advantages:

- Reduces the complexity of each part (e.g. virtual machine does not worry about syntax errors)
- Allows compiling common queries once and caching the bytecode for improved performance

With this in mind, let's refactor our `main` function and support two new keywords in the process:

```
 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) {
-      exit(EXIT_SUCCESS);
-    } else {
-      printf("Unrecognized command '%s'.\n", input_buffer->buff
+    if (input_buffer->buffer[0] == '.') {
+      switch (do_meta_command(input_buffer)) {
+        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_UNRECOGNIZED_STATEMENT):
+        printf("Unrecognized keyword at start of '%s'.\n",
+               input_buffer->buffer);
+        continue;
+    }
+
+     execute_statement(&statement);
+     printf("Executed.\n");
    }
  }
```

Non-SQL statements like `.exit` are called "meta-commands". They all start with a dot, so we check for them and handle them in a separate function.

Next, we add a step that converts the line of input into our internal representation of a statement. This is our hacky version of the sqlite front-end.

Lastly, we pass the prepared statement to `execute_statement`. This function will eventually become our virtual machine.

Notice that two of our new functions return enums indicating success or failure:

```
typedef enum {
  META_COMMAND_SUCCESS,
  META_COMMAND_UNRECOGNIZED_COMMAND
} MetaCommandResult;

typedef enum { PREPARE_SUCCESS, PREPARE_UNRECOGNIZED_STATEMENT
```

"Unrecognized statement"? That seems a bit like an exception. I prefer not to use exceptions (and C doesn't even support them), so I'm using enum result codes wherever practical. The C compiler will complain if my switch statement doesn't handle a member of the enum, so we can feel a little more confident we handle every result of a function. Expect more result codes to be added in the future.

`do_meta_command` is just a wrapper for existing functionality that leaves room for more commands:

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

Our "prepared statement" right now just contains an enum with two possible values. It will contain more data as we allow parameters in statements:

```
typedef enum { STATEMENT_INSERT, STATEMENT_SELECT } StatementTy

typedef struct {
  StatementType type;
} Statement;
```

`prepare_statement` (our "SQL Compiler") does not understand SQL right now. In fact, it only understands two words:

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

  return PREPARE_UNRECOGNIZED_STATEMENT;
}
```

Note that we use `strncmp` for "insert" since the "insert" keyword will be followed by data. (e.g. `insert 1 cstack foo@bar.com`)

Lastly, `execute_statement` contains a few stubs:

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

Note that it doesn't return any error codes because there's nothing that could go wrong yet.

With these refactors, we now recognize two new keywords!

```
~ ./db
db > insert foo bar
This is where we would do an insert.
Executed.
db > delete foo
Unrecognized keyword at start of 'delete foo'.
db > select
This is where we would do a select.
Executed.
db > .tables
Unrecognized command '.tables'
db > .exit
~
```

The skeleton of our database is taking shape… wouldn't it be nice if it stored data? In the next part, we'll implement `insert` and `select`, creating the world's worst data store. In the mean time, here's the entire diff from this part:

```
@@ -10,6 +10,23 @@ struct InputBuffer_t {
 } InputBuffer;

+typedef enum {
+  META_COMMAND_SUCCESS,
+  META_COMMAND_UNRECOGNIZED_COMMAND
+} MetaCommandResult;
+
+typedef enum { PREPARE_SUCCESS, PREPARE_UNRECOGNIZED_STATEMENT
+
+typedef enum { STATEMENT_INSERT, STATEMENT_SELECT } StatementTy
+
+typedef struct {
+  StatementType type;
+} Statement;
+
 InputBuffer* new_input_buffer() {
   InputBuffer* input_buffer = malloc(sizeof(InputBuffer));
   input_buffer->buffer = NULL;
@@ -40,17 +57,67 @@ void close_input_buffer(InputBuffer* input_b
     free(input_buffer);
 }

+MetaCommandResult do_meta_command(InputBuffer* input_buffer) {
+  if (strcmp(input_buffer->buffer, ".exit") == 0) {
+    close_input_buffer(input_buffer);
+    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;
+    return PREPARE_SUCCESS;
+  }
+  if (strcmp(input_buffer->buffer, "select") == 0) {
+    statement->type = STATEMENT_SELECT;
+    return PREPARE_SUCCESS;
+  }
+
+  return PREPARE_UNRECOGNIZED_STATEMENT;
+}
+
+void execute_statement(Statement* statement) {
+  switch (statement->type) {
+    case (STATEMENT_INSERT):
+      printf("This is where we would do an insert.\n");
+      break;
+    case (STATEMENT_SELECT):
```

```
+      printf("This is where we would do a select.\n");
+      break;
+  }
+}
+
 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->buff
+    if (input_buffer->buffer[0] == '.') {
+      switch (do_meta_command(input_buffer)) {
+        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_UNRECOGNIZED_STATEMENT):
+        printf("Unrecognized keyword at start of '%s'.\n",
+               input_buffer->buffer);
+        continue;
+    }
+
+    execute_statement(&statement);
+    printf("Executed.\n");
   }
 }
```

< Part 1 - Introduction and Setting up the REPL

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

rss | subscribe by email

This project is maintained by cstack