Let's Build a Simple Database

Writing a sqlite clone from scratch in C

Overview

View on GitHub (pull requests welcome)

Part 5 - Persistence to Disk

< Part 4 - Our First Tests (and Bugs)

Part 6 - The Cursor Abstraction >

> *"Nothing in the world can take the place of persistence."* – *Calvin Coolidge*

Our database lets you insert records and read them back out, but only as long as you keep the program running. If you kill the program and start it back up, all your records are gone. Here's a spec for the behavior we want:

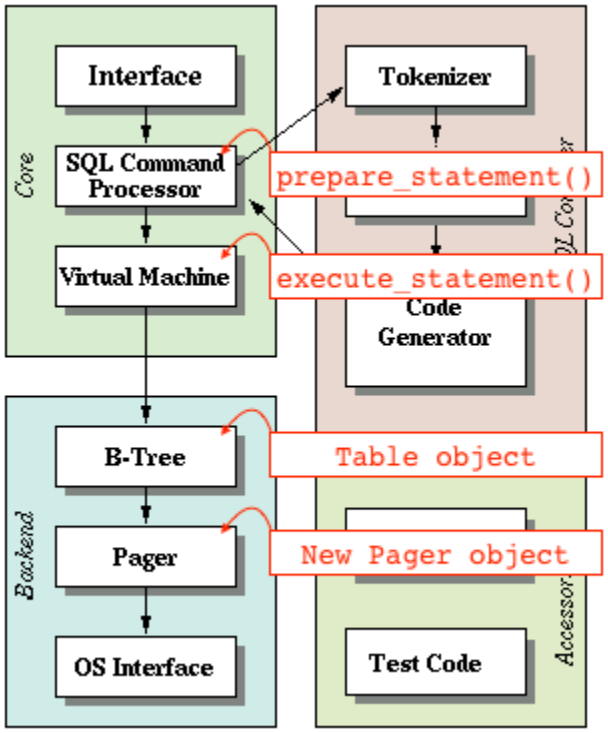
```
it 'keeps data after closing connection' do
  result1 = run_script([
    "insert 1 user1 person1@example.com",
    ".exit",
  ])
  expect(result1).to match_array([
    "db > Executed.",
    "db > ",
  ])
  result2 = run_script([
    "select",
    ".exit",
  ])
  expect(result2).to match_array([
    "db > (1, user1, person1@example.com)",
    "Executed.",
    "db > ",
```

```
  ])
end
```

Like sqlite, we're going to persist records by saving the entire database to a file.

We already set ourselves up to do that by serializing rows into page-sized memory blocks. To add persistence, we can simply write those blocks of memory to a file, and read them back into memory the next time the program starts up.

To make this easier, we're going to make an abstraction called the pager. We ask the pager for page number x, and the pager gives us back a block of memory. It first looks in its cache. On a cache miss, it copies data from disk into memory (by reading the database file).



How our program matches up with SQLite architecture

The Pager accesses the page cache and the file. The Table object makes requests for pages through the pager:

```
+typedef struct {
+  int file_descriptor;
+  uint32_t file_length;
+  void* pages[TABLE_MAX_PAGES];
```

```
+} Pager;
+
 typedef struct {
-  void* pages[TABLE_MAX_PAGES];
+  Pager* pager;
   uint32_t num_rows;
 } Table;
```

I'm renaming `new_table()` to `db_open()` because it now has the effect of opening a connection to the database. By opening a connection, I mean:

- opening the database file
- initializing a pager data structure
- initializing a table data structure

```
-Table* new_table() {
+Table* db_open(const char* filename) {
+  Pager* pager = pager_open(filename);
+  uint32_t num_rows = pager->file_length / ROW_SIZE;
+
   Table* table = malloc(sizeof(Table));
-  table->num_rows = 0;
+  table->pager = pager;
+  table->num_rows = num_rows;

   return table;
 }
```

`db_open()` in turn calls `pager_open()`, which opens the database file and keeps track of its size. It also initializes the page cache to all `NULL`s.

```
+Pager* pager_open(const char* filename) {
+  int fd = open(filename,
+                O_RDWR |      // Read/Write mode
+                    O_CREAT,  // Create file if it does not ex
+                S_IWUSR |     // User write permission
+                    S_IRUSR   // User read permission
+                );
+
```

```
+  if (fd == -1) {
+    printf("Unable to open file\n");
+    exit(EXIT_FAILURE);
+  }
+
+  off_t file_length = lseek(fd, 0, SEEK_END);
+
+  Pager* pager = malloc(sizeof(Pager));
+  pager->file_descriptor = fd;
+  pager->file_length = file_length;
+
+  for (uint32_t i = 0; i < TABLE_MAX_PAGES; i++) {
+    pager->pages[i] = NULL;
+  }
+
+  return pager;
+}
```

Following our new abstraction, we move the logic for fetching a page into its own method:

```
 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);
-  }
+  void* page = get_page(table->pager, page_num);
   uint32_t row_offset = row_num % ROWS_PER_PAGE;
   uint32_t byte_offset = row_offset * ROW_SIZE;
   return page + byte_offset;
 }
```

The `get_page()` method has the logic for handling a cache miss. We assume pages are saved one after the other in the database file: Page 0 at offset 0, page 1 at offset 4096, page 2 at offset 8192, etc. If the requested page lies outside the bounds of the file, we know it should be blank, so we just allocate some memory and return it. The page will be added to the file when we flush the cache to disk

later.

```
+void* get_page(Pager* pager, uint32_t page_num) {
+  if (page_num > TABLE_MAX_PAGES) {
+    printf("Tried to fetch page number out of bounds. %d > %d\r
+           TABLE_MAX_PAGES);
+    exit(EXIT_FAILURE);
+  }
+
+  if (pager->pages[page_num] == NULL) {
+    // Cache miss. Allocate memory and load from file.
+    void* page = malloc(PAGE_SIZE);
+    uint32_t num_pages = pager->file_length / PAGE_SIZE;
+
+    // We might save a partial page at the end of the file
+    if (pager->file_length % PAGE_SIZE) {
+      num_pages += 1;
+    }
+
+    if (page_num <= num_pages) {
+      lseek(pager->file_descriptor, page_num * PAGE_SIZE, SEEK_
+      ssize_t bytes_read = read(pager->file_descriptor, page, P
+      if (bytes_read == -1) {
+        printf("Error reading file: %d\n", errno);
+        exit(EXIT_FAILURE);
+      }
+    }
+
+    pager->pages[page_num] = page;
+  }
+
+  return pager->pages[page_num];
+}
```

For now, we'll wait to flush the cache to disk until the user closes the connection to the database. When the user exits, we'll call a new method called `db_close()`, which

- flushes the page cache to disk

- closes the database file
- frees the memory for the Pager and Table data structures

```
+void db_close(Table* table) {
+  Pager* pager = table->pager;
+  uint32_t num_full_pages = table->num_rows / ROWS_PER_PAGE;
+
+  for (uint32_t i = 0; i < num_full_pages; i++) {
+    if (pager->pages[i] == NULL) {
+      continue;
+    }
+    pager_flush(pager, i, PAGE_SIZE);
+    free(pager->pages[i]);
+    pager->pages[i] = NULL;
+  }
+
+  // There may be a partial page to write to the end of the fi
+  // This should not be needed after we switch to a B-tree
+  uint32_t num_additional_rows = table->num_rows % ROWS_PER_PAG
+  if (num_additional_rows > 0) {
+    uint32_t page_num = num_full_pages;
+    if (pager->pages[page_num] != NULL) {
+      pager_flush(pager, page_num, num_additional_rows * ROW_SI
+      free(pager->pages[page_num]);
+      pager->pages[page_num] = NULL;
+    }
+  }
+
+  int result = close(pager->file_descriptor);
+  if (result == -1) {
+    printf("Error closing db file.\n");
+    exit(EXIT_FAILURE);
+  }
+  for (uint32_t i = 0; i < TABLE_MAX_PAGES; i++) {
+    void* page = pager->pages[i];
+    if (page) {
+      free(page);
+      pager->pages[i] = NULL;
+    }
```

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

In our current design, the length of the file encodes how many rows are in the database, so we need to write a partial page at the end of the file. That's why `pager_flush()` takes both a page number and a size. It's not the greatest design, but it will go away pretty quickly when we start implementing the B-tree.

```
+void pager_flush(Pager* pager, uint32_t page_num, uint32_t size
+  if (pager->pages[page_num] == NULL) {
+    printf("Tried to flush null page\n");
+    exit(EXIT_FAILURE);
+  }
+
+  off_t offset = lseek(pager->file_descriptor, page_num * PAGE_
+
+  if (offset == -1) {
+    printf("Error seeking: %d\n", errno);
+    exit(EXIT_FAILURE);
+  }
+
+  ssize_t bytes_written =
+      write(pager->file_descriptor, pager->pages[page_num], siz
+
+  if (bytes_written == -1) {
+    printf("Error writing: %d\n", errno);
+    exit(EXIT_FAILURE);
+  }
+}
```

Lastly, we need to accept the filename as a command-line argument. Don't forget to also add the extra argument to `do_meta_command`:

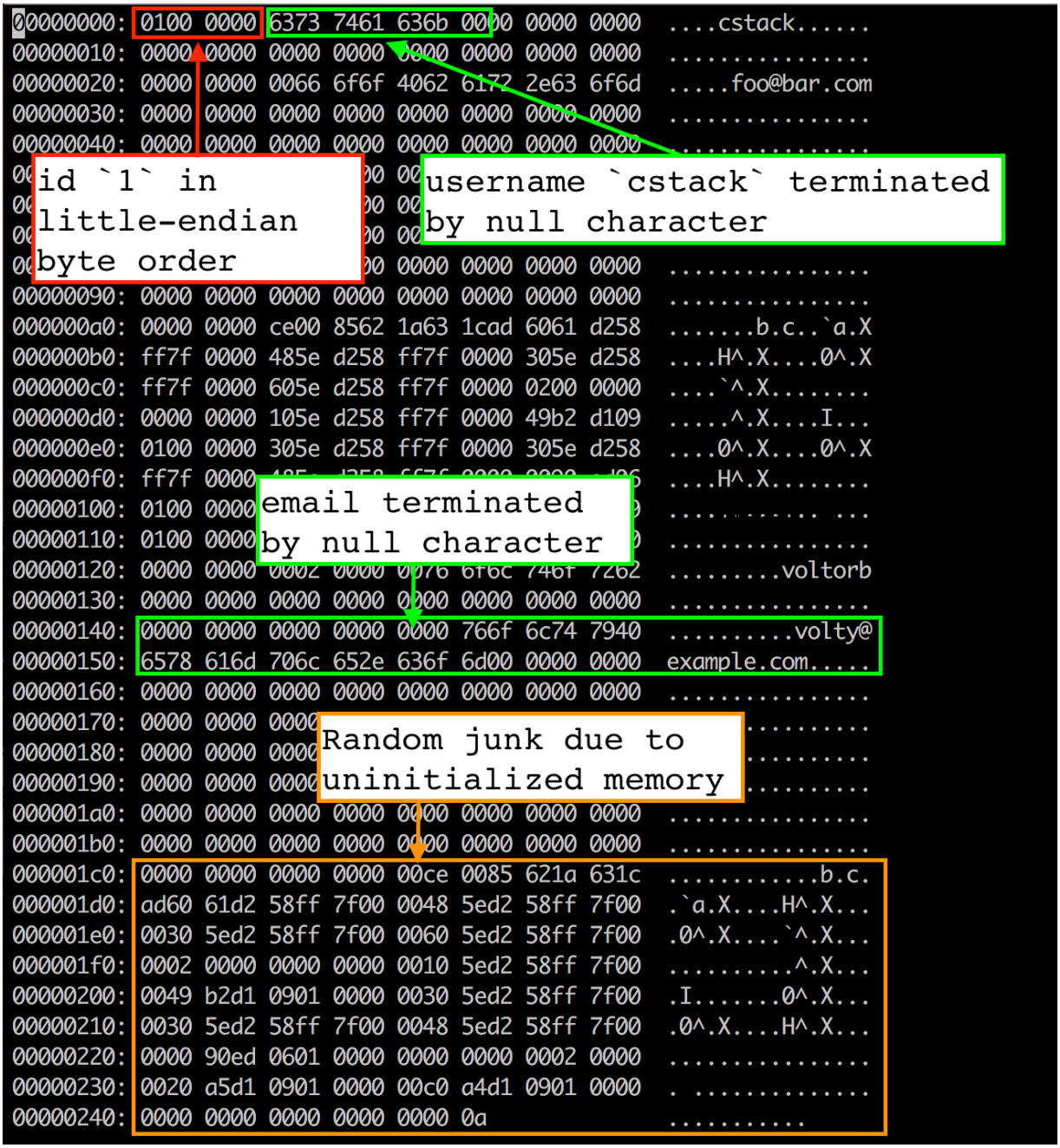
```
 int main(int argc, char* argv[]) {
-  Table* table = new_table();
+  if (argc < 2) {
+    printf("Must supply a database filename.\n");
+    exit(EXIT_FAILURE);
+  }
+
+  char* filename = argv[1];
+  Table* table = db_open(filename);
+
   InputBuffer* input_buffer = new_input_buffer();
   while (true) {
     print_prompt();
     read_input(input_buffer);

     if (input_buffer->buffer[0] == '.') {
-      switch (do_meta_command(input_buffer)) {
+      switch (do_meta_command(input_buffer, table)) {
```

With these changes, we're able to close then reopen the database, and our records are still there!

```
~ ./db mydb.db
db > insert 1 cstack foo@bar.com
Executed.
db > insert 2 voltorb volty@example.com
Executed.
db > .exit
~
~ ./db mydb.db
db > select
(1, cstack, foo@bar.com)
(2, voltorb, volty@example.com)
Executed.
db > .exit
~
```

For extra fun, let's take a look at `mydb.db` to see how our data is being stored. I'll use vim as a hex editor to look at the memory layout of the file:

```
vim mydb.db
:%!xxd
```



Current File Format

The first four bytes are the id of the first row (4 bytes because we store a `uint32_t`). It's stored in little-endian byte order, so the least significant byte comes first (01), followed by the higher-order bytes (00 00 00). We used `memcpy()` to copy bytes from our `Row` struct into the page cache, so that means the struct was laid out in memory in little-endian byte order. That's an attribute of the machine I compiled the program for. If we wanted to write a database file on my machine, then read it on a big-endian machine, we'd have to change our `serialize_row()` and `deserialize_row()` methods to always store and read bytes in the same order.

The next 33 bytes store the username as a null-terminated string. Apparently "cstack" in ASCII hexadecimal is `63 73 74 61 63 6b`, followed by a null character (00). The rest of the 33 bytes are unused.

The next 256 bytes store the email in the same way. Here we can see some random junk after the terminating null character. This is most likely due to uninitialized memory in our `Row` struct. We copy the entire 256-byte email buffer into the file, including any bytes after the end of the string. Whatever was in memory when we allocated that struct is still there. But since we use a terminating null character, it has no effect on behavior.

NOTE: If we wanted to ensure that all bytes are initialized, it would suffice to use `strncpy` instead of `memcpy` while copying the `username` and `email` fields of rows in `serialize_row`, like so:

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

Conclusion

Alright! We've got persistence. It's not the greatest. For example if you kill the program without typing `.exit`, you lose your changes. Additionally, we're writing all pages back to disk, even pages that haven't changed since we read them from disk. These are issues we can address later.

Next time we'll introduce cursors, which should make it easier to implement the B-tree.

Until then!

Complete Diff

```
+#include <errno.h>
+#include <fcntl.h>
```

```
 #include <stdbool.h>
 #include <stdio.h>
 #include <stdlib.h>
 #include <string.h>
 #include <stdint.h>
+#include <unistd.h>

 struct InputBuffer_t {
   char* buffer;
@@ -62,9 +65,16 @@ const uint32_t PAGE_SIZE = 4096;
 const uint32_t ROWS_PER_PAGE = PAGE_SIZE / ROW_SIZE;
 const uint32_t TABLE_MAX_ROWS = ROWS_PER_PAGE * TABLE_MAX_PAGES

+typedef struct {
+  int file_descriptor;
+  uint32_t file_length;
+  void* pages[TABLE_MAX_PAGES];
+} Pager;
+
 typedef struct {
   uint32_t num_rows;
-  void* pages[TABLE_MAX_PAGES];
+  Pager* pager;
 } Table;

@@ -84,32 +94,81 @@ void deserialize_row(void *source, Row* dest
   memcpy(&(destination->email), source + EMAIL_OFFSET, EMAIL_SI
 }

+void* get_page(Pager* pager, uint32_t page_num) {
+  if (page_num > TABLE_MAX_PAGES) {
+    printf("Tried to fetch page number out of bounds. %d > %d\
+      TABLE_MAX_PAGES);
+    exit(EXIT_FAILURE);
+  }
+
+  if (pager->pages[page_num] == NULL) {
+    // Cache miss. Allocate memory and load from file.
+    void* page = malloc(PAGE_SIZE);
+    uint32_t num_pages = pager->file_length / PAGE_SIZE;
```

```
+
+     // We might save a partial page at the end of the file
+     if (pager->file_length % PAGE_SIZE) {
+         num_pages += 1;
+     }
+
+     if (page_num <= num_pages) {
+         lseek(pager->file_descriptor, page_num * PAGE_SIZE, SE
+         ssize_t bytes_read = read(pager->file_descriptor, page
+         if (bytes_read == -1) {
+       printf("Error reading file: %d\n", errno);
+       exit(EXIT_FAILURE);
+         }
+     }
+
+     pager->pages[page_num] = page;
+   }
+
+   return pager->pages[page_num];
+ }
+
  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);
-   }
+   void *page = get_page(table->pager, page_num);
    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 = malloc(sizeof(Table));
-   table->num_rows = 0;
+Pager* pager_open(const char* filename) {
+   int fd = open(filename,
+         O_RDWR |      // Read/Write mode
```

```
+             O_CREAT,  // Create file if it does not exist
+         S_IWUSR |     // User write permission
+             S_IRUSR   // User read permission
+         );
+
+  if (fd == -1) {
+     printf("Unable to open file\n");
+     exit(EXIT_FAILURE);
+  }
+
+  off_t file_length = lseek(fd, 0, SEEK_END);
+
+  Pager* pager = malloc(sizeof(Pager));
+  pager->file_descriptor = fd;
+  pager->file_length = file_length;
+
   for (uint32_t i = 0; i < TABLE_MAX_PAGES; i++) {
-     table->pages[i] = NULL;
+     pager->pages[i] = NULL;
   }
-  return table;
+
+  return pager;
 }

-void free_table(Table* table) {
-  for (int i = 0; table->pages[i]; i++) {
-     free(table->pages[i]);
-  }
-  free(table);
+Table* db_open(const char* filename) {
+  Pager* pager = pager_open(filename);
+  uint32_t num_rows = pager->file_length / ROW_SIZE;
+
+  Table* table = malloc(sizeof(Table));
+  table->pager = pager;
+  table->num_rows = num_rows;
+
+  return table;
 }
```

```
 InputBuffer* new_input_buffer() {
@@ -142,10 +201,76 @@ void close_input_buffer(InputBuffer* input
   free(input_buffer);
 }

+void pager_flush(Pager* pager, uint32_t page_num, uint32_t size
+  if (pager->pages[page_num] == NULL) {
+     printf("Tried to flush null page\n");
+     exit(EXIT_FAILURE);
+  }
+
+  off_t offset = lseek(pager->file_descriptor, page_num * PAGE_
+                SEEK_SET);
+
+  if (offset == -1) {
+     printf("Error seeking: %d\n", errno);
+     exit(EXIT_FAILURE);
+  }
+
+  ssize_t bytes_written = write(
+     pager->file_descriptor, pager->pages[page_num], size
+     );
+
+  if (bytes_written == -1) {
+     printf("Error writing: %d\n", errno);
+     exit(EXIT_FAILURE);
+  }
+}
+
+void db_close(Table* table) {
+  Pager* pager = table->pager;
+  uint32_t num_full_pages = table->num_rows / ROWS_PER_PAGE;
+
+  for (uint32_t i = 0; i < num_full_pages; i++) {
+     if (pager->pages[i] == NULL) {
+         continue;
+     }
+     pager_flush(pager, i, PAGE_SIZE);
+     free(pager->pages[i]);
```

```
+     pager->pages[i] = NULL;
+   }
+
+   // There may be a partial page to write to the end of the fi
+   // This should not be needed after we switch to a B-tree
+   uint32_t num_additional_rows = table->num_rows % ROWS_PER_PAG
+   if (num_additional_rows > 0) {
+     uint32_t page_num = num_full_pages;
+     if (pager->pages[page_num] != NULL) {
+         pager_flush(pager, page_num, num_additional_rows * ROW
+         free(pager->pages[page_num]);
+         pager->pages[page_num] = NULL;
+     }
+   }
+
+   int result = close(pager->file_descriptor);
+   if (result == -1) {
+     printf("Error closing db file.\n");
+     exit(EXIT_FAILURE);
+   }
+   for (uint32_t i = 0; i < TABLE_MAX_PAGES; i++) {
+     void* page = pager->pages[i];
+     if (page) {
+         free(page);
+         pager->pages[i] = NULL;
+     }
+   }
+
+   free(pager);
+   free(table);
+}
+
 MetaCommandResult do_meta_command(InputBuffer* input_buffer, Ta
   if (strcmp(input_buffer->buffer, ".exit") == 0) {
     close_input_buffer(input_buffer);
-    free_table(table);
+    db_close(table);
     exit(EXIT_SUCCESS);
   } else {
     return META_COMMAND_UNRECOGNIZED_COMMAND;
```

```
@@ -182,6 +308,7 @@ PrepareResult prepare_insert(InputBuffer* ir
     return PREPARE_SUCCESS;

 }
+
 PrepareResult prepare_statement(InputBuffer* input_buffer,
                                 Statement* statement) {
   if (strncmp(input_buffer->buffer, "insert", 6) == 0) {
@@ -227,7 +354,14 @@ ExecuteResult execute_statement(Statement*
 }

 int main(int argc, char* argv[]) {
-  Table* table = new_table();
+  if (argc < 2) {
+      printf("Must supply a database filename.\n");
+      exit(EXIT_FAILURE);
+  }
+
+  char* filename = argv[1];
+  Table* table = db_open(filename);
+
   InputBuffer* input_buffer = new_input_buffer();
   while (true) {
     print_prompt();
```

And the diff to our tests:

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

```
@@ -28,6 +32,27 @@ describe 'database' do
     ])
   end

+  it 'keeps data after closing connection' do
+    result1 = run_script([
+      "insert 1 user1 person1@example.com",
+      ".exit",
+    ])
+    expect(result1).to match_array([
+      "db > Executed.",
+      "db > ",
+    ])
+
+    result2 = run_script([
+      "select",
+      ".exit",
+    ])
+    expect(result2).to match_array([
+      "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"
```

< Part 4 - Our First Tests (and Bugs)

Part 6 - The Cursor Abstraction >