**Building a Database From Scratch in Rust — Part 1**

[[Paolo Rechia](https://medium.com/@paolorechia?source=post_page-----6dfef2223673--------------------------------)](https://medium.com/@paolorechia?source=post_page-----6dfef2223673--------------------------------)

[Paolo Rechia](https://medium.com/@paolorechia?source=post_page-----6dfef2223673--------------------------------)

·

Follow

10 min read

·

Dec 31, 2023

108

4

**Introduction**

A couple of weeks ago, I took part in another Kaggle competition where I got stuck, so I’ve decided to start writing feature engineering pipelines in Rust (I’ll write more about this one another time) just as a learning experience.

Programming in Rust again after so long was a very rewarding experience, which prompted me to think: What could I build in Rust as a side-project that would be useful for studying purposes?

Since I work as a data engineer, the natural answer was: a Database! I could certainly learn better how beasts like Postgres work internally. And to me, the best way to learn is to build.

So after reading a [blog](https://cstack.github.io/db_tutorial/parts/part1.html) of someone that built a Database from scratch in C, using SQLite as the reference, I got inspired to do something similar.

I would follow the architecture of SQLite, giving me the freedom to diverge in some aspects.

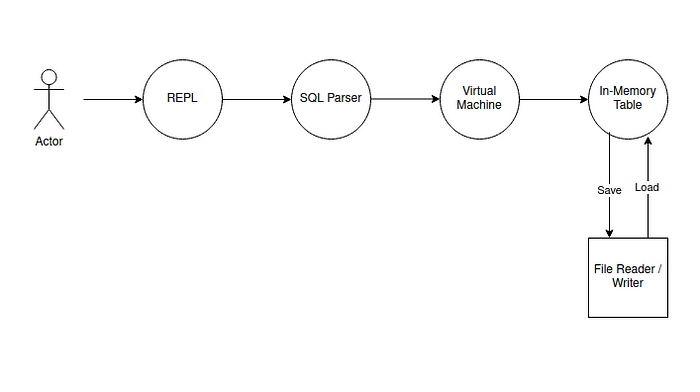
According to this blog (and the official SQLite documentation, this is roughly what we need to replicate SQL

1. A REPL
2. A tokenizer
3. A parser
4. A code generator
5. A virtual machine that interprets the generated code
6. A B+ Tree
7. Pager
8. OS Interface

However, since our goal is learning, we can adopt some simplifications, and build iteratively. So I created instead this roadmap for my first iteration on the database:

1. A REPL, with pretty printing of the query result
2. A tokenizer and a parser that support the SELECT clause
3. A code generator (more precisely, a command generator)
4. A virtual machine that interprets the generated code (or commands)
5. A table struct that stores data in a memory as a HashMap of Vectors
6. A hardcoded table for testing
7. Proper error propagation / handling (no panic, all errors are handled/displayed back to the user)
8. A serialization / deserialization method to write/read data from file

Here’s the overview of the architecture I ended up with:



SteelDB Architecture

[You can find the entire source code here, in my GitHub repository.](https://github.com/paolorechia/steeldb)

Over the remainder of this article, I’ll cover each component from this diagram, except the File Reader / Writer.

We’ll start with the in memory table struct, since it’s at the core of the database implementation.

**In Memory Table Struct**

I’ve defined our in-memory table like this:

#[derive(Debug, Clone)]  
pub enum DataType {  
 String(String),  
 Integer32(i32),  
 Float32(f32),  
}  
  
pub struct Table {  
 pub name: String,  
 pub fields: HashMap<String, DataType>,  
 pub columns: HashMap<String, Vec<DataType>>,  
 pub select\_columns: Vec<String>,  
}

Fields keeps track of the table ‘schema’, while columns contain the actual columns’ data. As you can probably guess from this code, we use a columnar format, rather than row-order.

You might also notice, I use ‘String’ type a lot. This is mostly for convenience, so I don’t have to deal with the lifetime annotations.

It does mean that my code makes a lot of unnecessary string clone operations around. This is unfortunately a trade-off I had to do taking into consideration my limited experience with Rust. Perhaps in the future, once I’m more comfortable managing string slices I might revisit this and attempt a more efficient implementation.

Anyway, let’s look at the signature of our most important methods, save and load:

pub fn save(  
 &self,  
 mode: SaveMode,  
 format: FileFormat  
) -> Result<(), TableErrors>  
  
pub fn load(  
 table\_name: String,  
 select\_columns: Vec<String>,  
 format: FileFormat,  
) -> Result<Table, TableErrors>

Let’s take a peek at the most important lines of the save method. You can see it’s pretty simple, it resolves the table path, and instantiates the appropriate file writer depending on the format.

let path = Path::new(&s);  
  
// Pick up correct writer  
let writer: Box<dyn Writer>;  
match format {  
 FileFormat::SimpleColumnar => {  
 writer = ColumnarWriter::new();  
 }  
}  
// Adapt to the given mode  
match mode {  
 SaveMode::Overwrite => {  
 let f = OpenOptions::new().write(true).create\_new(true).open(path);  
 if f.is\_err() {  
 println!("{:?}", f.unwrap\_err());  
 return Err(TableErrors::TableAlreadyExists);  
 }  
 let write\_result = writer.write(&self.fields, &self.columns, f.unwrap());  
 if write\_result.is\_err() {  
 let s = format!("{:?}", write\_result.unwrap\_err());  
 return Err(TableErrors::WriteError(s));  
 }  
 }

Using std::fs::OpenOptions gives us more flexibility in how we want to access our file. In this way, we create a new file if it does not exist, and generate an error if the file already exists.

Note: we assume here our write error is because the table already exists, which most often would be the case, but there’s a chance it’s caused by another reason. Therefore, for now I simply log the original error message along.

The load method is similar, but the reverse. Maybe it’s interesting to notice why we have this parameter select\_columns: Vec<String>. It’s used in this part of the function:

let result = reader.read(f, select\_columns.clone());  
// (...) check for error before unwrap  
let (fields, columns) = result.unwrap();  
  
for select\_col in select\_columns.iter() {  
 if !fields.contains\_key(select\_col) {  
 return Err(TableErrors::ColumnNotFound(select\_col.clone()));  
 }  
}

Notice we pass the select columns to the reader, so it can load in memory only what was requested (and not the whole table). After it’s loaded, we can check if everything we asked was found.

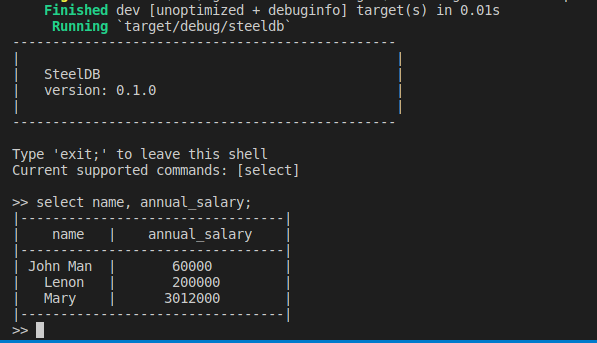
I will not go in details regarding the Reader / Writer, as that code is not so interesting in my opinion. It simply writes / reads this weird format I’ve created for storing data:

TABLE COLUMNAR FORMAT HEADER  
Field name: final\_grade; Type: f32; Number of elements: 3  
4.0  
3.2  
5  
Field name: name; Type: String; Number of elements: 3  
John Man  
Lenon  
Mary  
Field name: annual\_salary; Type: i32; Number of elements: 3  
60000  
200000  
3012000

When reading this format, the code has to handle several cases and it ends up being quite tedious to read. Specially considering it’s an inefficient format — not worth diving deeper in there. The curious reader can find the implementation in the [GitHub repository](https://github.com/paolorechia/steeldb/blob/main/src/database/file_io.rs) though.

**REPL**

Let’s circle back and take a look at the frontend:



Example of select query

As you can see, the table is resolved automatically, as there is no From clause implemented yet. For now, I’m using cargo test to run a test which creates the table for me.

If we delete or rename the test table, we see this instead:

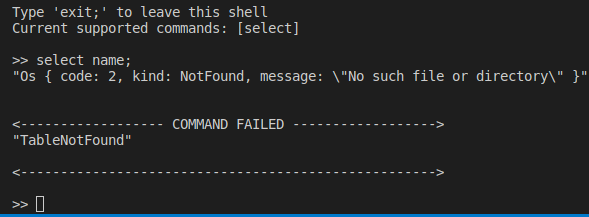
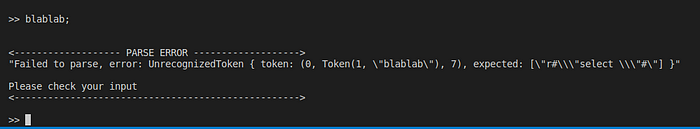


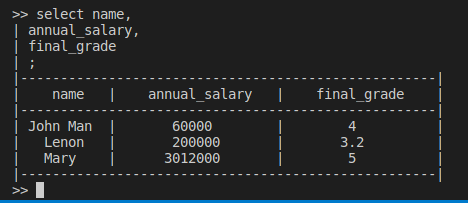
Table not found example

And if we type gibberish, we see this:



Parse Error Example

So how does this work? The REPL is fundamentally simple: a while true loop, that keeps storing the input into buffers, until it sees a ;. We store each line individually, so the user can break the command down into several lines, for instance:



Multi-line input example

Once the ; is found, we simply feed into a SteelDB struct instance and call execute on the query, which gives us a enum as a result, which makes it very clear which cases we should handle in our REPL:

if self.buffer.contains(";") {  
 if self.buffer.contains("exit") {  
 break;  
 }  
 self.is\_in\_multiline = false;  
 let execution\_result = self  
 .database  
 .execute(self.previous\_lines.join(" ").to\_lowercase());  
  
 match execution\_result {  
 ExecutionResult::VoidOK() => {  
 println!("OK!");  
 }  
 ExecutionResult::TableResult(table) => {  
 self.print\_table(&table);  
 }  
 ExecutionResult::ParseError(error) => {  
 println!("");  
 println!("");  
 println!("<------------------- PARSE ERROR ------------------->");  
 println!("{:?}", error);  
 println!("");  
 println!("Please check your input");  
 println!("<--------------------------------------------------->");  
 println!("");  
 }  
 ExecutionResult::CommandError(error) => {  
 println!("");  
 println!("");  
 println!("<------------------ COMMAND FAILED ------------------>");  
 println!("{:?}", error);  
 println!("");  
 println!("<---------------------------------------------------->");  
 println!("");  
 }  
 }

The print\_table is actually quite complicated, and calculates the length of the column size for printing into the terminal. It also depends on the in-memory Table struct we saw earlier. This REPL is not perfect, and does not handle large column values, but works rather fine for testing the database.

You probably notice the REPL calls database.execute, which is defined in the REPL new method:

pub fn new() -> Repl {  
 return Repl {  
 buffer: String::new(),  
 previous\_lines: Vec::<String>::new(),  
 database: SteelDB::new(),  
 is\_in\_multiline: false,  
 padding: 4,  
 };  
}

Here’s our SteelDB, which is our entry point into the actual database code. While it’s not in the diagram, it essentially glues the REPL, the Parser and the virtual machine together:

pub struct SteelDB {  
 virtual\_machine: VirtualMachine,  
}  
  
pub enum ExecutionResult {  
 TableResult(Table),  
 VoidOK(),  
 ParseError(String),  
 CommandError(String),  
}  
  
impl SteelDB {  
 pub fn new() -> SteelDB {  
 return SteelDB {  
 virtual\_machine: VirtualMachine::new(),  
 };  
 }  
 pub fn execute(&mut self, user\_input: String) -> ExecutionResult {  
 let result = parse(user\_input);  
 match result {  
 Ok(commands) => {  
 let command\_result = self.virtual\_machine.execute(commands);  
 // translate CommandResult into ExecutionResult  
 // we do not want to make the outer layer import any enum except ExecutionResult  
 match command\_result {  
 CommandResult::RetrievedDataSuccess(table) => {  
 return ExecutionResult::TableResult(table);  
 }  
 CommandResult::VoidSuccess => return ExecutionResult::VoidOK(),  
 CommandResult::Error(error) => {  
 return ExecutionResult::CommandError(error);  
 }  
 }  
 }  
 // translate ParseError into ExecutionResult  
 Err(ParseError::Error(error)) => {  
 return ExecutionResult::ParseError(error);  
 }  
 }  
 }  
}

**The Parser**

Inside the SteelDB main source code, our parser source code is quite small:

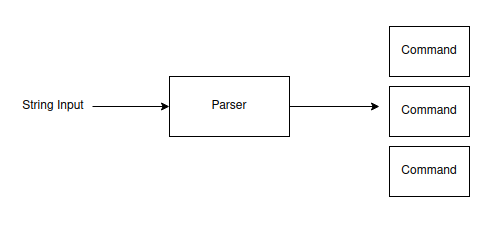
use super::command::Command;  
use super::config::DEFAULT\_TABLE;  
pub use steeldb\_parser::{parse\_select, ParseError};  
  
  
pub fn parse(input: String) -> Result<Vec<Command>, ParseError> {  
 let result = parse\_select(input);  
 match result {  
 Ok(columns) => {  
 return Ok(vec![Command::SelectFrom(  
 columns,  
 DEFAULT\_TABLE.to\_string(),  
 )]);  
 }  
 Err(error) => {  
 return Err(error);  
 }  
 }  
}

This is because the actual parser is written as a separate crate steeldb\_parser.

Before we look into that, notice how our parser is simply mapping input into commands, which in turn are also defined as enums:

pub enum Command {  
 SelectFrom(Vec<String>, String), // columns, table\_name  
 Stub,  
}

Graphically, this is what the parser is doing:



The parser converts a string into commands

If you glance back at the SteelDB code, you’ll notice the command is fed into the virtual machine.

So how do we build this parser?[Using this amazing library: lalrpop.](https://github.com/lalrpop/lalrpop)

Basically, we just have to define a grammar in a file that ends with .larlpop, for instance, here’s how I defined the SELECT clause currently:

grammar(v: &mut Vec<String>);  
  
pub Select: () = {  
 SELECT <c:Columns> SEMICOLON => {}  
};  
  
Columns: () = {  
 <l:LITERAL> => v.push(l),  
 Columns "," <l:LITERAL> => {  
 v.push(l);  
 }  
}  
  
SELECT: String = <s:r"select "> => s.to\_string();  
LITERAL: String = <s:r"[a-z\\*\_0-9]+"> => s.to\_string();  
SEMICOLON: String = <s:r";"> => s.to\_string();

Then in my steeldb\_parser `lib.rs` file I can define the following:

use lalrpop\_util::lalrpop\_mod;  
  
lalrpop\_mod!(pub select); // synthesized by LALRPOP  
  
#[derive(Debug)]  
pub enum ParseError {  
 Error(String),  
}  
  
pub fn parse\_select(input: String) -> Result<Vec<String>, ParseError> {  
 let mut result: Vec<String> = vec![];  
 let parser = select::SelectParser::new();  
 let maybe\_error = parser.parse(&mut result, input.as\_str());  
 match maybe\_error {  
 Ok(\_) => {  
 return Ok(result);  
 }  
 Err(error) => {  
 let error = format!("{:?}", error);  
 return Err(ParseError::Error(format!(  
 "Failed to parse, error: {}",  
 error  
 )));  
 }  
 };  
}

This library uses a build hook to automatically generate the select module with the SelectParser that implements a parser for the grammar we defined previously. The function parse\_select is what we use in our database. Of course, once we add a second command, we’ll have to make this a bit more elaborate, as it is hardcoded for the select command.

Either way, next to the last topic: the virtual machine.

**Virtual Machine**

This is actually a very simple implementation. We just take the list of commands we had and for each of them, execute the appropriate code. In particular for this iteration, we map the command Command::SelectFrom to a Table::load. Like the parser, this will be extended to support more commands, but overall I expect less structural changes in this layer. Here’s the source code:

impl VirtualMachine {  
 pub fn new() -> VirtualMachine {  
 return VirtualMachine {};  
 }  
  
 pub fn execute(&self, commands: Vec<Command>) -> CommandResult {  
 // keep track of last command execution  
 // might be useful when implementing nested commands  
 let mut maybe\_command\_result: Option<CommandResult> = None;  
  
 // the reason we implement this as a list of commands is to supported  
 // the execution of nested commands in the future  
 // this assumes the parser built a list of commands in the right order of execution  
 for command in commands {  
 if let Command::SelectFrom(columns, table\_name) = command {  
 let table\_result = Table::load(table\_name, columns, FileFormat::SimpleColumnar);  
  
 // if we found an error, we want to immediately abort the nested execution  
 if table\_result.is\_err() {  
 let error = format!("{:?}", table\_result.unwrap\_err());  
 return CommandResult::Error(error);  
 } else {  
 // if our command succeeds, we want to save the result in case the next command needs it  
 let table = table\_result.unwrap();  
 maybe\_command\_result = Some(CommandResult::RetrievedDataSuccess(table));  
 }  
 } else if let Command::Stub = command {  
 return CommandResult::VoidSuccess;  
 };  
 }  
  
 // once we finish going through the list, the last command result is our final one, let's return it  
 match maybe\_command\_result {  
 Some(command\_result) => command\_result,  
 None => CommandResult::Error("Empty command FIFO".to\_string()),  
 }  
 }  
}

**Conclusion**

Building a simple database from scratch in Rust was actually much easier than I initially imagined! Of course, it’s still just a toy project, but nothing stops someone from taking this skeleton and transforming it into an actually useful database.

I plan to continue extending it, and possibly writing more articles on the subject as I implement more features for the SteelDB.

What I really liked about writing it in Rust is how explicit you have to be with error handling. It makes it really easy to propagate all cases back to the frontend, rather than some weird crash along the way. All one has to do is to consistently handle all cases of Option and Result types.

I also noticed that my code ended up quite verbose when handling errors, which made me realize how useful it is to master the ? [operator](https://doc.rust-lang.org/book/ch09-02-recoverable-errors-with-result.html#where-the--operator-can-be-used). Unfortunately, I’m not quite there yet and need additional practice before writing cleaner Rust code.

Anyway, hope you enjoyed this article! Have a good one!