SQL Query Parsing Using LL(1) Parser

Arjun Menon
Department of Computer Science and Engineering
Amrita School of Computing, Bengaluru
Amrita Vishwa Vidyapeetham
India
bl.en.u4aie20003@bl.students.amrita.edu

Gaurang Chaudhary
Department of Computer Science and Engineering
Amrita School of Computing, Bengaluru
Amrita Vishwa Vidyapeetham
India
bl.en.u4aie20035@bl.students.amrita.edu

Mudragiri Nitish Narayan
Department of Computer Science and Engineering
Amrita School of Computing, Bengaluru
Amrita Vishwa Vidyapeetham
India
bl.en.u4aie20035@bl.students.amrita.edu

Dr. Meena Belwal.*
Department of Computer Science and Engineering
Amrita School of Computing, Bengaluru
Amrita Vishwa Vidyapeetham
India
b_meena@blr.amrita.edu

Abstract: In this paper we build a LL(1) parser which is a top-down parser that parses the valid SQL(Structured Query Language) statements. By using LL(1) parsing techniques, this paper helps to recognize the complex syntax of SQL queries using the grammar defined. SQL grammar is pre-defined which is augmented grammar with no left recursions. It is left factored. It initially computes first and follow sets for the SQL grammar. LL(1) parsing table is constructed based on the first and follow sets. Then for parsing we do the stack implementation for the input SQL query. The code is done in java. The method proposed analyze and validates the SQL queries efficiently.

Keywords: **LL(1) parser, SQL query, first and follow sets, Stack Implementation**

I. INTRODUCTION

In the area of database management systems, SQL is a generic language used to communicate with the relational database. Parsing SQL query is the process of analysing them and processing them by validating the syntactic structure. Predefined grammar helps for the validation and efficient execution of the SQL query. There are mainly two types of parsing techniques. They are top-down parsing and bottom-up parsing. Top-down parsing includes recursive descent parser and predictive or LL(1) parser whereas bottom-up parsing includes LR(0) parser, SLR parser, Canonical LR(1) parser, LALR(1) parser. Moreover, the selection of parsing technique is done based on the complexity. This paper proposes an alternative approach of LL(1) parser which is efficient and easy to implement. The methodology includes computing first and follow sets, generating parsing table and stack based implementation to parse the string.

II. LITERATURE SURVEY

[1] A new top-down parsing algorithm to accommodate ambiguity and left recursion in polynomial time

This work describes a novel top-down parsing algorithm that can handle left recursion and ambiguity in polynomial time. The benefits of top-down backtracking language processors are discussed in the study, including their capacity to build ambiguous grammars, simplicity in use, modularity, and tight kinship between code and grammar structures. In addition, the paper tells prevalent myths concerning top-down processing, such as the idea that left-recursion must be eliminated before top-down processing may be applied. The novel technique described in the study can handle left-recursive productions and prevent exponential complexity. In general, the study offers insights into the creation of effective and dependable parsing algorithms that can cope with difficult grammars.

III. THEORY AND CONCEPT

Left-to-right, Leftmost derivation with 1 symbol lookahead is the abbreviation for the LL(1) parser. It is a form of top-down parsing method applied to the study and processing of context-free grammars in computer science. A parsing table created from the grammar is often used to implement LL(1) parsers.

IV. METHODOLOGY

The below are the steps in the code
First we will be asked to enter the number of tables in our SQL query and the we will enter the tables. Then we will be asked to enter the column name if we have to make any operations on columns. Then it will print the first and follow sets and parsing table. We have to enter the SQL query. Then the stack implementation is done to verify whether the input query is valid or not. The string will be accepted if $ will remain in stack after parsing complete input.

- Representation of grammar:

A <Map> object called grammar is used to represent the grammar rules. A list of productions that are derived from each non-terminal symbol is provided.

- Computation of First Sets:

Each non-terminal symbol in the grammar has its first sets determined by the computeFirstSets() method.
The first set is calculated for a particular non-terminal symbol using the computeFirstSet(String nonTerminal) function.
The set of terminals that can be the first symbol in a derivation from a non-terminal is known as the first set of that non-terminal.

- Computation of Follow sets:

For each non-terminal symbol in the grammar, the follow sets are computed using the computeFollowSets() method.
The follow set for a certain non-terminal symbol is calculated using the computeFollowSet(String nonTerminal) function. The set of terminals that may follow a non-terminal in a derivation is known as the follow set of a non-terminal.

- Table Generation for Parsing:

The initial sets, follow sets, and grammar rules are used to create the parsing table by the generateParsingTable() method. The parsing table is represented as a Map>>, where the list denotes the production to be used and the inner map denotes terminals. The outside map denotes non-terminals. To determine whether there are any conflicts in the parsing table, conflict detection is carried out.
If conflicts exist, the grammar is not LL(1) and the program terminates.

- Parsing input:

A request for a SQL query is made to the user. Tokenizing the input string results in its storage in an array of tokens.
Based on the parsing table, the parsing algorithm performs parsing using a stack. The start sign is present in the stack at first. The top of the stack is inspected at each step. It is compared to the current input token if it is a terminal. The relevant production is pulled from the parsing table and pushed onto the stack if it is a non-terminal. Until the stack is empty or all input tokens have been processed, the parsing process continues. The input is accepted if the stack is empty and all input tokens have been handled. If not, it is turned down.

V. RESULTS

The below is the implementation of an SQL query with table name as customers and column name age.
SQL query : DELETE FROM customers WHERE age > 50

```
Enter the number of table(s): 1
Enter the table name #1: customers
Accepted table name: customers
Does the statement include a change in the column?
1. Yes
2. No
1
Enter the number of column(s): 1
Enter the column name #1: age
```

Figure 1: Entering table name and column name

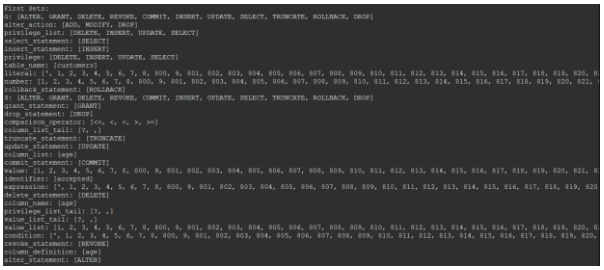

Figure 2: First Sets

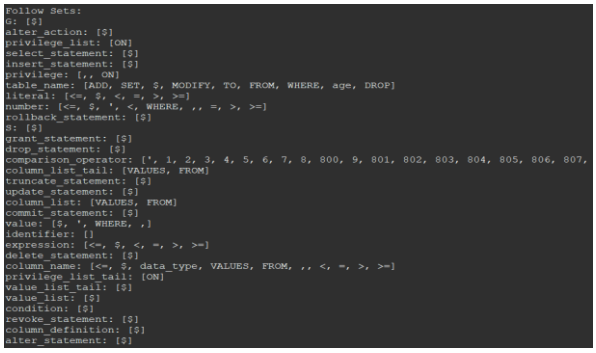

Figure 3: Follow Sets

```
[G]
[$, S]
[$, delete_statement]
[$, condition, WHERE, table_name, FROM, DELETE]
[$, condition, WHERE, table_name, FROM]
[$, condition, WHERE, table_name]
[$, condition, WHERE, customers]
[$, condition, WHERE]
[$, condition]
[$, expression, comparison_operator, expression]
[$, expression, comparison_operator, column_name]
[$, expression, comparison_operator, age]
[$, expression, comparison_operator]
[$, expression, >]
[$, expression]
[$, literal]
[$, number]
[$, 50]
[$]
```

Figure 4: Stack Implementation

```
----------------Parsing Result----------------

Input: DELETE FROM customers WHERE age > 50 $
Acceptance: true
DELETE FROM customers WHERE age > 50 $ is valid input
```

Figure 5: Parsing Result

VI. CONCLUSION

In this project we were able to successfully parse SQL queries with LL(1) parser. LL(1) parsers are efficient and can handle a wide range of context-free grammars. Parser is able to identify and parse different SQL statements, such as SELECT, INSERT, UPDATE, and DELETE, along with their expressions. Parser can be integrated into database management systems, query optimization, data analysis, or other related domains to improve query processing which enhances overall system performance.

ACKNOWLEDGEMENT

We are thankful to our course instructor Dr. Meena Belwal for providing this opportunity and guiding us through out the project.

REFERENCES

[1] Frost, Richard & Hafiz, Rahmatullah. (2006). A new top-down parsing algorithm to accommodate ambiguity and left recursion in polynomial time. SIGPLAN Notices. 41. 46-54. 10.1145/1149982.1149988.

[2] "Compilers: Principles, Techniques, and Tools" by Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman

[3] "Modern Compiler Implementation in C" by Andrew W. Appel

[4] https://www.codingninjas.com/codestudio/library/ll-parser-in-compiler-design

[5] https://andrewbegel.com/cs164/ll1.html