Let's Build a Simple Database

Writing a sqlite clone from scratch in C

Overview

View on GitHub (pull requests welcome)

Part 1 - Introduction and Setting up the REPL

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

As a web developer, I use relational databases every day at my job, but they're a black box to me. Some questions I have:

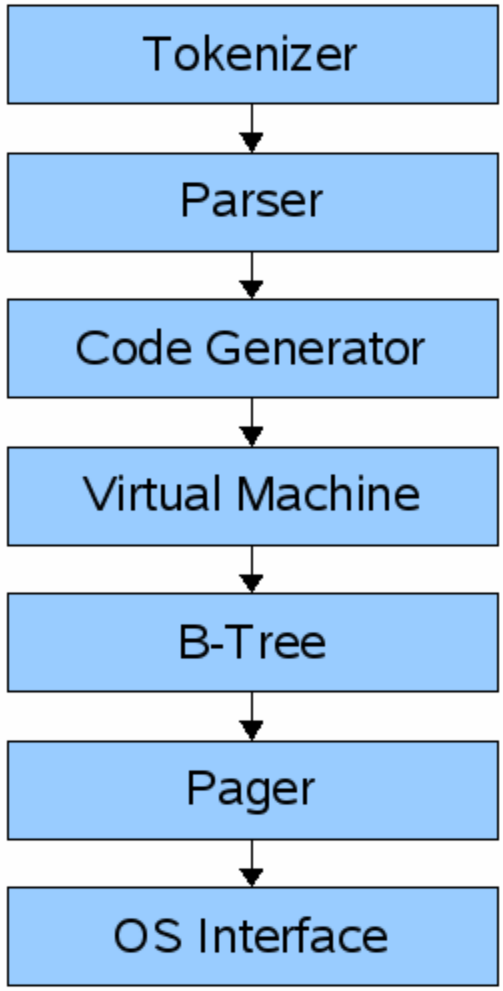
- What format is data saved in? (in memory and on disk)
- When does it move from memory to disk?
- Why can there only be one primary key per table?
- How does rolling back a transaction work?
- How are indexes formatted?
- When and how does a full table scan happen?
- What format is a prepared statement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

In other words, how does a database **work**?

To figure things out, I'm writing a database from scratch. It's modeled off sqlite because it is designed to be small with fewer features than MySQL or PostgreSQL, so I have a better hope of understanding it. The entire database is stored in a single file!

Sqlite

There's lots of documentation of sqlite internals on their website, plus I've got a copy of SQLite Database System: Design and Implementation.



sqlite architecture (https://www.sqlite.org/zipvfs/doc/trunk/www/howitworks.wiki)

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

- tokenizer
- parser
- code generator

The input to the front-end is a SQL query. the output is sqlite virtual machine bytecode (essentially a compiled program that can operate on the database).

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 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. In this tutorial, I'm not going to support multiple platforms.

A journey of a thousand miles begins with a single step, so let's start with something a little more straightforward: the REPL.

Making a Simple REPL

Sqlite starts a read-execute-print loop when you start it from the command line:

```
~ sqlite3
SQLite version 3.16.0 2016-11-04 19:09:39
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite> create table users (id int, username varchar(255), emai
sqlite> .tables
users
sqlite> .exit
~
```

To do that, our main function will have an infinite loop that prints the prompt, gets a line of input, then processes that line of input:

```
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->buffe
    }
  }
}
```

We'll define `InputBuffer` as a small wrapper around the state we need to store to interact with getline(). (More on that in a minute)

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

InputBuffer* new_input_buffer () {
  InputBuffer* input_buffer = (InputBuffer*)malloc(sizeof( Input
  input_buffer->buffer = NULL;
  input_buffer->buffer_length = 0;
  input_buffer->input_length = 0;

  return input_buffer;
}
```

Next, `print_prompt()` prints a prompt to the user. We do this before reading each line of input.

```
void print_prompt() { printf("db > "); }
```

To read a line of input, use getline():

```
ssize_t getline(char **lineptr, size_t *n, FILE *stream);
```

`lineptr` : a pointer to the variable we use to point to the buffer containing the read line. If it set to `NULL` it is mallocatted by `getline` and should thus be freed by the user, even if the command fails.

`n` : a pointer to the variable we use to save the size of allocated buffer.

`stream` : the input stream to read from. We'll be reading from standard input.

`return value` : the number of bytes read, which may be less than the size of the buffer.

We tell `getline` to store the read line in `input_buffer->buffer` and the size of the allocated buffer in `input_buffer->buffer_length`. We store the return value in `input_buffer->input_length`.

`buffer` starts as null, so `getline` allocates enough memory to hold the line of input and makes `buffer` point to it.

```
void read_input(InputBuffer* input_buffer) {
  ssize_t bytes_read =
      getline(&(input_buffer->buffer), &(input_buffer->buffer_l

  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;
}
```

Now it is proper to define a function that frees the memory allocated for an instance of `InputBuffer *` and the `buffer` element of the respective structure (`getline` allocates memory for `input_buffer->buffer` in `read_input`).

```
void close_input_buffer (InputBuffer* input_buffer) {
    free(input_buffer->buffer);
    free(input_buffer);
}
```

Finally, we parse and execute the command. There is only one recognized command right now : `.exit`, which terminates the program. Otherwise we print an error message and continue the loop.

```
if (strcmp(input_buffer->buffer, ".exit") == 0) {
  close_input_buffer(input_buffer);
  exit(EXIT_SUCCESS);
} else {
  printf("Unrecognized command '%s'.\n", input_buffer->buffer);
}
```

Let's try it out!

```
~ ./db
db > .tables
Unrecognized command '.tables'.
db > .exit
~
```

Alright, we've got a working REPL. In the next part, we'll start developing our command language. Meanwhile, here's the entire program from this part:

```
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct {
  char* buffer;
```

```
  size_t buffer_length;
  ssize_t input_length;
} InputBuffer;

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_l

  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);
}

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->buffe
    }
  }
}
```

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