Implementation of an SQL Parser in Go and RUST

Ravina Gelda, Abhay Pande

June 2020

Abstract

As part of the project for CSE 210A, we implemented a simple SQL parser in Go and Rust. The goal of the project was to gain fluency in writing code in two new languages while following software development principles. We also hoped to compare the two languages in terms of the ease of writing and maintaining code as well as in terms of latency.

1 Introduction

The motivation for this project was to familiarize ourselves with the syntax as well as the coding practices for two relatively new languages, Golang and Rust. We chose to implement a parser for a subset of the Structured Query Language (SQL), in both languages to achieve this objective. SQL in itself is a fairly complex declarative language which in itself has different types of commands. While the Data Definition Language of SQL allows users to define the relational schema of the underlying database, the Data Control Language defines access to different resources within the database. For the project, given the limited amount of time, we concentrated only on a subset of the Data Modification Language of SQL. We implemented parsers that, given an SQL query, generated an internal Abstract Syntax Tree, in both languages. Our main learning goal for this project was to learn new languages and gain some insights into their implementation details by developing a non-trivial library in both.

1.1 Scope of the Project

For the purposes of our implementation, we selected a subset of SQL DML queries from [1] that the parser will target. We limited the scope to simple SELECT, INSERT, UPDATE and DELETE queries. Given the limited amount of time we had for the project, we chose not to address nested queries at the moment. In addition, we had initially planned to target GROUP BY and HAVING clauses of the SQL DML, but given the limitations of time we were unable to complete their implementations. In addition, the implementations currently

do not have any support for different SQL dialects. All queries support the WHERE clause and multiple conditional clauses can be ANDed together.

2 Parser Design

The design of the parsers in the two languages followed the same logical process. Each parser consisted of two separate modules: a tokenizer and a parser.

2.1 Tokenizer

The tokenizer parsed the input query and split it into a collection of tokens. These tokenizer identified special SQL keywords like SELECT, DELETE FROM, UPDATE , INSERT INTO, WHERE, SET, AND. It also identified the operators (<=, <, >=, >, =, ! =) that one expected in the SQL query. All other tokens were treated as query specific information, like names of tables, fields etc. The output of the tokenizer was a list of tokens, represented in our implementation as strings, which were used to build the internal Query AST. The tokenizer also identified certain malformed queries. For examples, scenarios where there were no corresponding closing braces for a given opening brace.

2.2 Parser

The parser used the tokens generated by the tokenizer to build the internal query representation. The parser is implemented as a Finite State Machine. Based on the current position in the finite state machine and the token being processed, an appropriate aspect of the query is built. If the token being processed is not as expected, an appropriate error is raised.Table 1 gives an overview of the transitions in the finite state machine. In case of scenarios where no valid token is found in the table, an appropriate error is raised.

3 Implementations

3.1 Golang

The Golang parser is implemented in 3 parts and is composed of 3 main files described in subsections 3.1.1 3.1.2 3.1.3. The overall Go implementation included about 968 lines of code. Table 3 provides details about lines of code written for each module implementation.

3.1.1 AST

Structure for AST is defined in the file query.go. The struct in 1 consist of the fields which are defined as shown in Table 2.

Initial State	Token	Next State
stepType	SELECT	selectField
stepType	INSERT INTO	insertTable
stepType	UPDATE	updateTable
stepType	DELETE FROM	deleteTable
selectField	”field name”	selectCommaOrFrom
selectCommaOrFrom	,	selectField
selectCommaOrFrom	FROM	selectFromTable
selectFromTable	”table name”	stepWhere
stepWhere	WHERE	whereField
whereField	”field name”	whereOperator
whereOperator	”valid operator”	whereValue
whereValue	”field value”	whereAnd
insertTable	”table namme ”	insertFieldsOpenaingParems
insertFieldsOpenainParens	’(’	insertFields
insertFields	”field value”	insertFieldsCommaOrClosingParens
insertFieldsCommaOrClosingParens	’,’	insertFields
insertFieldsCommaOrClosingParens	’)	’ insertFieldValues
insertFieldValues	VALUES	insertFieldValues
deleteTable	”table name”	stepWhere
updateTable	”table name”	updateSet
updateSet	SET	updateField
updateField	”field name”	updateEquals
updateEquals	=	updateValue
updateValue	”value”	updateCommaOrWhere
updateCommmaOrWhere	WHERE	whereField
updateCommaOrWhere	,	updateField

Table 1: The state transitions defined for the SQL Parser for both Languages

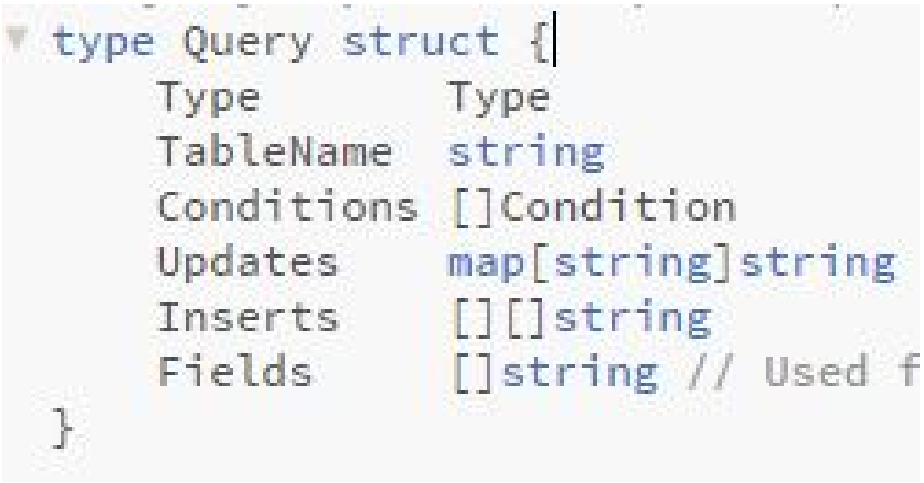

Figure 1: AST struct for SQL parser in Go

Field	Definition
Type	Records type of query (SELECT/UPDATE/INSERT/DELETE)
TableName	String for table name
Conditions	list of operators and operands under the where clause
Updates	map of variables to values to be updated
Inserts	list of values to be inserted
Fields	fields selected from table for insert into or select clause

Table 2: The per module break down of the lines of code for Go

3.1.2 Tokenizer

Tokenizer is implemented in token.go file. It has two main function as described below:

1. peek: Takes SQL query as input and check the words in the string if it belongs to identifiers(lke select, insert into etc.), does not belong to keywords and is string inside quotes, or is it an operator, or asterisk or comma or white space. and returns the parsed word

2. pop is same as peek with only one difference that along with returning the parsed word it also increases the tokenizer position to the next word in the SQL string

3.1.3 Parser

Parser main functionality is implemented in parser.go file.

1. doParse: In this function a tokenizer is called and the token is used to determine the next step in the FSM in fig 2. This process is repeated till

the end of SQL string is reached. As the FSM has predefined transition states, if any anomaly is detected in the SQL string, it is reported as error.

2. validate: In this function, one last validation is done after the end of query is reached. The parser might find the end of the string before arriving at a complete query definition. It's probably a good idea to implement a function that looks at the generated "query" struct, and returns an error if it's incomplete or otherwise wrong.

Working with Golang was surprisingly a pleasant experience. Having worked mostly with Python, it was good to see that Go also has string manipulation functions like append, trimspace, split etc. Go also follows the same noting of slicing as in Python, which makes it so much easier to work with strings, which was a major part of the Tokenizer in the parser implementation. I liked the dependency-wise scalability in Go. Unused dependencies are a compile-time error in Go. The compiler reads the object file for dependency, not its source code, which makes compiler efficient. Go has package system which has properties similar to libraries. this design provides clarity. Community support for the language is vast and is increasing as the popularity of the language is increasing. But there were times when I was stuck and couldn't debug the code, and that lead to discoveries of some weird syntax in Go like An identifier may be exported to permit access to it from another package only if the first character of the identifier's name is a Unicode upper case letter; and the identifier is declared in the package block or it is a field name or method name. So basically only functions / variables starting with a capital letter would be usable outside the package. [10]. One missing feature in Go is that it does not have classes.

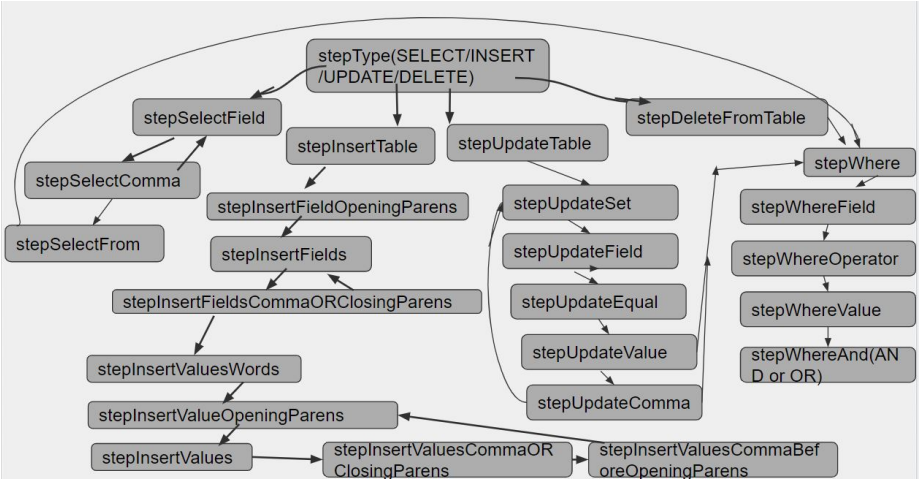

Figure 2: Test Case execution for the Go Implementation of the Parser

Module	Lines
AST(query.go)	60
Tokenizer(token.go)	75
parser(parser.go)	400
parser testing (parsertest.go)	420
main (SQLparsermain.go)	13

Table 3: The per module break down of the lines of code for Go

Module	Lines
AST	81
Parser	414
Parser Tests	557
Tokenizer	245

Table 4: The per module break down of the lines of code for Rust

3.2 Rust

Adhering to the design of the parser, the Rust implementation has three main submodules :

1. AST: This submodule defined the structs for the query object. This is the internal representation generated from the input query string to be parsed by the parser.

2. Tokens: This submodule captures the implementation of the tokenizier in rust. An object of type Tokenizer is used by the parser module to delegate all tokenization related functions to it.

3. Parser: This submodule contains the implementation of the parser.

The overall rust implementation, including the implementation of test cases, spans about 1300 lines. Table 4 below gives the details of the lines that each module spans. Working with Rust, especially for someone who is getting started, can have its own learning curve. There are certain aspects of using the language that are fairly well defined. In particular, the package manager cargo is well documented and is recommended to use from the default rust installation page [6] . We were also able to find linter plugins for VSCode which really helped understanding the errors and debugging them. There is significant support for learning the language online in terms of tutorials [5] as well as stack overflow. However, we fund Rust to be very verbose. Some simple operations, like printing an enum to stdout, required implementation of specific traits (here Debug). Moreover, the rigorous compiler and the scope rules for rust, which make the language so attractive in terms of avoiding memory related issues, were difficult to get used to.

4 Testing

4.1 Golang

The testing package provides the tools we need to write unit tests and the go test command runs tests. For the source file named something like parser.go, and the test file for it would then be named `parser_test.go`. A test is created by writing a function with a name beginning with Test. Writing tests can be repetitive, so it's idiomatic to use a table-driven style, where test inputs and expected outputs are listed in a table and a single loop walks over them and performs the test logic. t.Error* will report test failures but continue executing the test. t.Fail* will report test failures and stop the test immediately. t.Run enables running "subtests", one for each table entry. These are shown separately when executing go test -v [7]. Sample output after running testcases for Go implementation is as shown in fig 3.

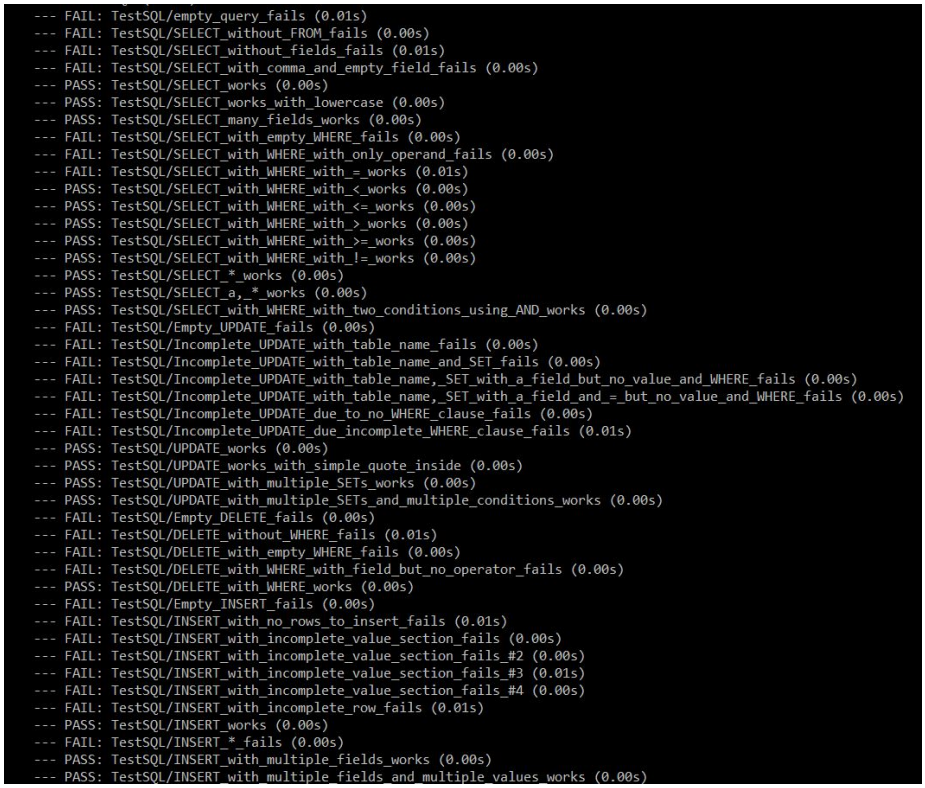

Figure 3: Test Case execution for the Go Implementation of the Parser

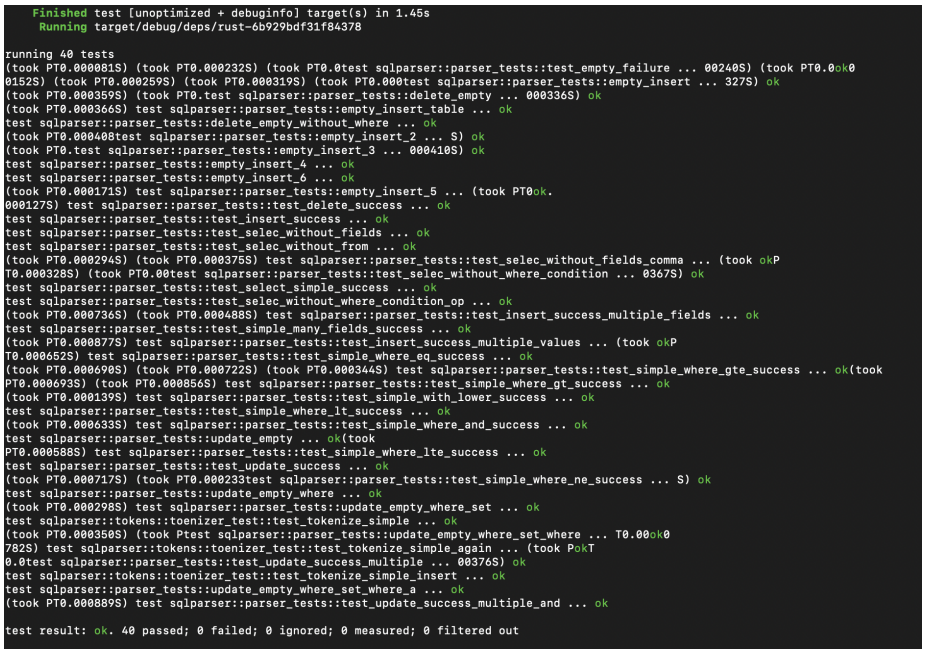

Figure 4: Test Case execution for the Rust Implementation of the Parser

4.2 Rust

We did not need to include any external crates(packages) other than the standard distribution, to start writing unit tests for our Rust. While writing tests can be repetitive, we chose to do so, as it allows us with flexibility to run single tests when writing them. We implemented about 40 unit test cases for the rust implementation of the parser, as can be seen in figure 4. The tests could be run by executing the command `cargo test` in the root folder, which contains the Cargo.toml file.

5 Latency Comparison

We compared the latency of the parsing process for the two parser implementations that we had. For the purposes of the comparison, we selected a set of 10 queries, that included SELECT, UPDATE, INSERT as well as DELETE queries in its mix. We parsed the queries in a single threaded manner in each parser and computed the amount of time taken by each implementation to complete the parsing process. Figure 5 shows the average execution time for each implementation. We see that our rust implementation is significantly faster than our Go implementation. We investigated the cause for the delay in the Go implementation, but could not find any significant problem. Thus, we have come to the conclusion that the Rust implementation is faster than that of Go.

Figure 5: Comparison of the Execution time for Go and Rust Implementations

6 Contributions

While we started off with the idea of equally dividing work in both implementations between both teammates, due to the constraints in time, we ended up each contributing more significantly to one implementation than the other. Thus, the implementation of the Go parser was primarily done by Ravina and the Rust parser was done by Abhay. Each team member contributed to collaborated in the design of the parsers, and helped with troubleshooting wherever possible. The team members worked on their corresponding sections for the presentation and final reports.

7 Challenges

Apart from the challenges mentioned in the corresponding sections on Golang parser implementation and Rust implementation, One of the major challenges was to decide the AST struct for parsed queries so that it can be used by the applications (Database drivers) and scaled for parsing other SQL queries which are not yet implemented as a part of our parser implementation.

Given the circumstances, it was sometimes difficult to work together leaving less occasions for collaboration.

8 Conclusion

The project gave us an opportunity to familiarise ourselves with two languages that we were not familiar with. We also learned about the complexities of SQL which explains its wide use in the database community. The significant complexities of the project, along with the significant learning curve, resulted in us having to reduce the scope of the implementation. However, as the project ended we feel that while we might not know everything about these languages, we have laid a decent foundation to build upon our knowledge of these languages.

9 References

References

[1] SQL Grammar Reference

[2] https://www.miek.nl/go/

[3] https://github.com/dariubs/GoBooks

[4] https://www.rust-lang.org/learn

[5] Rust Tutorials

[6] Rust Installation Page

[7] Testing Golang

[8] How to implement an SQL parser

[9] Go download and environment setup

[10] Go syntax help