Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Automated Testing of Database Engines

Project report Part 2

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 reducer. It provides a technical description of the tool, including the design choices we made and their limitations.

1.1 Reducer

1.1.1 Inputs

The reducer takes the following arguments as input:

- `--query`: The SQL query that the reducer will attempt to minimize.
- `--test`: A shell script that checks whether the minimized query still triggers the bug.

1.1.2 Architecture Overview

We used sqlglot to parse the SQL statements into an AST. The reducer follows a hierarchical reduction strategy. It applies the following reduction techniques:

1. **Statement-level reduction**: Eliminates entire SQL statements that are not essential for bug reproduction
2. **Structural simplification**: Removes unused database objects such as tables, columns, and constraints
3. **Expression-level reduction**: Simplifies arithmetic, boolean, and complex expressions to their minimal forms
4. **Delta debugging**: Systematically removes sub-components using a divide-and-conquer approach

Each reduction may enable further reductions made possible by other techniques. Therefore, the main reduction loop performs up to seven iterations, applying the above-mentioned reduction techniques one after another. Before every reduction, we make a copy of the whole AST. If the reduction causes the bug to disappear, we revert the change.

The fixed iteration limit of 7 prevents infinite loops while allowing enough passes for convergence. (In the 20 queries, the maximum number of iterations a query used when testing was four.)

1.1.3 Implementation Details

In this part we look at the implementation details of the four reduction techniques used.

Statement-level reduction

Eliminates entire SQL statements that are not essential for bug reproduction.

Key Functions:

- `reduce_statements_aggressively()`: Removes trivial statements such as `SELECT TRUE`, and filters out non-essential statement types
- `reduce_insert_statements_aggressively()`: Reduces multiple INSERT statements per table to single statements by testing each individual INSERT statement per table
- `reduce_unused_table_inserts()`: Eliminates INSERT statements for tables not referenced in SELECT queries through dependency analysis
- `eliminate_unused_tables()`: Removes CREATE TABLE statements for unreferenced tables by analyzing column references in SELECT statements
- `remove_unused_ctes()`: Eliminates unused Common Table Expressions

Structural simplification

Removes unused database objects such as tables, columns, and constraints

Key Functions:

- `reduce_columns_minimally()`: Removes unused columns from CREATE TABLE statements based on SELECT query analysis
- `reduce_create_table_columns()`: Eliminates columns not referenced in other statements
- `reduce_create_table()`: Removes table constraints, column types
- `flatten_subqueries()`: Eliminates nested subqueries

Expression-level reduction

Simplifies arithmetic, boolean, and complex expressions to their minimal forms

Key Functions:

- `evaluate_arithmetic_expressions()`: Performs constant folding for arithmetic, comparison, and boolean operations
- `remove_unnecessary_parentheses()`: Removes unnecessary parentheses
- `simplify_boolean_expressions()`: Reduces complex boolean expressions to TRUE/FALSE through simplification
- `simplify_case_expressions()`: Replaces CASE expressions with their default values
- `simplify_subquery_expressions()`: Simplifies subqueries by removing ORDER BY/LIMIT/OFFSET clauses and replacing with default values
- `simplify_where_clause()`: Applies arithmetic evaluation to WHERE conditions with iterative simplification passes
- `simplify_join_conditions()`: Reduces complex JOIN conditions to simpler forms
- `reduce_values_expressions()`: Simplifies complex expressions in INSERT VALUES clauses, handling nested subquery structures
- `simplify_sql_constants()`: Applies constant simplification

Delta debugging

We apply the delta debugging approach seen in the lecture. We use delta debugging over the AST. Our delta debugging works on two levels: the statement level and the node level.

The statement level operates on complete SQL statements, attempting to remove entire statements like CREATE, INSERT, or SELECT statements.

The node level operates on individual Abstract Syntax Tree nodes within statements, attempting to remove expression components, clauses, and substructures.

Key Functions:

- `delta_debug_node()`: Applies node level delta debugging
- `delta_debug_statements()`: Applies statement level delta debugging.

1.1.4 Limitations

Performance and Scalability

The reducer has a high copying overhead. For every reduction step, we create a copy of the entire AST.

Writing every reduction to files and creating a subprocess to check the reduction introduces I/O bottlenecks and overhead from subprocess creation.

Delta debugging has exponential worst-case complexity. That's why it is applied last among the four reduction techniques—to reduce the query using simpler and faster techniques first, before applying delta debugging.

Supported syntax

We use sqlglot and support only SQLite. Since the queries are produced by fuzzers, the syntax is sometimes abnormal and accepted by SQLite but not supported by sqlglot. Therefore, we occasionally had to manually correct the parsed statements.

Reduction

The reducer cannot effectively reduce SQL statements that require complex interactions between multiple statements because our reduction techniques treat each component independently, potentially eliminating the specific sequence of CREATE, INSERT, and SELECT operations necessary to trigger the bug. As a result, it may fail to find the minimal necessary sequence to reproduce the issue.

1.2 Test Scripts

Based on the given oracle, we distinguish between two types of bugs: **DIFF** and **CRASH**.

1.2.1 DIFF

The **DIFF** category refers to behavioral differences between SQLite versions 3.26.0 and 3.39.4, where either the query output differs or different types of errors are produced. This occurs in queries 1, 2, 3, 4, 5, 7, 8, 10, 11, 12, 15, 17, and 19. For each query, a shell script invokes the same Python file, checks whether the difference persists after reduction, and exits accordingly.

1.2.2 CRASH

The **CRASH** category involves errors that cause SQLite to crash. Here, we only consider the version specified by the oracle—in our case, SQLite 3.26.0.

Crashes occur in queries 6, 9, 13, 14, 16, 18, and 20. In all these queries except for query 14, a segmentation fault causes the execution to crash. Query 14 instead produces a ”database disk image is malformed” error. As with DIFF, the shell script runs a Python script for each query and checks whether the specific error still occurs after reduction.

Chapter 2

Evaluation

In this chapter, we evaluate our reducer with respect to two main aspects: the quality of the reduction (measured by the number of tokens removed) and its performance (measured execution time in seconds). Table 2.1 presents the results of our evaluation.

Query	**Original Tokens**	**Reduced Tokens**	**Reduction** (%)	**Time** (s)
Query 1	216	36	83.33	3.35
Query 2	481	68	85.86	4.34
Query 3	1564	47	96.99	4.30
Query 4	955	108	88.69	3.64
Query 5	68	43	36.76	2.65
Query 6	3810	162	95.75	10.48
Query 7	619	23	96.28	12.63
Query 8	1694	226	86.66	5.86
Query 9	1245	36	97.11	5.75
Query 10	259	73	71.81	3.68
Query 11	114	55	51.75	4.53
Query 12	3818	81	97.88	4.48
Query 13	287	87	69.69	13.09
Query 14	10787	896	91.69	52.06
Query 15	224	110	50.89	12.70
Query 16	5051	264	94.77	9.32
Query 17	10559	247	97.66	3.11
Query 18	209	166	20.57	8.26
Query 19	211	79	62.56	5.63
Query 20	9539	362	96.21	12.20

Table 2.1: Statistics per query

2.1 Quality of reduction

To evaluate the quality of the reduction, we measure the number of tokens from the original benchmark query and measure it against the number of tokens from the reduced query. To get the number of tokens we used the tokenizer from sqlglot.

Based on the results in Table 2.1, the reducer achieved an average token reduction of 78.65%, with the average size of a reduced query being 158.45 tokens. The most significant reduction was observed in Query 12, with a reduction of 97.88%, while Query 18 had the lowest reduction at 20.57%. Together with Query 5 they are clear outliers. Since both queries are relatively short, there was limited potential for further reduction. We also attempted to reduce those two manually but were unable to achieve a significantly better result. It is also visible in Figure 2.1 that larger queries tend to be more effectively reduced, as many of their statements do not influence whether the bug is triggered or not. This can also be seen in Figure 2.2 as larger queries have a much steeper line.

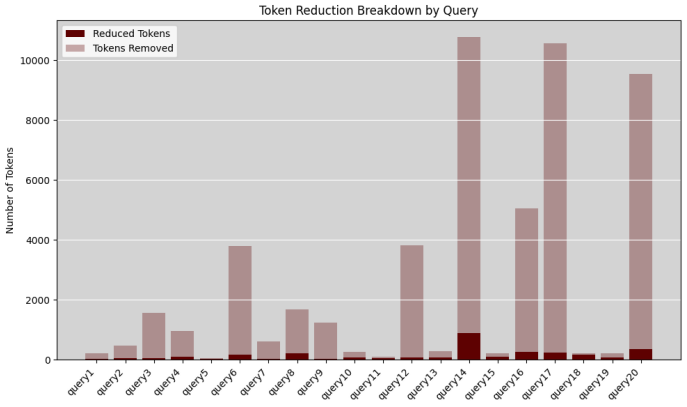

Figure 2.1: Token reduction by query

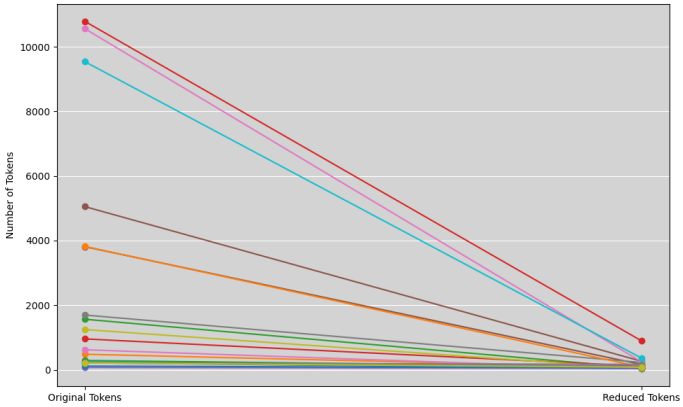

Figure 2.2: Token reduction by query

2.2 Performance

Our setup for measuring performance was the following:

- Intel® Core™ i7-1165G7 Processor
- Python's built-in time module to measure time

Based on the results in Table 2.1, the reducer completes the reduction in an average of 9.1 seconds corresponding to a throughput of approximately 6.59 queries per minute. The reduction times vary significantly, ranging from 2.65 seconds(Query 7 with an initial size of 619 tokens) to 52.06 seconds (Query 20 with an initial size of 9539 tokens).

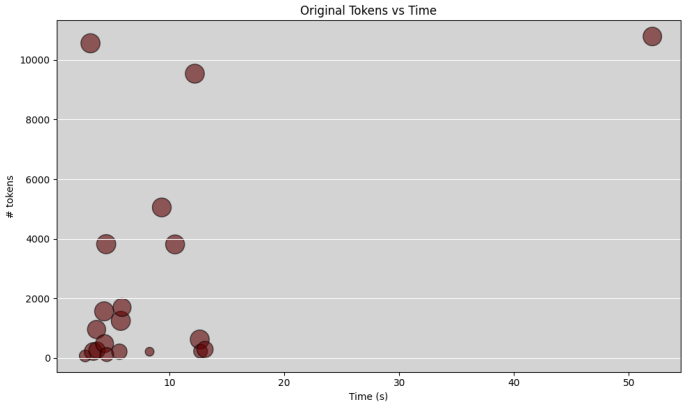

Figure 2.3: Relationship between number of tokens and reduction time

As shown in Figure 2.3, larger queries generally require more time to be reduced. Additionally, the size of the circles in the figure indicates the percentage of reduction achieved by the reducer. There is a clear outlier,

namely Query 17, which starts with 10,559 tokens and is reduced by 97.72% in only approximately 3 seconds. This query contains many INSERT and CREATE INDEX statements, which our reducer quickly discards. As a result, it only needs to reduce a smaller SELECT statement. The same applies to query 20, except that this query has a lot of nested 'SELECT' statements, where only 1 triggers the bug. Based on our design choices, this can be reduced quickly by our reducer. Apart from these cases, we observe that in most instances, the reducer can reduce queries proportionally to their size.