Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Automated Testing of Database Engines

Project report

Elias Achermann / Luca Schnyder

May 15, 2025

Department of Computer Science, ETH Zürich

Contents

Chapter 1

Implementation

In this chapter, we discuss the implementation details of the fuzzer and the SQL statement generator. It provides a technical description of both tools, including the design choices we made and their limitations.

1.1 Fuzzer

1.1.1 Architecture Overview

The source code of the fuzzer is located in fuzzer.py. The Fuzzer uses the SQL statement generator described below to create syntactically valid, diverse SQL statements. It uses differential testing, executing the different SQL statements against different SQLite versions. The fuzzer uses a multiprocessing approach for performance optimization. Since our SQL statement generator is designed to only produce valid SQL statements there is also a random mutator included for Error testing.

1.1.2 Design choices

Differential Testing Approach

The core design uses differential testing using a newer version of SQLite as an oracle. This avoids the need to define exact correct behavior and works effectively with complex queries where output is difficult to predict.

Temporary Database Files

We decided to use temporary database files for each test. This prevents cross-test contamination making it easier to identify the bug and at the same time enables easy parallel testing.

Error Reporting

Each difference found by the fuzzer is saved in a separate file containing the SQL statement and the output as well as the produced errors of both versions.

Parallel Exection Model

We use parallelism with the help of Python's multiprocessing module. Each process handles test cases independently. This significantly improves the throughput of the fuzzer.

Timeout Protection

Each test has a 5-second limit to prevent resource exhaustion. We consider 5 seconds as a reasonable threshold because a usual query gets executed much faster and a higher limit reduces the performance of the fuzzer too much. With this protection mechanism, it is possible to identify infinite loops or extremely slow queries with a reasonable trade-off. A time-out will also produce an Error report.

1.1.3 Inputs

The fuzzer takes the following arguments as input:

- `--iterations`: number of SQL statements to produce and test
- `--processes`: number of parallel processes
- `--old-sqlite`: path to the SQLite version to test
- `--new-sqlite`: path to the SQLite version used as an oracle

1.1.4 Implementation Details

First, the fuzzer creates with the help of the Python multiprocessing package `--iterations` number of tasks that are processed by `--processes` number of processes. Each task consists of first creating an SQL statement with the SQL statement generator and then creating two temporary database files. Then execute the same SQL statement on both versions `--old-sqlite` and `--new-sqlite` in a subprocess. Each process gets a timeout of 5 seconds to prevent infinite loops or extremely slow queries.

After the execution, the output is compared. It detects differences in standard output between versions and identifies cases where one version produces an error while another succeeds.

If a difference is detected or an error occurs, a log gets stored including the SQL statement, the output, and errors of both versions.

At the end for Error testing, there is a 10% probability that the SQL statement gets randomly mutated, executed, and compared again.

1.1.5 Limitations

Since we used the differential testing approach it suffers from the standard limitations of differential testing. It cannot detect issues if both versions have the same incorrect behavior, it may report intended behavior changes as differences and doesn't explicitly verify SQL standard compliance.

1.2 SQL statement generator

The SQL generator is located in the file sql_statement_generator.py. It is designed to generate syntactically valid SQL statements which then are used as input to the fuzzer. The SQL statements range from simple queries to complex statements involving joins, window functions, subqueries, CTEs, JSON operations, and transactions.

1.2.1 Architecture Overview

The generator is built around a class-based architecture with the SQLiteGrammar class serving as the core component. The first design of the SQLiteGrammar class, including the keywords used, was inspired by ANTLR (https://github.com/antlr/antlr4). From then on, the class has been extended. The class keeps an internal state about the generated database objects using a tables_info structure that tracks table schemas, column definitions, and constraints. This state allows for the generation of valid SQL statements that reference existing objects without generating errors. The SQLiteGrammar class contains methods for generating different SQL statement types, helper functions that provide random strings, values, and identifiers needed for the SQL statements. Finally, there is a Test Case generator that is used for the creation of complete SQL statements using the methods and helper functions.

1.2.2 Design choices

We initially built a randomized approach in which the query generator selected SQL clauses at random to construct queries. However, this led to a high number of syntactically or semantically invalid queries, which were quickly rejected by both SQLite versions, resulting in no bugs being discovered after a long testing time. To improve the effectiveness of our fuzzing process, we transitioned to an SQL statement generator that uses a grammar-based approach, where we divide SQL into components that are generated separately with controlled randomness. This way we can easily

produce syntactically valid statements while also ensuring high diversity with the possibility to extend the generator with more components. The expression depth can be customized with a variable. We decided to use an expression depth of 4 to be able to create complex enough statements to produce bugs but still not too large such that the execution takes too long. Since we test versions v3.26.0 and v3.39.4 and we decided to use v3.49.1 as an oracle for both, the SQL statement generator only produces SQL statements with components that are available since v3.26.0.

1.2.3 Detailed technical Description of Methods

generate_create_table_stmt()

Generates `CREATE TABLE` statements with randomized structure:

- Creates random table names with prefix `t_` followed by 3-10 random alphanumeric characters
- Generates between 1-10 columns with randomly selected data types from SQLite types (INTEGER, TEXT, REAL, NUMERIC)
- Randomly adds constraints: PRIMARY KEY, DEFAULT values, COLLATE specifications
- Supports table-level constraints: PRIMARY KEY, UNIQUE, and FOREIGN KEY
- Implements WITHOUT ROWID optimization with 10% probability (only when a PRIMARY KEY is defined)
- Returns a tuple containing table metadata with info `tables_info` about the table and its specification/constraints

generate_alter_table_stmt()

Generates `ALTER TABLE` statements with three operation types and maintains schema context:

- `RENAME TABLE`: Changes a table's name while preserving all column information
- `RENAME COLUMN`: Renames an existing column while preserving its type and constraints
- `ADD COLUMN`: Adds a new column with random type and constraints
- Returns both the updated `tables_info` dictionary to maintain schema state

generate_create_index_stmt()

Creates indexes on tables with various options:

- Generates UNIQUE indexes with 20% probability
- Supports multi-column indexes with up to 3 columns
- Includes functional indexes using UPPER, LOWER, ABS, or LENGTH (20% probability)
- Adds collation specifications (BINARY, NOCASE, RTRIM) with 20% probability
- Sets sort orders (ASC/DESC) with 50% probability per column
- Adds WHERE clauses for partial indexes with 30% probability

generate_create_trigger_stmt()

Generates trigger definitions with timing and event options:

- Randomly selects BEFORE/AFTER timing specifiers
- Supports INSERT, UPDATE, DELETE events with equal probability
- Implements column-specific UPDATE triggers with 30% probability
- Creates trigger bodies that prevent tables from growing beyond 1000 rows using RAISE(IGNORE)

generate_insert_stmt()

Creates INSERT statements with these features:

- Implements multiple variant forms: INSERT OR IGNORE, INSERT OR REPLACE, REPLACE INTO
- Generates VALUES lists with 1-2 rows of appropriate data types
- Creates SELECT-based inserts as an alternative to VALUES
- Ensures type-aware value generation based on column definitions
- Adds WITH clauses as CTEs with 10% probability

generate_update_stmt()

Generates UPDATE statements with various conflict resolution strategies:

- Creates statements with UPDATE, UPDATE OR REPLACE, UPDATE OR IGNORE variants
- Avoids modifying PRIMARY KEY columns to prevent constraint violations

- Can create NULL values with for non-primary key columns
- Can generate WHERE clauses targeting specific rows

generate_delete_stmt()

Produces DELETE statements with configurable targeting:

- Adds WITH clauses as CTEs with 10% probability
- Creates complex WHERE conditions with 30% probability using various expressions
- Ensures expressions reference existing columns with appropriate operators

generate_upsert_stmt()

Generates UPSERT statements:

- Identifies conflict targets based on unique/primary key constraints
- Creates DO NOTHING conflict resolutions
- Implements DO UPDATE SET conflict resolutions with 1-3 columns

generate_default_values_stmt()

Creates INSERT statements using SQLite's DEFAULT VALUES syntax:

- Chooses from multiple INSERT variants (INSERT INTO, INSERT OR REPLACE INTO, etc.)

generate_select_stmt() and generate_select_core()

Creates SELECT statements with extensive features and proper SQL semantics:

- Implements WITH clauses and CTEs
- Supports compound queries (UNION, INTERSECT, EXCEPT) with 20% probability
- Generates DISTINCT selections with 20% probability
- Creates complex result columns: expressions, aliases, wildcards
- Implements JOIN operations (INNER, LEFT, NATURAL, CROSS) with appropriate ON/USING clauses
- Generates WHERE clauses with 70% probability

- Creates GROUP BY clauses with 30% probability and optional HAVING clauses with 30% probabilty
- Includes ORDER BY clauses with positional references and sort directions
- Adds LIMIT and OFFSET clauses with configurable parameters

generate_recursive_cte()

Builds recursive common table expressions for hierarchical queries:

- Creates numeric sequence generators using recursive anchor and recursive member
- Generates type-safe operations using CASE expressions to handle non-numeric data
- Implements proper recursion termination with bounded limits

generate_complex_window_query()

Creates queries using SQLite's window function:

- Implements multiple window functions: ROW_NUMBER, RANK, DENSE_RANK in a single query
- Creates running totals using SUM with frame specifications
- Implements sliding window analytics using LAG and LEAD functions
- Creates proper PARTITION BY and ORDER BY clauses for window definitions

generate_json_query()

Produces queries to test SQLite's JSON functionality:

- Implements four distinct query styles with varying complexity
- Uses json_extract with various JSON path expressions (simple, array, nested)
- Creates and manipulates JSON using json_array, json_object constructors
- Implements modification functions: json_set, json_insert, json_replace, json_remove
- Creates advanced queries using json_each and json_tree table-valued functions

generate_subquery_madness()

Creates complex queries with multiple levels of nesting and correlation:

- Generates correlated subqueries in SELECT, WHERE, and ORDER BY clauses
- Creates scalar subqueries returning single values with aggregate functions
- Implements EXISTS subqueries for conditional testing
- Uses subqueries in FROM clauses to create derived tables
- Combines window functions with subqueries
- Ensures stable ordering with ORDER BY clauses

generate_multi_table_join()

Produces queries joining multiple tables:

- Supports joining up to 3 tables with different join types
- Creates join conditions based on schema awareness
- Generates WHERE with 80% probability
- Ensures stable ordering with ORDER BY clauses

generate_expr()

Recursively generates SQL expressions with controlled complexity:

- Implements depth-limited recursion with max_depth parameter (default: 3)
- Creates literal values of various types (numbers, strings, blobs, NULL)
- Generates column references from schema-aware tables_info
- Implements unary operations (NOT, +, -, ~)
- Creates binary operations with 22 different operators
- Generates function calls with correct argument counts and types
- Creates CASE expressions (both simple and searched forms)
- Adds subqueries at lower depths for expressions
- Controls aggregate function usage based on context

generate_column_reference()

Creates references to columns that exist in the schema:

- Only references tables with at least one defined column
- Falls back to literal value ”1” when no valid columns exist

generate_literal_value() and generate_values_expr() and random_value()

Produces SQL literals and VALUES expressions with proper syntax:

- Creates integer literals in range [-1000000, 1000000]
- Generates string literals with 1-20 random characters
- Creates hex blob literals
- Produces boolean literals (TRUE/FALSE)
- Generates NULL literals
- Constructs multi-row VALUES expressions with specified dimensions

1.2.4 Limitations

The generator is specifically for SQLite and does not support other SQL dialects.

It contains only features that are available since v3.26.0 and does not generate statements with components that were introduced in later versions.

The generator is only able to produce a subset of all possible SQL statements as described in the sections above and does not support any arbitrary SQL constructs.

Since the SQL statement generator is designed to produce only valid SQL statements it is not suitable to produce SQL statements for Error Testing.

Despite efforts to only produce valid SQL states, about 0.5% of the statements produced result in an error.

Chapter 2

Evaluation

2.1 Bug-finding capability

During the project we were able to find 9 unique Bugs (1 Crash and 8 Logic Bugs) in v3.26.0 and 5 unique Bugs (0 Crashes and 5 Logic Bugs) in v3.39.4.

2.2 Fuzzer

2.2.1 Experimental setup

In the following section we used the following setup:

- 10,000 queries
- 10 Processes
- SQLite version under test: v3.26.0
- SQLite version used as oracle: v3.49.1
- Intel® Core™ i7-1165G7 Processor

2.2.2 Code coverage

To measure the code coverage we used cgov. After 10,000 queries we achieved 41.46% code coverage using in the version v3.26.0.

2.2.3 Performance

The fuzzer completed the 10,000 queries in 1067 seconds = 17min. 47 sec. Each of the queries was executed on both versions and the outputs were compared. Out of the 10,000 queries, the fuzzer reported a difference between the outputs in 74 cases and in 4 cases only one of the versions reported an

error. 56 queries were invalid and the fuzzer reported the same error for both versions.

2.3 SQL Statement Generator

2.3.1 Experimental setup

In the following section we used the following setup:

- 10,000 queries
- 1 Process
- Intel® Core™ i7-1165G7 Processor

2.3.2 Characteristics of the generated SQL queries

In figure 2.1 we show the query validity: The ratio of semantically valid SQL queries (those that execute without errors) to invalid queries (those that contain syntax or logical errors).

In figure 2.2 we show the average frequency of SQL clauses per query over 10,000 queries.

In figure 2.3 the total frequency of SQL clauses over 10,000 queries.

As already described in the design choices of the SQL statement generator the expression depth can be customized with a variable. We decided to use an expression depth of 4 to be able to create complex enough statements to produce bugs but still not too large such that the execution takes too long.

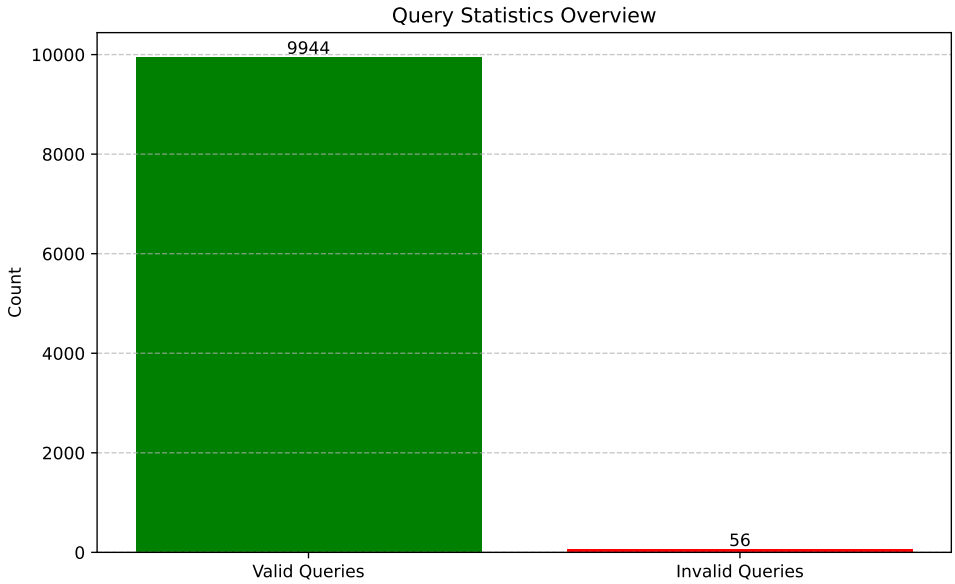

Figure 2.1: Number of valid vs invalid queries over 10000 queries

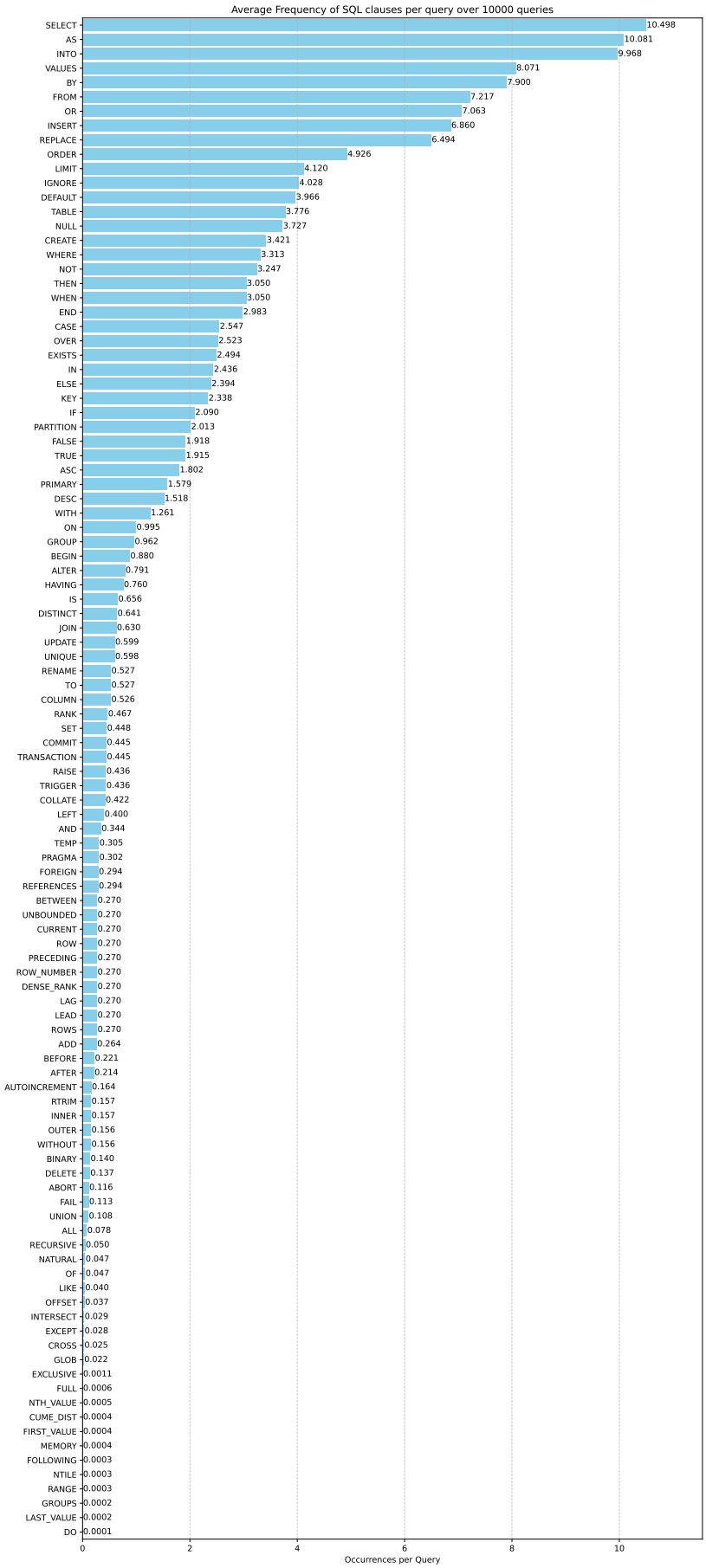

Figure 2.2: Average Frequency of SQL clauses per query over 10000 queries

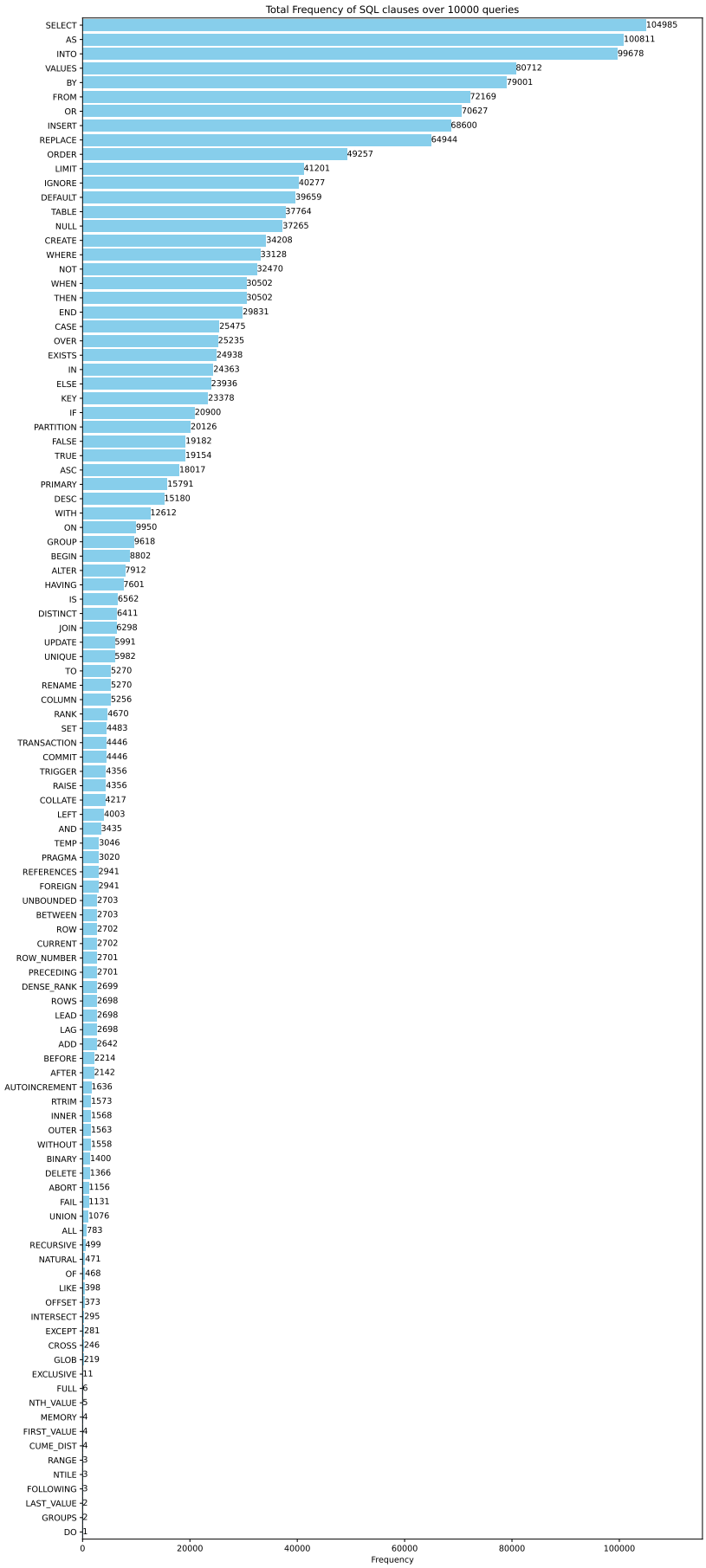

Figure 2.3: Total Frequency of SQL clauses over 10000 queries

2.3.3 Performance

The SQL statement generator produced 10,000 queries in 2.02 seconds. Therefore the generator would produce around 297,030 queries per minute. This could even be improved by generating queries in parallel using more processes, but since the fuzzer is clearly the bottleneck there was no need to improve this.