Report

Addressing the Challenges of SQL with SaneQL

Introduction:-

SQL, while widely used for querying databases, presents several challenges ranging from irregular syntax to lack of extensibility and portability. These challenges hinder its usability and lead to the exploration of alternative query languages. SaneQL emerges as a solution, offering improved query expression, modularity, and extensibility. This report delves into the problems faced with SQL and how SaneQL addresses them, focusing on its features, compiler phases, and benefits.

Problems with SQL:-

1. Syntax Irregularities: SQL's irregular syntax leads to unhelpful error messages, hindering readability and causing confusion during query composition.

2. Learning Difficulty: SQL poses significant challenges for learners due to its complexity and irregularities, resulting in a high error rate.

3. Joins and Aggregation: Issues with SQL's join syntax and aggregation mechanisms often surprise users and lead to errors.

4. Window Functions: The usage of window functions introduces complexities in SQL queries, challenging the declarative nature of the language.

5. Lack of Portability: Despite being an ISO standard, SQL lacks true portability across systems due to divergences in implementations.

Features of SaneQL:

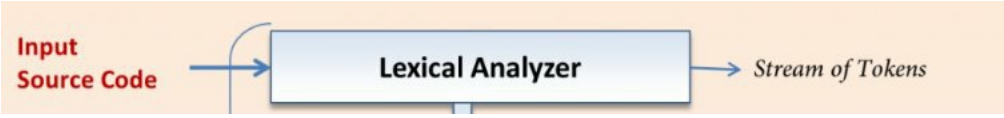
1. Improved Query Expression: SaneQL offers a more systematic and intuitive approach to expressing queries, addressing SQL's awkward and complex structures.

2. Modularity and Extensibility: SaneQL supports logic reuse through parameterized queries, enabling modularity and efficient sharing of logic across multiple queries.

Compiler Phases of SaneQL:-

1. Lexical Analysis:

->Objective: The lexical analysis phase is the first step in the compilation process of SaneQL. Its main objective is to tokenize the input source code, breaking it down into smaller units called tokens.



RULES :

1)Tokens like "select", "from", "where", etc. are specified directly as strings:

```
"select"    { printf("Found SELECT statement\n"); }
"from"      { printf("Found FROM clause\n"); }
"where"     { printf("Found WHERE clause\n"); }
```

2) Identifiers (words starting with a letter followed by zero or more letters or digits) are matched using [[:alpha:]][[:alnum:]]* pattern

```
[[:alpha:]][[:alnum:]]*    { printf("Found identifier: %s\n", yytext); }
```

3) Numbers are matched using [[:digit:]]+ pattern:

```
[[:digit:]]+    { printf("Found number: %s\n", yytext); }
```

4)Comments starting with // and ending with a newline character \n are matched using "//"(.)"\n" pattern

```
"//"(.)"\n"    / Comment handling, ignores everything after '//' till the end of line */
```

5)Other characters are ignored:

```
.|\n            /* Ignore other characters */
```

->Process: During lexical analysis, the input source code is scanned character by character. Each character or group of characters that form a meaningful unit, such as keywords, identifiers, literals, and operators, is recognized and classified into corresponding token types.

->Output: The output of this phase is a stream of tokens, where each token represents a fundamental building block of the SaneQL language, such as keywords (e.g., SELECT, FROM), identifiers (e.g., table names, column names), literals (e.g., numeric constants, string literals), and operators (e.g., arithmetic operators, comparison operators).

2. Parsing:-

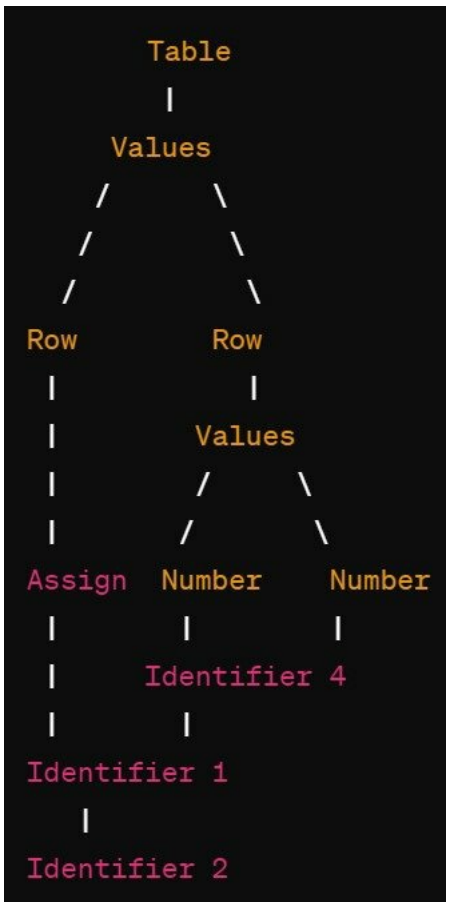
Objective: The parsing phase of the SaneQL compiler takes the stream of tokens generated by the lexical analysis phase and constructs a hierarchical structure known as the Abstract Syntax Tree (AST).

Input :

Tokens identified by the lexer : TABLE ,LPAREN, LBRACE,

IDENTIFIER (a) ,COLON_EQ NUMBER (1) ,COMMA ,IDENTIFIER (b), COLON_EQ, NUMBER (2), COMMA ,LBRACE ,NUMBER (3) ,COMMA, NUMBER (4) ,RBRACE, RPAREN

Output:



Production Rules used in the Program are :

```
 /* Program */
program: statement_list          { root = $1; }
       ;

 /* Stament List  */
statement_list: statement
              | statement_list statement
              ;
```

```
 /* Stament  */
statement: select_statement       { $$ = $1; }
         | insert_statement       { $$ = $1; }
         | update_statement       { $$ = $1; }
         | delete_statement       { $$ = $1; }
         | create_statement       { $$ = $1; }
         | alter_statement        { $$ = $1; }
         | drop_statement         { $$ = $1; }
         | truncate_statement     { $$ = $1; }
         | join_statement         { $$ = $1; }
         ;

/*  Select Stament */
select_statement: SELECT column_list FROM table_name ';' { $$ = create_node(SELECT, "", $2, NULL); }
                ;

 /* Coloumn List */
column_list: IDENTIFIER
           | column_list ',' IDENTIFIER
           ;
```

```
/* Table Name */
table_name: IDENTIFIER
          ;

/* Insert Statement */
insert_statement: INSERT INTO table_name VALUES '(' value_list ')' ';' { $$ = create_node(INSERT, "", $3, $6); }
                ;

/* Value List */
value_list: value
          | value_list ',' value
          ;

/* Value */
value: NUMBER
     | STRING
     ;

/* Update Statement */
update_statement: UPDATE table_name SET column_name '=' value WHERE condition ';' { $$ = create_node(UPDATE, "", $2,
                ;
```

```
/* Coloumn Name */
column_name: IDENTIFIER
           ;

/* Condition */
condition: expression
         ;

/* Expression */
expression: IDENTIFIER '=' value   { $$ = create_node(EQUAL, "=", create_node(IDENTIFIER, $1, NULL, NULL), create_nod
          | IDENTIFIER '>' value   { $$ = create_node(GREATER_THAN, ">", create_node(IDENTIFIER, $1, NULL, NULL), cre
          | IDENTIFIER '<' value   { $$ = create_node(LESS_THAN, "<", create_node(IDENTIFIER, $1, NULL, NULL), create
          | '(' expression ')'     { $$ = $2; }
          | expression AND expression { $$ = create_node(AND, "AND", $1, $3); }
          | expression OR expression { $$ = create_node(OR, "OR", $1, $3); }
          ;

/* Delete Statement */
delete_statement: DELETE FROM table_name WHERE condition ';' { $$ = create_node(DELETE, "", $3, $5); }
                ;
```

```
/* Create Statement */
create_statement: CREATE TABLE IDENTIFIER '(' column_def_list ')' ';' { $$ = create_node(CREATE_TABLE
                | ;
/* Coloumn Definition List */
column_def_list: column_def
               | column_def_list ',' column_def
               ;
/* Coloumn Definition */
column_def: IDENTIFIER
          ;
/* Alter Statement */
alter_statement: ALTER TABLE IDENTIFIER ADD IDENTIFIER ';' { $$ = create_node(ALTER_TABLE, "", $3, cr
               ;
/* Drop Statement */
drop_statement: DROP TABLE IDENTIFIER ';' { $$ = create_node(DROP_TABLE, "", $3, NULL); }
              ;
/* Truncate Statement */
truncate_statement: TRUNCATE TABLE IDENTIFIER ';' { $$ = create_node(TRUNCATE_TABLE, "", $3, NULL); }
                  ;
/* Join Statement */
join_statement: JOIN table_name ON condition { $$ = create_node(JOIN, "", $2, $4); }
              ;
```

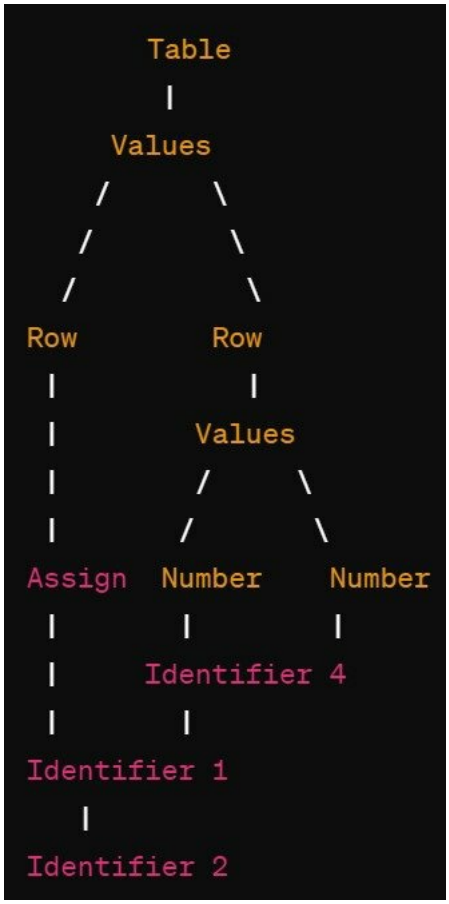
Process: Parsing involves analyzing the sequence of tokens according to the grammar rules of the SaneQL language. By applying parsing algorithms, such as recursive descent parsing or shift-reduce parsing, the compiler builds the AST, representing the syntactic structure of the source code.

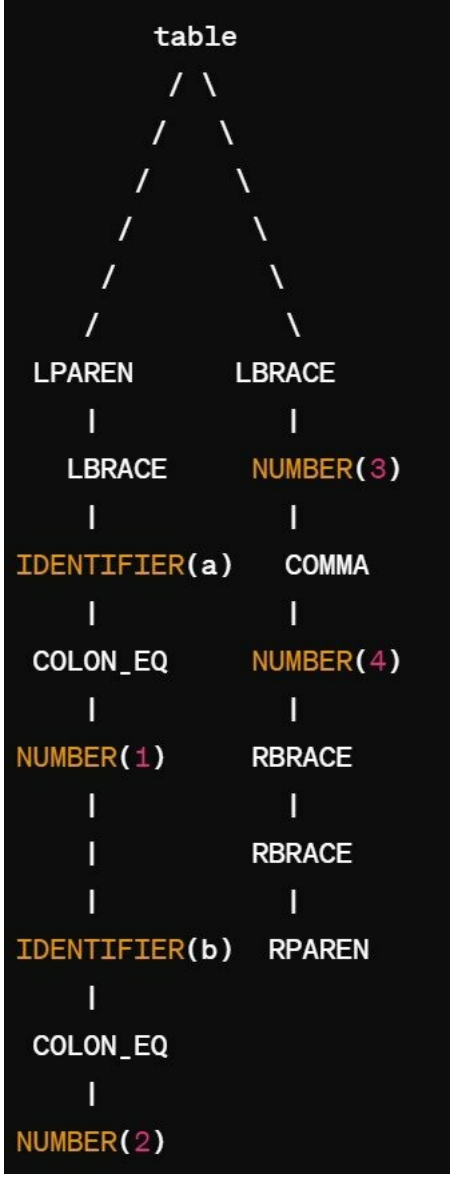
Output: The output of the parsing phase is the AST, where each node in the tree corresponds to a syntactic construct of the SaneQL language, such as expressions, statements, and declarations. The AST serves as an intermediate representation that facilitates subsequent semantic analysis and code generation.

3. Semantic Analysis:

Objective: Semantic analysis is responsible for ensuring the correctness and coherence of the SaneQL program beyond its syntax. It checks for semantic errors, enforces language rules, and collects information about identifiers and their types.

Input: output:





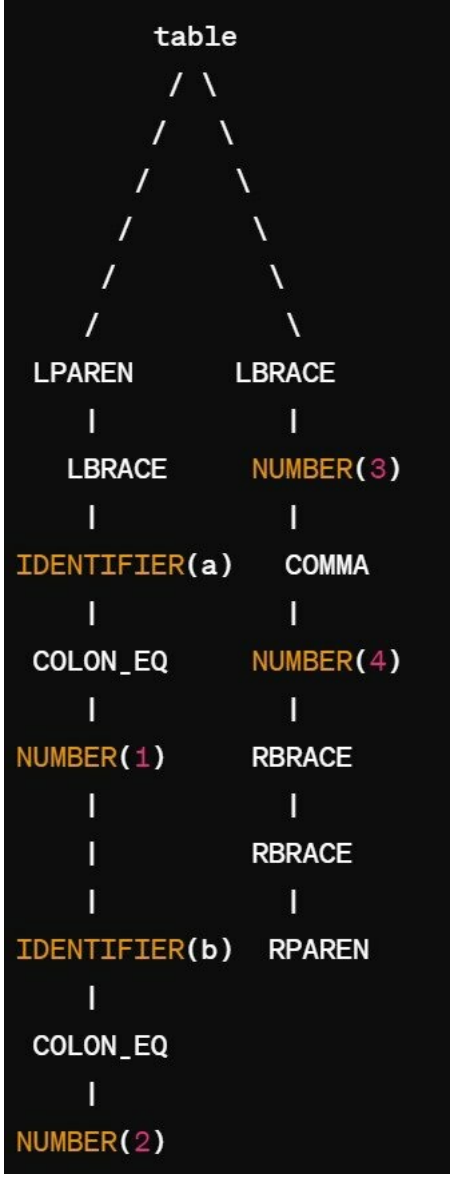
Process: During semantic analysis, the compiler performs various checks, including type checking, scope analysis, and symbol table management. It verifies that variables and functions are used consistently, resolves identifiers to their declarations, and detects any violations of semantic rules defined by the SaneQL language specification.

Output: The output of semantic analysis may include a symbol table, which is a data structure that maps identifiers to their attributes, such as type information and scope. Additionally, semantic analysis may annotate the AST with semantic information, preparing it for intermediate code generation.

4. Intermediate Code Generation:

Objective: The intermediate code generation phase translates the AST into an intermediate representation (IR) of the SaneQL program. The IR serves as a platform-independent abstraction that facilitates further optimizations and code transformations.

Input:



Output:

1. LOAD_CONST 3 // Load the constant value 3

2. STORE_VAR a // Store the value in variable 'a'

3. LOAD_CONST 4 // Load the constant value 4

4. STORE_VAR b // Store the value in variable 'b'

5. LOAD_CONST 1 // Load the constant value 1

6. STORE_VAR _ // Store the value in an unnamed temporary variable

7. LOAD_CONST 2 // Load the constant value 2

8. STORE_VAR _ // Store the value in the same unnamed temporary variable

Process: During intermediate code generation, the compiler traverses the AST and generates intermediate code instructions corresponding to each syntactic construct. These instructions capture the essential operations and control flow of the SaneQL program without targeting any specific hardware architecture.

Output: The output of intermediate code generation is the IR, which may take the form of three-address code, static single assignment (SSA) form, or another intermediate representation suitable for subsequent optimization passes and target code generation.

5. Code Optimization:

Objective: Code optimization aims to improve the efficiency and performance of the SaneQL program by applying various optimization techniques to the intermediate code.

Input:

1. LOAD_CONST 3 // Load the constant value 3

2. STORE_VAR a // Store the value in variable 'a'

3. LOAD_CONST 4 // Load the constant value 4

4. STORE_VAR b // Store the value in variable 'b'

5. LOAD_CONST 1 // Load the constant value 1

6. STORE_VAR _ // Store the value in an unnamed temporary variable

7. LOAD_CONST 2 // Load the constant value 2

8. STORE_VAR _ // Store the value in the same unnamed temporary variable

Output:

1. LOAD_CONST 3 // Load the constant value 3

2. DROP_TOP // Drop the top value from the stack 3. LOAD_CONST 4 // Load the constant value 4

4. DROP_TOP // Drop the top value from the stack 5. LOAD_CONST 1 // Load the constant value 1

6. STORE_VAR _ // Store the value in an unnamed temporary variable

7. LOAD_CONST 2 // Load the constant value 2

8. STORE_VAR _ // Store the value in the same unnamed temporary variable

Process: Code optimization involves analyzing the IR to identify and exploit opportunities for optimization, such as eliminating redundant computations, reducing memory usage, and improving cache locality. Optimization techniques may include constant folding, loop optimization, and inlining of functions.

Output: The output of code optimization is an optimized version of the IR, where unnecessary or inefficient code segments have been replaced or optimized to produce more efficient code while preserving the semantics of the original program.

6. Target Code Generation:-

Objective: The target code generation phase translates the optimized IR into machine-readable code specific to the target hardware platform, such as assembly language or low-level intermediate representation (e.g., LLVM IR).

Input:

1. LOAD_CONST 3 // Load the constant value 3

2. STORE_VAR a // Store the value in variable 'a'

3. LOAD_CONST 4 // Load the constant value 4

4. STORE_VAR b // Store the value in variable 'b'

5. LOAD_CONST 1 // Load the constant value 1

6. STORE_VAR _ // Store the value in an unnamed temporary variable

7. LOAD_CONST 2 // Load the constant value 2

8. STORE_VAR _ // Store the value in the same unnamed temporary variable

Output:

LOAD_CONST 3 // Load the constant value 3 POP // Pop the value from the stack(unused)

LOAD_CONST 4 // Load the constant value 4 POP // Pop the value from the stack (unused)

LOAD_CONST 1 // Load the constant value 1 STORE_TEMP_0 // Store the value in temporary variable 0

LOAD_CONST 2 // Load the constant value 2 STORE_TEMP_0 // Store the value in temporary variable 0

Process: Target code generation involves mapping the optimized IR instructions to the instructions supported by the target architecture, taking into account the hardware constraints and optimization goals. It generates code that efficiently utilizes the resources of the target platform while adhering to the semantics of the SaneQL program.

Output: The output of target code generation is the machine-readable code, which can be directly executed by the target hardware platform. Depending on the target platform, the generated code may be in the form of assembly language instructions, bytecode, or machine code.

Benefits of SaneQL:

1. Simple and Expressive Queries: SaneQL offers a clear improvement over SQL's syntax, making queries more readable and understandable.

2. Modularity and Efficiency: SaneQL's modularity and parameterized queries enhance efficiency and maintainability, enabling logic reuse and sharing.

Conclusion:

SaneQL addresses the challenges of SQL by providing a simple, expressive, and modular query language. Through features like improved query expression, modularity, and compiler phases, SaneQL offers a promising alternative for database querying, catering to both beginners and experienced users alike. Its extensibility and efficiency make it a compelling choice for developing complex and modular database systems, potentially reshaping the landscape of data querying and manipulation.