SQL Query Parser & Validator

1. Overview

This project implements a **token-based SQL query parser and semantic validator**, focused primarily on `SELECT` queries, with an architecture that is designed to be **extensible to other SQL statements** (`INSERT`, `UPDATE`, `DELETE`, `CREATE`, etc.).

The goal of the system is **not to execute SQL**, but to **validate SQL syntax and semantics** in a structured, SQL-faithful way — similar to how a database query planner would reject invalid queries *before execution*.

Key design goals: - Clear separation of concerns (tokenization, clause parsing, semantic checks) - SQL-standard–aware behavior - Early, precise error reporting - Extensibility for future SQL features and dialects

---

2. High-Level Architecture

The system is divided into three major layers:

1. **Query-level parsing**
   Handles tokenization, semicolon validation, and routing to the correct statement parser.

2. **Statement-level parsing (SELECT)**
   Coordinates validation of each SQL clause in the correct order.

3. **Clause-specific helpers**
   Each SQL clause (`SELECT`, `FROM`, `WHERE`, `GROUP BY`, `HAVING`, `ORDER BY`, `LIMIT`) has its own helper module responsible for:

4. Syntax validation
5. Semantic validation
6. SQL-specific rules

This layered approach ensures that logic remains modular, testable, and easy to reason about.

---

3. File & Module Structure

```
select/
├── clauseChecksHelper.py
├── selectChecksHelper.py
├── fromChecksHelper.py
├── whereChecksHelper.py
```

```
├── groupByChecksHelper.py
├── havingChecksHelper.py
├── orderByHelpers.py
├── selectParser.py
├── utils.py

queryParser.py
```

Responsibilities by module

File	Responsibility
`queryParser.py`	Entry point, tokenization, query routing
`selectParser.py`	Coordinates full SELECT validation
`clauseChecksHelper.py`	Clause order, duplicates, mandatory clauses
`selectChecksHelper.py`	SELECT list parsing & validation
`fromChecksHelper.py`	FROM tables, aliases, JOIN validation
`whereChecksHelper.py`	WHERE boolean & expression validation
`groupByChecksHelper.py`	GROUP BY semantics
`havingChecksHelper.py`	HAVING semantics
`orderByHelpers.py`	ORDER BY resolution
`utils.py`	Shared token utilities

---

4. Tokenization Strategy

Tokenization is handled by `QueryParser.tokenize()`.

Design choices

- All identifiers are lowercased for normalization
- Multi-character operators (`<=`, `!=`, `>=`) are grouped
- SQL symbols are emitted as standalone tokens
- Whitespace is ignored

Example:

```sql
SELECT a, b FROM table1 WHERE x >= 10;
```

Becomes:

```
['select', 'a', ',', 'b', 'from', 'table1', 'where', 'x', '>=', '10']
```

This simplified token model makes clause-level parsing deterministic and easy to reason about.

---

5. Query Routing

After tokenization, `QueryParser.analyse()`: 1. Validates semicolon usage 2. Extracts the query type (`select`, `insert`, etc.) 3. Routes the query to the appropriate statement parser

Currently, `SELECT` **is fully implemented**, while other statement types are stubbed for future extension.

---

6. SELECT Parser – Core Orchestration

The heart of the system is `SelectParser`, which acts as a **semantic pipeline**.

Validation phases (in order):

1. **Clause-level checks**
2. Duplicate clauses
3. Missing mandatory clauses (`SELECT`, `FROM`)
4. Clause ordering
5. **SELECT validation**
6. Expression ordering
7. `*` usage rules
8. Aggregate syntax
9. Alias extraction
10. Qualified column collection
11. **FROM validation**
12. Table references
13. Alias uniqueness
14. JOIN validation
15. **Alias resolution**

16. Ensures `table.column` references refer to known tables

17. **WHERE validation**

18. **GROUP BY validation**
19. **HAVING validation**
20. **ORDER BY validation**
21. **LIMIT validation**

Each phase stops execution immediately on error, ensuring **early and precise feedback**.

---

7. Clause-Specific Design Details

This section explains **how each SQL clause is parsed and validated**, the **rules enforced**, and the **techniques used** internally.

7.1 SELECT Clause

Purpose: Defines which expressions are returned by the query.

How it works: - Tokens between `SELECT` and `FROM` are extracted. - The SELECT list is parsed using a **state machine** to ensure correct ordering of expressions, aliases, and commas. - Parenthesis depth is tracked so nested expressions don't interfere with top-level parsing.

Rules enforced: 1. SELECT list cannot be empty. 2. SELECT cannot start or end with a comma. 3. `*` can only appear alone (cannot be mixed with other columns). 4. Aggregate functions must use parentheses. 5. Nested aggregate functions are not allowed. 6. Column names cannot be SQL keywords. 7. Aliases: - Explicit (`AS alias`) and implicit (`expr alias`) aliases are supported. - Only one alias per expression is allowed.

Techniques used: - State-machine parsing - Parenthesis depth tracking - Expression-level validation delegation

Handled by `selectChecksHelper.py`.

Key techniques: - State-machine–based parsing (`EXPECT_COLUMN`, `EXPECT_ALIAS_OR_COMMA`, etc.) - Explicit handling of aggregates - Prevention of nested aggregates - SQL-faithful alias rules - Top-level comma tracking using parenthesis depth

This avoids building a full AST while still enforcing strong correctness.

7.2 FROM & JOIN Handling

Purpose: Defines the data sources and relationships between tables.

How it works: - Tokens after `FROM` are extracted until another clause begins. - The parser detects whether comma joins or explicit JOIN syntax is used. - JOIN chains are validated incrementally, tracking alias scope.

Rules enforced: 1. FROM clause must contain at least one table. 2. Table names must be valid identifiers. 3. Table aliases must be unique. 4. Cannot mix comma joins with `JOIN ... ON` syntax. 5. Every JOIN must have a valid ON condition. 6. ON conditions may only reference tables that are already introduced.

Techniques used: - Incremental alias scope tracking - Join-chain parsing - Boolean expression reuse for ON validation

Handled by `fromChecksHelper.py`.

Key rules enforced: - No mixing of comma joins and explicit `JOIN ... ON` - Strict alias uniqueness - Validation of `ON` conditions - Alias scope tracking for JOIN chains

JOIN validation mimics real SQL engines by enforcing alias visibility incrementally.

7.3 WHERE Clause

Purpose: Filters rows before grouping and aggregation.

How it works: - WHERE tokens are validated recursively as boolean expressions. - Expressions are split on top-level `AND` / `OR`. - Comparisons and arithmetic expressions are validated at leaf nodes.

Rules enforced: 1. Parentheses must be balanced. 2. Boolean operators require operands on both sides. 3. Comparisons must have exactly one comparison operator. 4. Aggregates are NOT allowed in WHERE. 5. Arithmetic expressions must alternate operands and operators.

Techniques used: - Recursive descent validation - Top-level operator splitting using depth counters

Handled by `whereChecksHelper.py`.

Approach: - Recursive descent validation - Top-level operator splitting using parenthesis depth - Strict distinction between: - Boolean operators (`AND`, `OR`) - Comparison operators (`=`, `>`, etc.) - Arithmetic expressions

Aggregates are **explicitly disallowed** in WHERE.

7.4 GROUP BY Clause

Purpose: Defines grouping keys for aggregation.

How it works: - GROUP BY expressions are split at top-level commas. - SELECT expressions are normalized and compared against GROUP BY expressions.

Rules enforced: 1. GROUP BY expressions cannot contain aggregates. 2. Every non-aggregated SELECT expression must appear in GROUP BY. 3. Expressions must match exactly after normalization.

Techniques used: - Expression normalization - Set comparison

Handled by `groupByChecksHelper.py`.

Semantic rules enforced: - No aggregates allowed in GROUP BY - Every non-aggregated SELECT expression must appear in GROUP BY - Expressions must match *exactly* (after normalization)

This implements **strict SQL GROUP BY semantics**.

7.5 HAVING Clause

Purpose: Filters groups after aggregation.

How it works: - HAVING expressions are recursively validated similar to WHERE. - Special rules are applied for aggregates and grouped expressions.

Rules enforced: 1. HAVING cannot be empty. 2. Aggregates are allowed. 3. Non-aggregated columns must be grouped or aliased. 4. Boolean expressions must be valid.

Techniques used: - Recursive boolean validation - Alias and GROUP BY set resolution

Handled by `havingChecksHelper.py`.

HAVING validation allows: - Aggregate expressions - Grouped columns - SELECT aliases

But disallows: - Ungrouped, non-aggregated columns

Validation is recursive and boolean-aware.

7.6 ORDER BY Clause

Purpose: Defines result ordering.

How it works: - ORDER BY items are split at top-level commas. - Each expression is resolved against known aliases and expressions.

Rules enforced: 1. ORDER must be followed by BY. 2. ORDER BY cannot be empty. 3. ORDER direction must be last (`ASC` / `DESC`). 4. ORDER BY expressions may reference: - SELECT aliases - SELECT expressions - GROUP BY expressions 5. Aggregates must be valid in context.

Techniques used: - Expression normalization - Resolution priority strategy

Handled by `orderByHelpers.py`.

Resolution priority: 1. SELECT aliases 2. SELECT expressions 3. GROUP BY expressions 4. Any valid expression

Aggregates in ORDER BY are restricted unless legally defined.

7.7 LIMIT Clause

Purpose: Limits the number of rows returned.

How it works: - LIMIT tokens are extracted and validated directly.

Rules enforced: 1. LIMIT must contain exactly one value. 2. Value must be a non-negative integer literal.

Techniques used: - Literal validation

Handled in `SelectParser` using shared utilities.

Rules enforced: - LIMIT must contain exactly one value - Value must be a non-negative integer

---

8. Key Techniques Used

- **Token-based parsing** instead of full ASTs
- **Parenthesis depth tracking** for correctness
- **Normalization-based expression comparison**
- **Incremental alias scope resolution**
- **Early-fail validation pipeline**
- **SQL-standard–inspired semantics**

This results in a parser that is strict, predictable, and easy to extend.

---

9. Extensibility & Future Work

The architecture intentionally supports: - Additional SQL statements (`INSERT`, `UPDATE`, `DELETE`) - Subqueries - Window functions - SQL dialect switches (Postgres / MySQL / SQL Server) - AST generation for query planning

---

10. Summary

This project implements a **clean, modular, SQL-faithful SELECT query validator**.

Rather than relying on heavy grammar tooling, it uses **carefully designed token analysis, recursive validation, and semantic checks** to achieve correctness, clarity, and extensibility.

The result is a system that collaborators can easily understand, debug, and extend.

Author's intent: Build a readable, maintainable SQL validation engine that mirrors how real databases think about queries — not just how they look syntactically.