A Critique of Modern SQL And A Proposal Towards A Simple and Expressive Query Language

Prepared by:
Mayank Mishra(211210038)
Sonali Kashyap(211210065)

Submitted to:
Dr. Shelly Sachdeva

Table of Content

Critique of Modern SQL

1.1. Irregular Pseudo-English Syntax

•Complex Grammar: The irregularity of modern SQL makes it hard to learn, write, and implement, leading to impenetrable error messages.

•Challenges for Learners: SQL's irregular syntax poses difficulties for learners, especially when faced with real-world queries.

1.2.Lack of Extensibility and Abstraction

•Absence of Abstraction: SQL lacks mechanisms for abstraction, hindering its power and flexibility.

•Functional Limitations: The inability to define new operators and pass expressions limits the language's capabilities.

1.3: Semantic Ordering Constraints

•Implicit Ordering Rules: Semantical ordering constraints not visible in the syntactic structure.

•Lack of Portability: SQL's lack of portability across different systems.

Syntax-Related Challenges

2.1: Unhelpful Error Messages

•Impenetrable Messages: SQL's irregular grammar leads to unhelpful error messages, making debugging and implementation challenging.

•Compile Time Errors: A significant number of queries result in compile time errors, indicating the syntax-related challenges faced by learners.

2.2: Lack of Syntactical Orthogonality

•Inconsistent Syntax: The syntax of SQL lacks orthogonality, leading to difficulties in writing and understanding complex queries.

•Implicit Ordering Rules: SQL relies on implicit ordering rules that are not visible in its syntactic structure, complicating query composition.

SaneQL: A New Query Language

3.1: Simple and Regular Syntax

•**Improved Learnability:** SaneQL features a straightforward and consistent syntax, improving its learnability and ease of implementation.

•**Extensibility:** SaneQL allows the definition of new operators that integrate seamlessly with the existing built-in ones, enhancing its capabilities.

v3.2: Embracing Core Principles

•**Multiset Semantics**: SaneQL fully embraces the core principles behind SQL, especially multiset semantics, ensuring the enduring success of relational database technology.

- **Modularity**: SaneQL enables the reuse of logic across queries, providing a more flexible and accessible language interface.

HOW IT WORKS

1. Input Receives an Saneql query string.
2. Lexical Analysis : Breaks down the query into individual tokens(keyword, identifier, operator etc.)
3. Syntactic Analysis : Checks if the token sequence adheres to Saneql grammar rules.
4. Semantic Analysis : Verifies if the tokens and their relationships makes sense within the context of Saneql.
5. Output: Produces a structured representation of the query , often as a parse tree or abstract syntax tree. This output can be used for further processing or query execution.

SANEQL PARSE PHASE

±SANEQL Parser overview

±Language: ANTLR (Another tool for language Recognition)

±Grammar: Saneql Parser

±Lexer : Saneql Lexer

±Purpose:

1. Parse Saneql queries
2. Validate syntax and semantics
3. Generate a parse tree for further processing

PARSER PROCESS

1.Lexical Analysis (Lexer)

§Tokenization: Breaks the input query into tokens.

§ Ignoring whitespaces and comments: Removes unnecessary characters.

§ Recognizing keywords, identifiers, literals, and symbols: Identifies the basic components of the SaneQL query.

2. Syntax Analysis (Parser)

§ Constructing a parse tree: Builds a hierarchical representation of the query.

§ Validating the query's syntax: Checks if the query adheres to the defined grammar rules

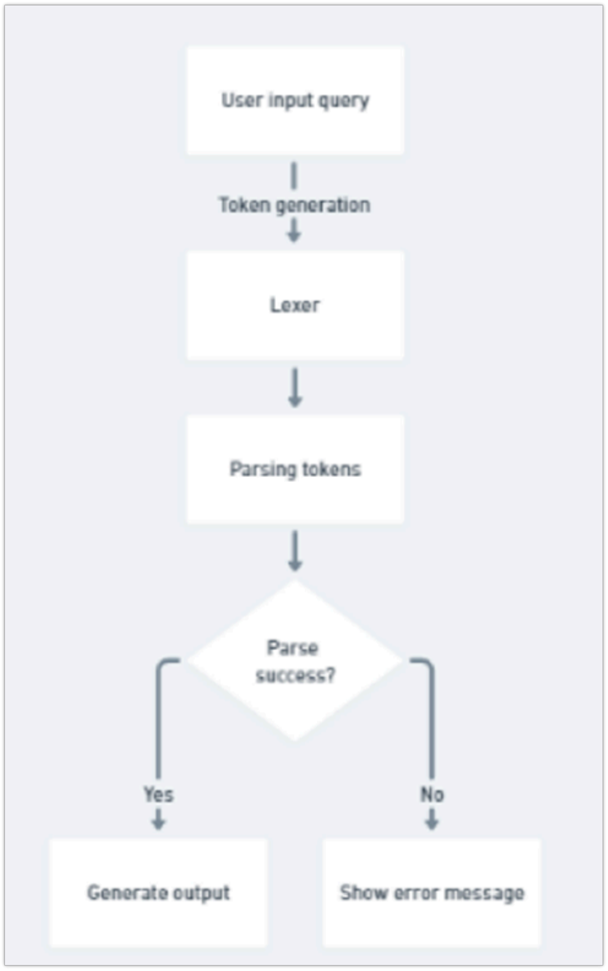
§ Checking for semantic errors: Verifies the meaningfulness of the query.

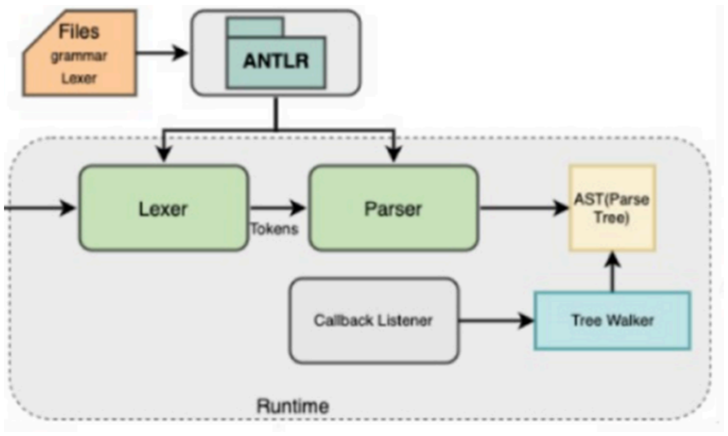
3. Semantic Analysis (Parser)

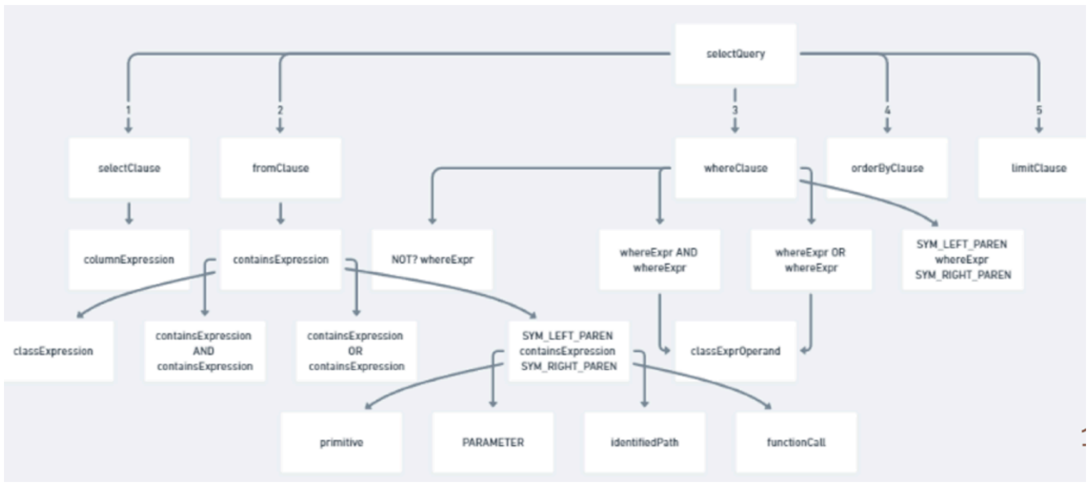
§ Validating the correctness of the query: Ensures the query is logically correct.

§ Checking for undeclared variables: Identifies variables that are not declared

§ Ensuring correct usage of functions and operators: Verifies the proper use of SaneQL functions and operators.


User input query
Token generation
Lexer
Parsing tokens
Parse success?
Yes
No
Generate output
Show error message


Files
grammar
Lexer
ANTLR
Lexer
Tokens
Parser
AST(Parse Tree)
Callback Listener
Tree Walker
Runtime



IMPLEMENTATION DETAILS

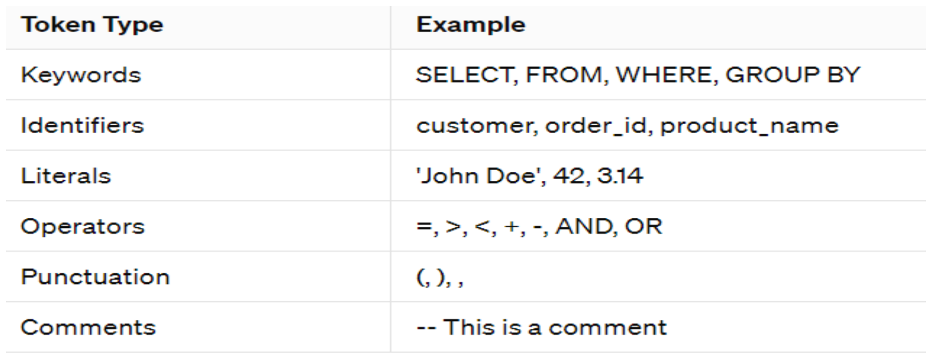
Tokens used -

Keywords: Reserved words in the language that have special meanings and cannot be used as identifiers.
Examples include "if," "else," "while," "for," etc.

Identifiers: Names used to identify variables, functions, classes, or other entities in the program. These are typically user-defined and can consist of letters, digits, and underscores.

Literals: Constants representing fixed values such as numbers, strings, or boolean values.
For example, "42”is a numeric literal, and "'hello'" is a string literal.

Operators: Symbols used to perform operations on operands. Examples include arithmetic operators (+, -, *,/), assignment operators (=), comparison operators (==, !=, <, >), logical operators (&&, ||), etc.



Token Type	Example
Keywords	SELECT, FROM, WHERE, GROUP BY
Identifiers	customer, order_id, product_name
Literals	'John Doe', 42, 3.14
Operators	=, >, <, +, -, AND, OR
Punctuation	(,), ,
Comments	-- This is a comment

- Grammar Development:
    - Lexer Grammar Definition: Define a lexer grammar file to establish the lexical structure of SaneQL, specifying tokens like keywords, identifiers, literals, and operators.
    - Parser Grammar Specification: Craft a parser grammar file to articulate the syntax rules of SaneQL, defining the hierarchical structure encompassing expressions, clauses, and statements.
    -
- ANTLR4 Processing:
    - Input to ANTLR4: Provide the lexer and parser grammar files as input to ANTLR4, ensuring meticulous definition to capture lexical and syntactic intricacies of SaneQL.
    - Code Generation: Utilize ANTLR4 to automatically generate lexer and parser code in a target programming language. Java can be chosen for its versatility and widespread adoption.
    -
- Why and What is ANTLR4:

  - Definition of Grammar: Employ ANTLR4 to define grammar rules specifying the syntax of SaneQL.
  - Code Transformation: ANTLR4 seamlessly transforms grammar specifications into executable lexer and parser code, reducing manual effort and potential errors.
  - Features and Capabilities: Leverage ANTLR4's features like ease of use, powerful parsing algorithms (such as LL(*) parsing), target language support, error handling, and incremental parsing.
  - Reference to ANTLR4 Tool: Utilize the official ANTLR4 tool (https://www.antlr.org/) for grammar processing, code generation, and language recognition tasks.

- Input of Sample SaneQL Query:
  - Utilize the lexer and parser code generated by ANTLR4 to enable effective parsing of SaneQL queries.
  - Input a representative sample SaneQL query into the system to validate the parsing mechanism's functionality.

- Parse Tree Generation:
  - The parser component processes the SaneQL query using lexer-generated tokens, constructing a parse tree.
  - This parse tree serves as a structured representation of the query's syntactic structure, capturing hierarchical relationships between query components.

- Parse Tree Analysis:
  - Structured Query Representation: The generated parse tree provides a structured view of the SaneQL query, facilitating systematic analysis and manipulation.
  - Programmatic Analysis: Leveraging the parse tree, apply programmatic analysis techniques to extract relevant query information.

- Programmatic Analysis:

- Syntax Validation: Validate the syntax of the parsed SaneQL query to ensure adherence to grammar rules.
- Information Extraction: Extract query components programmatically for further processing or transformation.

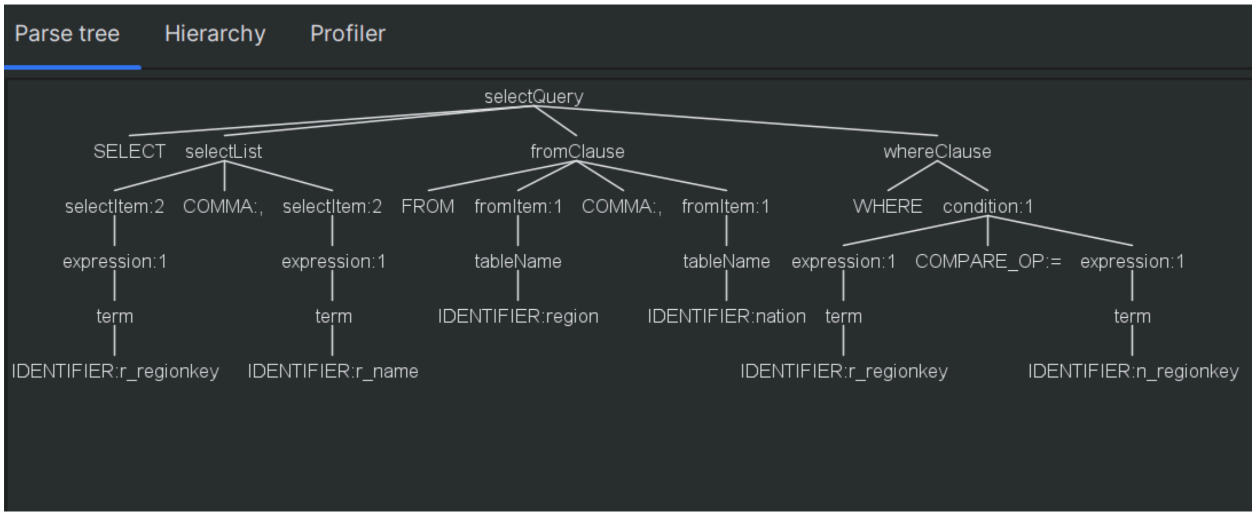
QUERIES-

The grammar includes elements typical of SANEQL syntax, such as SELECT queries with various clauses like FROM, WHERE, GROUP BY, HAVING, and ORDER BY. It also handles functions, aggregate functions, and window functions, among other SANEQL constructs.

1. Problem Statement- The challenge is to effectively express the join condition between the region and nation tables using implicit cross products, considering the normalized structure of the tables and the relational model's emphasis on joins for data retrieval.

QUERY-
SELECT r_regionkey, r_name
FROM region, nation
WHERE r_regionkey = n_regionkey

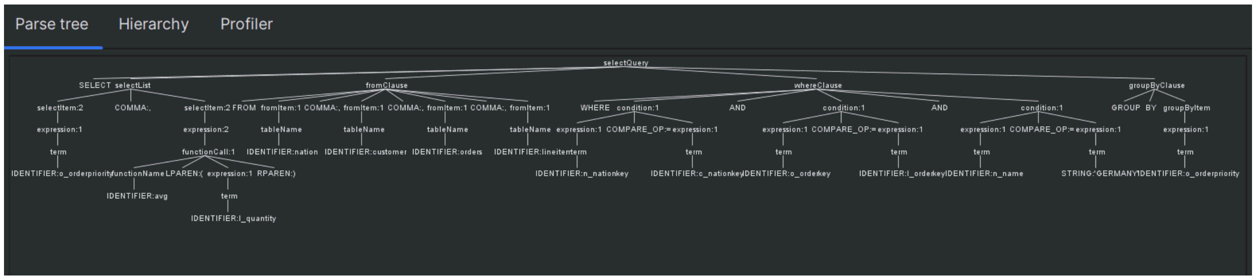
PARSER-



2. Problem Statement-Ensuring that the query retrieves the o_orderpriority and the average of l_quantity for orders from customers in Germany (n_name = 'GERMANY').

QUERY-
```
SELECT o_orderpriority, avg(l_quantity)
FROM nation, customer, orders, lineitem
WHERE n_nationkey = c_nationkey
AND o_orderkey = l_orderkey
AND n_name = 'GERMANY'
GROUP BY o_orderpriority
```

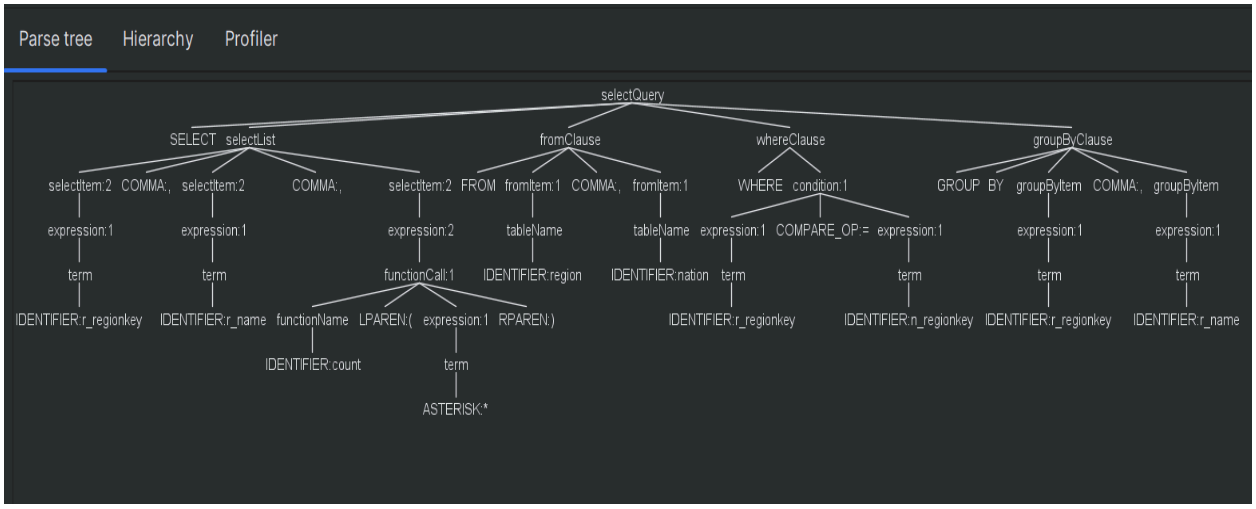
PARSER-



3. Problem Statement- The challenge lies in translating the group aggregation operation (count(*) in this case) while maintaining the join condition and groupings based on the r_regionkey and r_name columns in the context of SQL syntax, particularly with implicit cross products.

QUERY-
```
SELECT r_regionkey, r_name, count(*)
FROM region, nation
WHERE r_regionkey = n_regionkey
GROUP BY r_regionkey, r_name
```

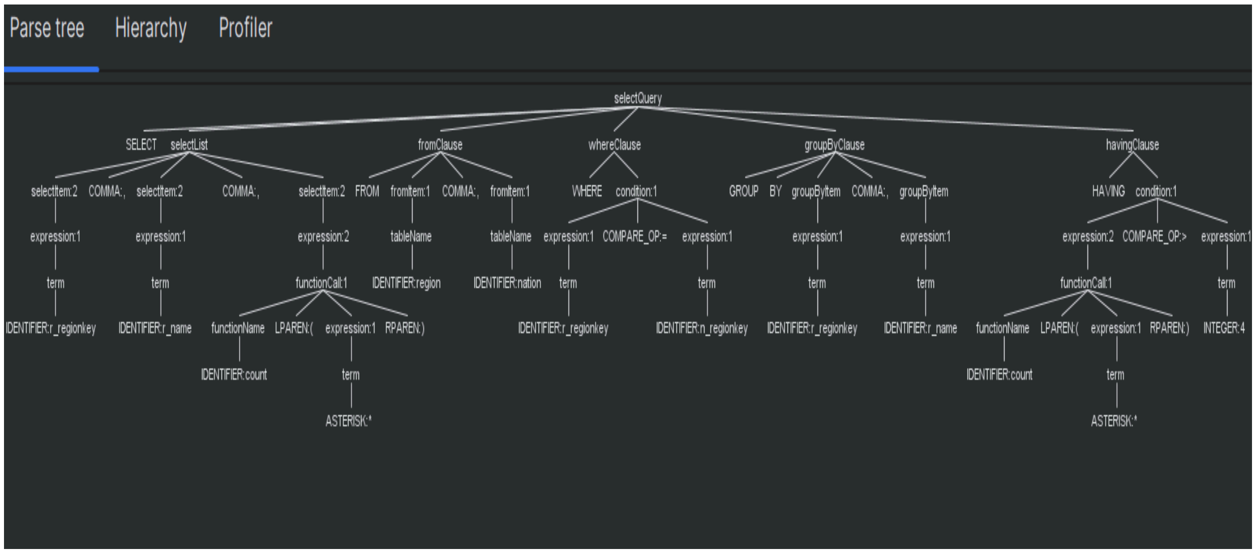
PARSER-



4. Problem Statement- Group aggregation operation (count(*) in this case) while maintaining the join condition and groupings

QUERY-

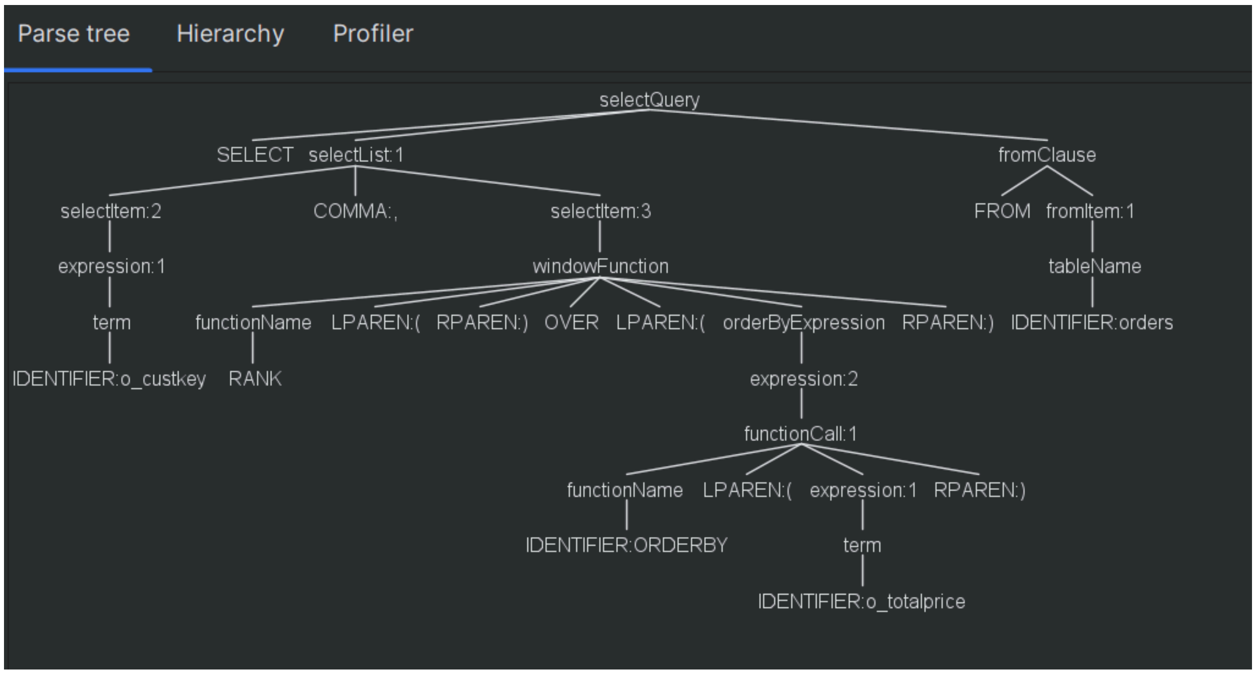
SELECT r_regionkey, r_name, count(*)
FROM region, nation
WHERE r_regionkey = n_regionkey
GROUP BY r_regionkey, r_name
HAVING count(*) > 4
PARSER-



5. Problem Statement- SQL relies on the following syntax in case of window function

QUERY-
SELECT o_custkey, rank() OVER (ORDERBY (o_totalprice))
FROM orders

PARSER-

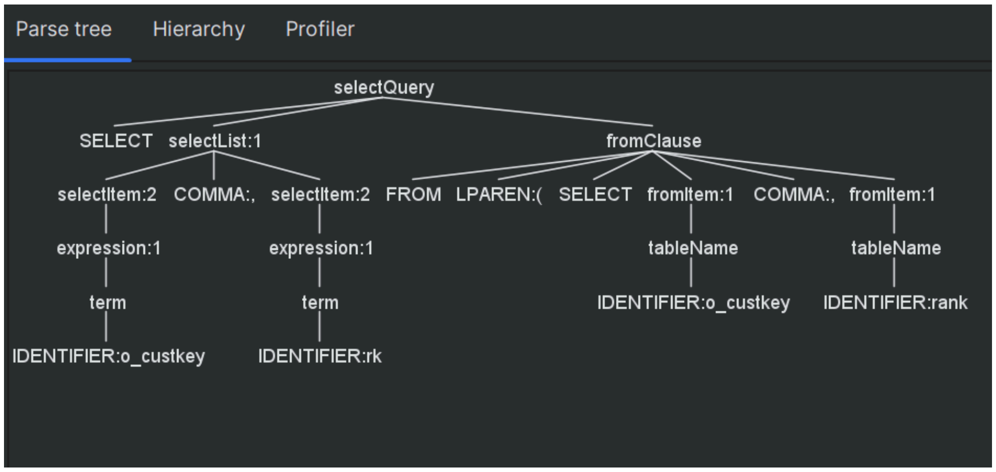


Problem Statement- It is not possible to access window functions in the WHERE clause. Thus, a common operation such as filtering based on a rank of the query.

QUERY-
SELECT o_custkey, rk
FROM (SELECT o_custkey,

rank() OVER (ORDER BY o_totalprice) rk
FROM orders)
WHERE rk < 4

PARSER-



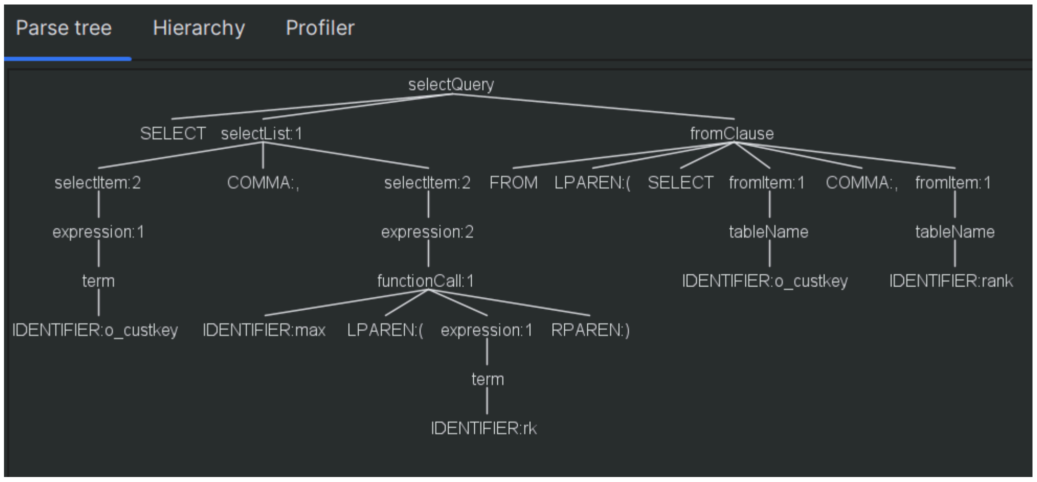
6. Problem Statement- window functions are executed after aggregation is that while it is possible to access aggregates in window functions (e.g., rank() OVER(ORDER BY count(*))), applying an aggregate on a window function again requires a subquery:

QUERY-

```
SELECT o_custkey, max(rk)
FROM (SELECT o_custkey,

rank() OVER (ORDER BY o_totalprice) rk
FROM orders)
GROUP BY o_custkey
```

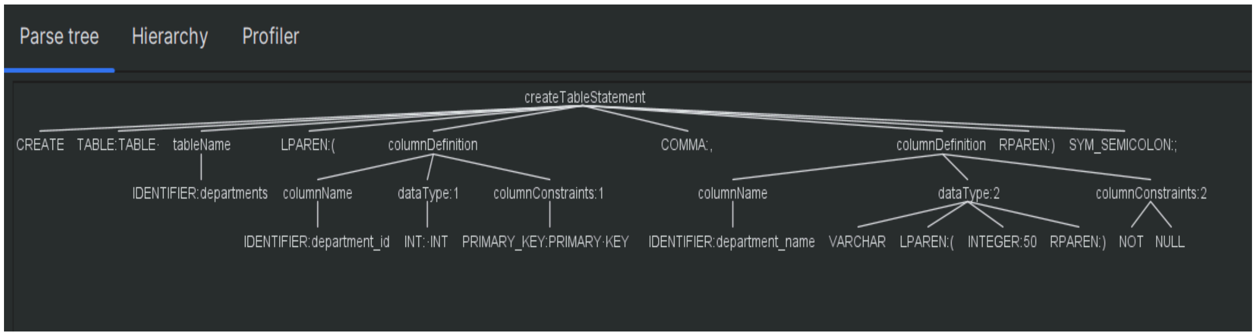
PARSER-



7. Problem Statement- To use create in the saneql syntax:

   DDL Statements - CREATE TABLE:

```
CREATE TABLE departments (
  department_id INT PRIMARY KEY,
  department_name VARCHAR(50) NOT NULL
);
```



Conclusion

In conclusion, the development and implementation of the SaneQL processing system using ANTLR4 have shown promising outcomes. The successful generation of lexer and parser components, along with the parsing of sample SaneQL queries, validates the effectiveness of the methodology in enabling SaneQL query parsing.

The structured representation provided by the parse tree facilitates systematic analysis and manipulation of parsed SaneQL queries, enhancing the system's usability across various scenarios. Through programmatic analysis techniques, users can extract essential information, validate syntax, and adapt queries to meet specific requirements.

Overall, the SaneQL processing system demonstrates considerable potential in supporting efficient data querying and retrieval tasks. Moving forward, continuous refinement of the system and exploration of advanced parsing techniques could further enhance its capabilities and broaden its applicability across diverse domains, ensuring its relevance and effectiveness in various application contexts.