Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Reducing SQLite queries

Fuzzily

Edi Zeqiri, Sascha Tran

Jun 10, 2025

, ETH Zürich

Contents

Chapter 1

Introduction

The SQLite3 Query Reducer is a system designed to streamline SQLite3 queries while preserving their original semantics. It employs a multi-stage reduction methodology to simplify complex SQL statements, yielding minimal yet functionally equivalent queries suitable for debugging, performance tuning, and test-case generation.

The reduction process begins by parsing the input SQL into an Abstract Syntax Tree (AST). A delta debugging phase then iteratively removes and tests query fragments to identify expendable components, followed by a transformation step that refactors the remaining structure for clarity and efficiency. A subsequent brute-force reduction phase examines individual elements to achieve further minimization. Throughout this sequence, unused tables and related DDL/DML statements (such as `CREATE TABLE`, `INSERT`, `UPDATE`, and `DELETE`) are discarded to maintain schema consistency. Finally, the reduced query is validated against the original to ensure equivalent behavior. Implemented in Rust for performance and safety, the system includes a comprehensive suite of test cases to confirm its correctness across diverse query patterns.

Chapter 2

Methodology

Given a SQL query there are many steps to tackle. In the following, the architecture of our SQL reducer is described (Section 2.2), the approach (Section 2.4) of reducing the queries and how we test our reducer (Section 2.3) are described.

2.1 Tools

The reducer in this project was written in **Rust**. Rust was chosen for its combination of high performance and strong compile-time guarantees. Its ownership model enforces memory safety without a garbage collector, and zero-cost abstractions permit fine-grained control over data layout

2.1.1 Parser

We have tried 3 different parsers:

1. sqlparser (Rust)
2. sqlite3-parser (Rust)
3. sqlglot (Python)

None of them could parse all the queries without errors. The not parsed queries were also not the same and did differ, which made it difficult to built around them. Thus, to parse the SQL queries, we have partially implemented our own SQL parser and partially used 1. sqlparser for the queries on which it worked flawlessly.

2.2 Architecture

Our reducer was created such that it can be used following this command:

```
./reducer --query <query-to-minimize> --test <an arbitrary-script>
```

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

Upon calling the reducer using said command, the following process is triggered:

1. Read in SQL query from the given path of `--`**query**.
2. **Reduce** query once and **save** the reduced query into the given environment variable **TEST_CASE_LOCATION**. If this variable doesn't exist, save it at **./query.sql**.
3. **Check** the reduced query using the given test script by `--`**test**. Keep the changes when the reduction was successful.
4. Repeat step 2 until all possible reductions were performed.

2.3 Test reducer

As mentioned, every reduction step needs to be tested on. The test script can be arbitrary. However, it must always end with

- **exit 0**: The bug still occurs, meaning the reduction is valid.
- **exit 1**: The bug no longer occurs, meaning the last modification should be reverted.

In this project there are only two kinds of bugs that could occur:

- **DIFF**: Running the given query into sqlite3-3.26.0 and sqlite3-3.39.4 returns different values.
- **CRASH(3.26.0)**: There was a crash in sqlite3-3.26.0 (e.g. Segmentation fault).

The reducer will be tested on 20 given benchmarks.

2.3.1 The test script

Given the above information, we have composed a test script that **runs** the given query into both database versions and **saves** their output.

In case of a "**DIFF**-bug", it exits 0 when **both outputs are not the same**.

As for the "**CRASH**-bug", the script saves the crash output into a variable, and whenever the script is called, it compares if the output is the **same as the crash output**. If it is, the script exists with 0.

Note: For the DIFF-case, we have observed that the return values of both databases are not exactly the same (for string comparison) but they mean the same thing.
For example "Error: near line ..." and "Parse error near line ..."
If compared this way, we would not have any queries left at the end of the reduction which is why the return values were preprocessed for such cases before comparison.

2.4 Approach

A single SQL script often consists of multiple individual statements, each of which is a complete command that the database executes in sequence. In this context, we define a statement as any standalone SQL command that:

- Begins at the current position in the script (or immediately after the previous semicolon)
- Concludes with a terminating semicolon (;)

Given a SQL query, there are many parts that need to be reduced. Our approach works as follows:

1. First, reduce the number of statements with Delta Debugging (Section 2.4.1).
2. Then, simplify `SELECT` statements.(Section 2.4.2)
3. At the end, reduce the remaining tokens with delta debugging.

2.4.1 Delta Debugging Reduction Step

For this reduction step, we opt for the **delta debugging** algorithm introduced in the lecture, which works as follows (simplified):

1. Receive a list of all individual statements.
2. Split them into x-parts (we decided on setting the granularity x to 2 at the beginning).
3. Run the first part into the test script and check whether this reduction is sufficient. If yes, keep this reduced list and go to step 2 (with x=2). Otherwise continue with the next step.
4. Run the complementary part into the test script (e.g. all = [1, 2, 3], firstpart = [1] => complementary part = [2, 3]). If the reduction worked,

make the next chunks bigger by reducing the granularity size by one and go to step 2. Otherwise, double the granularity.

The algorithm is stopped if the granularity is below 2 or above the number of total statements.
As there can be redundant statements that are in between relevant ones, we will reduce the statements by iteratively **leaving out** a single statement, after delta debugging.

2.4.2 Simplifying `SELECT` statements

To simplify the `SELECT` statements, we will use a **constant folding transformer**.
It evaluates and simplifies constant expressions at compile time, primarily focusing on arithmetic and boolean operations. It handles **numeric operations** (addition, subtraction, multiplication, division) by evaluating them when both operands are constant values, converting the result back into a SQL expression.
For **boolean operations**, it evaluates logical expressions (`AND`, `OR`, `NOT`) and comparisons (equals, not equals, greater than, less than) when the operands are constant boolean values.
The transformer also handles **unary operations** like negation (-) and logical `NOT`, simplifying expressions like `NOT` false to true or -5 to -5. When folding nested expressions, it recursively evaluates inner expressions first, allowing for complex constant expressions to be simplified into their final constant values.

Remark: For this reduction to work, it is required to use the sqlparser mentioned in Section 2.1.1. Unfortunately, this library is unable to parse every query. Thus, only those queries that were successfully parsed could do this reduction step. The other ones skip this step.

2.4.3 Table Removal Step

The approach here is to collect all table names from the `CREATE TABLE` statement and chose which table to remove from the query using delta debug (described in Section 2.4.1).

When decided on a table to remove, the `CREATE TABLE` statement of this table will be deleted and all the corresponding `INSERT`, `CREATE TRIGGER`, `DELETE`, `ALTER` statements will be removed. `SELECT` statements will partially be refactored to remove the table by reducing the `SELECT` without the table. This makes sure that the query doesn't throw any kind of "table not found error" as these will never be successful reductions.

This method of reduction is not used in our final reducer, as it only reduced one query by 5 % of it's original token size. The detailed table can be found in the appendix in Section 5.1.1.

2.4.4 Token Delta Debug Step

Once the preceding steps are complete, we apply token-level delta debugging to exhaustively remove every token. Conceptually, this corresponds to the method described in Section 2.4.1, but operating at the granularity of individual tokens. This final reduction phase serves as a last resort to eliminate any tokens that earlier steps may have overlooked. Chapter 3 shows, that this exhaustive approach substantially increases the overall query-reduction time.

Chapter 3

Results

As mentioned in Chapter 2 there are several reduction steps done. Details about how effective each individual reduction (isolated run) is can be found in the appendix (Section 5.1). We will sometimes refer to statements as stmt.

This section focuses on analyzing our final reducer as described in Section 2.4.
Additionally, the reasoning of why we apply delta debugging on all tokens for the last step will be covered in Section 3.4.

3.1 Overall

Table 3.1 reports statistics for twenty individual queries, measuring the quality of the reducer, along with the time required and the percentage change in token count. Each row corresponds to a query labeled 1 through 20. For each query, the "Orig. Stmts" column shows the number of statements originally present, and "Reduced Stmts" shows how many remain after removal. Likewise, "Orig. Tokens" and "Reduced Tokens" record the token counts before and after removal, while "Time [s]" gives the duration of the removal operation in seconds, and "Token change [%]" indicates the relative reduction in tokens.

Across these examples, most queries lose a vast majority of their tokens, often over 85 percent, demonstrating that the removed content was substantial. A handful of queries (notably 16 and 18) show minimal token reduction, under 5 percent, indicating very little removal. Processing times range widely from under one second for simpler queries to several tens or even hundreds of seconds for more complex ones, and there is no strict correlation between original query size and runtime, suggesting that factors beyond raw token count, such as content complexity or system variability,

also influence performance.

Table 3.1: Summary of the quality of reducing SQl queries.

Query	Orig. Stmts	Reduced Stmts	Orig. Tokens	Reduced Tokens	Time [s]	Token change [%]
1	2	2	51	19	0.89	62.75
2	18	3	284	92	3.26	67.61
3	117	7	901	41	2.22	95.45
4	42	4	504	67	3.99	86.71
5	5	3	44	25	1.11	43.18
6	16	7	2685	228	9.71	91.51
7	32	2	366	17	1.18	95.36
8	41	4	1044	131	7.82	87.45
9	30	22	780	507	46.16	35.00
10	8	2	156	52	2.16	66.67
11	4	3	49	31	1.25	36.73
12	190	4	2834	57	6.02	97.99
13	13	4	169	71	2.61	57.99
14	534	50	7569	580	40.97	92.34
15	13	6	139	75	2.06	46.04
16	2	2	3588	3488	458.71	2.79
17	65	1	4738	4	0.22	99.92
18	2	2	149	146	3.90	2.01
19	4	4	106	97	4.43	8.49
20	16	2	6748	261	5.62	96.13

3.2 Token Reduction

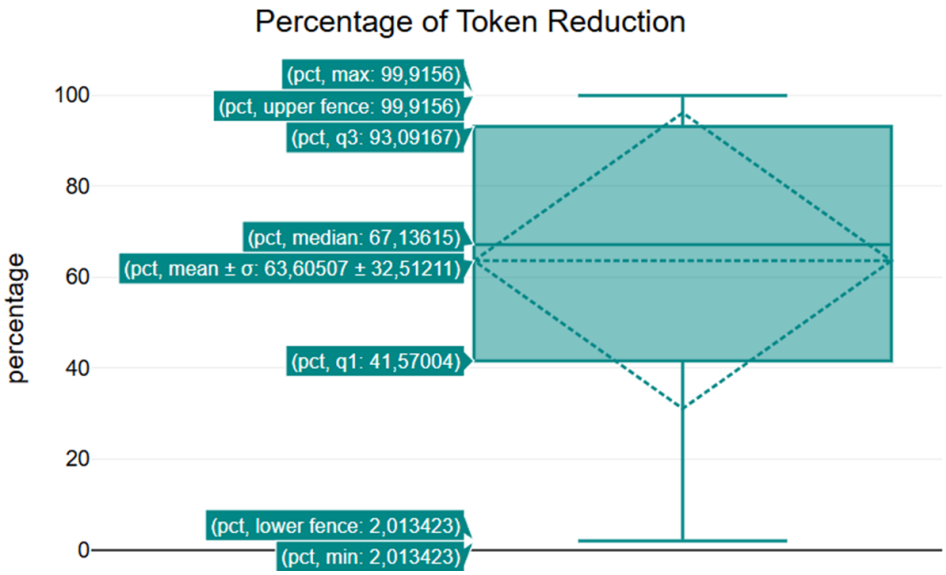

Figure 3.1: Percentage of Token Reduction. Boxplot showing the reduction rate of tokens in percentage.

Figure 3.2 displays the reduction rate of tokens in percentage. It can be observed that on average **63.6** % of tokens with a standard deviation of **32.5** % can be reduced.

3.3 Performance

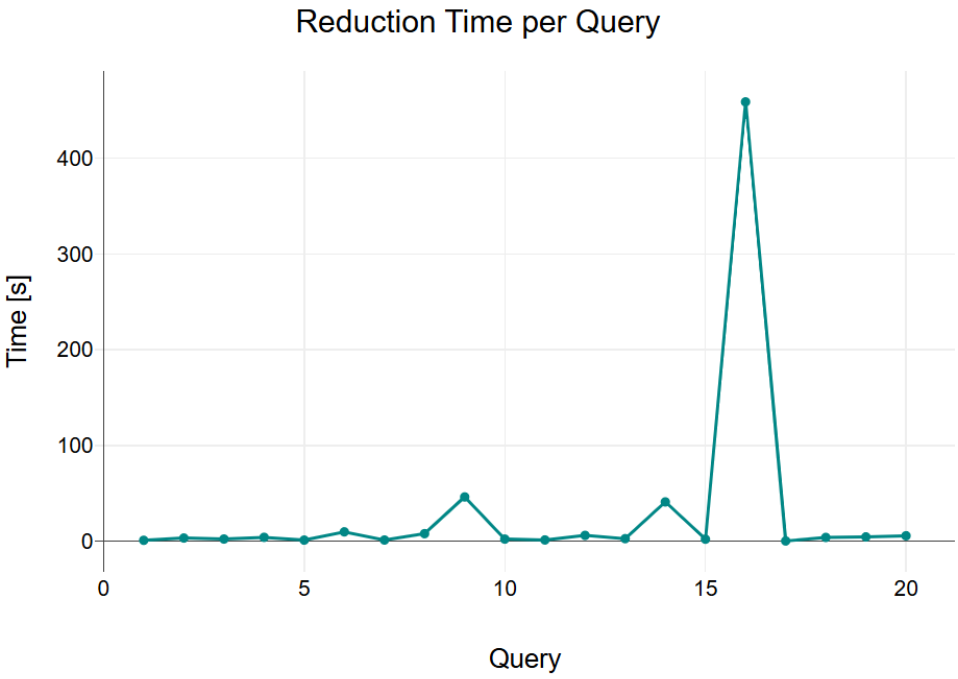

Figure 3.2: Reduction time per query

In Fig. 3.2 it can be observed that there is an extreme outlier at query 16. In order to capture the runtime trend among the queries, Fig. 3.3 is the same figure but with setting the runtime of query 16 to 0. Thus, ignore query 16 in this plot.

The reduction times for each query vary dramatically, ranging from a fraction of a second, up to several minutes.
Most queries complete very quickly, with eleven of the twenty finishing in under three seconds (for example, queries 1, 3, 5, 7, 10, 11, 13, 15, and 17 all take between 0.22 s and 2.61 s), which indicates that the removal operation is lightweight for many inputs. A smaller group of queries fall into a mid range cluster between roughly four and ten seconds (queries 2, 4, 6, 8, 12, 14, 18, 19, and 20), suggesting moderate processing complexity.
Beyond these, three queries stand out as clear outliers, with query 9 taking about 46 s and query 14 about 41 s, while query 16 soars to approximately 459 s, nearly eight minutes. These long runtimes do not simply correlate with original token counts. It is especially visible for query 16, which has 3’588 tokens before reduction but runs orders of magnitude slower than even

larger inputs, implying that certain structural features or content peculiarities can trigger much heavier computation. Overall, the pattern shows that while most removals are nearly instantaneous, occasional cases impose a heavy performance penalty.

Figure 3.3: Reduction time per query excluding query 16. Here, the runtime of query 16 is set to 0. Ignore this data point.

As shown in the table below, token-level processing remains extremely fast (mean 0.02 per token), whereas statement-level execution exhibits much greater variability (mean 11.86, with peaks up to 229.36).

	mean	**std**	**min**	**25%**	**50%**	**75%**	**max**
time-per-token [s]	0.02	0.03	0.0	0.0	0.01	0.03	0.13
time-per-stmt [s]	11.86	51.20	0.0	0.1	0.22	0.44	229.36

Table 3.2: Overview of the runtime of times per token resp. statement.

3.4 Brute Force Step Time Increase

The optimization system demonstrates effective token reduction capabilities, achieving a 90% success rate (18/20 cases) with an average reduction of 10.3 tokens per case and a maximum reduction of 100 tokens. Token reductions varied from 2 to 100 tokens, with larger inputs generally showing greater absolute reductions, while two cases showed no change, suggesting already-optimal configurations.

However, this token efficiency comes at a severe computational cost, with 100% of cases experiencing processing time increases ranging from 1.4× to an extreme 6,723× slowdown. The typical slowdown factor is 8-10×, with Row 16 showing a particularly concerning 68ms → 459s increase that suggests algorithmic complexity issues. While the system successfully reduces tokens, the dramatic processing time penalties indicate this optimization is only suitable for offline batch processing where token efficiency is prioritized over real-time performance, as the modest token savings rarely justify the substantial time costs for most applications.

Table 3.3: Reduced Tokens Comparison Token Delta Debug

Query	Reduced Tokens Without Token Delta Debug	Reduced Tokens With Token Delta Debug	Change
1	26	19	-7
2	96	92	-4
3	65	41	-24
4	72	67	-5
5	27	25	-2
6	243	228	-15
7	17	17	0
8	148	131	-17
9	539	507	-32
10	57	52	-5
11	31	31	0
12	63	57	-6
13	85	71	-14
14	625	580	-45
15	80	75	-5
16	3588	3488	-100
17	11	4	-7
18	149	146	-3
19	103	97	-6
20	267	261	-6

Table 3.4: Processing Time Comparison Token Delta Debug (milliseconds)

Query	Time in Seconds Without Token Delta Debug	Time in Seconds With Token Delta Debug	Change Factor
1	53.45	887.66	16.6× slower
2	267.16	3,255.60	12.2× slower
3	703.56	2,221.54	3.2× slower
4	801.07	3,992.82	5.0× slower
5	127.94	1,111.57	8.7× slower
6	940.48	9,713.41	10.3× slower
7	697.87	1,180.55	1.7× slower
8	316.72	7,822.29	24.7× slower
9	2,141.34	46,164.39	21.6× slower
10	243.91	2,156.60	8.8× slower
11	141.80	1,252.64	8.8× slower
12	4,256.90	6,017.58	1.4× slower
13	306.06	2,609.22	8.5× slower
14	13,866.25	40,971.52	3.0× slower
15	268.42	2,056.60	7.7× slower
16	68.24	458,712.25	6,722× slower
17	107.10	219.96	2.1× slower
18	64.07	3,899.68	60.9× slower
19	137.23	4,426.58	32.3× slower
20	291.18	5,620.42	19.3× slower

Chapter 4

Limitations and Future Work

One of the primary challenges we encountered during benchmark reduction was reliable query parsing. To maximize efficiency, we opted not to develop a parser from scratch—since robust SQL parsers already exist—but we were unable to identify an off-the-shelf solution that consistently handled the specific queries needed for our reductions. Consequently, our reductions struggled with queries containing deeply nested `SELECT` statements, which remained largely unparsed and thus minimally reduced.

Several promising reduction strategies were left unimplemented due to these parsing limitations:

1. `SELECT`**-Statement Reduction.** For example, Query16 comprises only two statements—a `CREATE TABLE` and a deeply nested `SELECT`—yet spans 3 588 tokens. This query failed to parse under `sqlparser`, preventing us from applying reductions such as:
    - Iteratively stripping `WHERE` and `WHEN` clauses.
    - Refactoring nested sub-queries into temporary variables and then eliminating them via delta debugging.
2. **Column- and** `INSERT`**-Statement Reduction.** Similar to our approach for pruning unnecessary tables (see Section 2.4.3), we envisioned surgically removing redundant columns from all clauses. However, this too depended on correctly parsing the `SELECT` statements to identify and excise superfluous columns.

Overcoming these parsing hurdles represents a clear avenue for future work. Integrating a more flexible SQL parser—or extending an existing one—would unlock deeper reductions across complex queries and enable the full potential of our delta-debugging framework.

Chapter 5

Appendix

5.1 Analysis of individual reduction methods

This section displays the detailed results of the individual reduction methods (Section 2.4), demonstrating the reason for not implementing them resp. implementing them.

Note: The following data were run using a compiled version of ”dev” mode (instead of ”release” mode). Thus, they can not be compared with the results given in Chapter 3 in terms of run time.

5.1.1 Remove tables from the statements

This method requires **0.07** $\pm$ **0.12 s** per token (mean $\pm$ standard deviation). It can be observed that only query 14 was reduced by 5.76 % of all tokens with this method. Since there are queries (e.g. query 8 or query 14) that would require additional 20 seconds for using this reduction method and only one query was effectively reduced, we have opted of not using it in our reducer.

Table 5.1: Summary of **removing tables** in a query

Query	Orig. Stmts	Reduced Stmts	Orig. Tokens	Reduced Tokens	Time [s]	Token change [%]
1	2	2	51	51	0.07	0.00
2	18	18	284	284	0.21	0.00
3	117	97	901	901	6.38	0.00
4	42	42	504	504	1.80	0.00
5	5	4	44	44	0.10	0.00
6	16	16	2685	2685	0.40	0.00
7	32	32	366	366	3.71	0.00
8	41	41	1044	1044	23.30	0.00
9	30	30	780	780	4.30	0.00
10	8	8	156	156	0.13	0.00
11	4	4	49	49	0.09	0.00
12	190	190	2834	2834	7.54	0.00
13	13	13	169	169	0.72	0.00
14	534	483	7569	7133	20.91	5.76
15	13	13	139	139	0.71	0.00
16	2	2	3588	3588	0.09	0.00
17	65	64	4738	4738	0.59	0.00
18	2	2	149	149	0.08	0.00
19	4	4	106	106	0.09	0.00
20	16	16	6748	6748	0.79	0.00

5.1.2 Reducing number of statements

Across the dataset of Table 5.2, most queries see only modest statement-level reductions, but token counts often drop dramatically. For example, query 14's statements shrink from 534 to 50, and its tokens from 7'569 to 625, yielding a 91.74 % reduction. Query 17 achieves the highest proportional token reduction, 99.77 %, by reducing from 4'738 to just 11 tokens. Conversely, queries 16, 18, and 19 exhibit no token reduction (0.00 % change), indicating that in those cases the reduction algorithm made no token-level cuts. Other queries, such as numbers 3 and 7, exceed 90 % reduction in tokens, reflecting particularly aggressive trimming.

Processing times vary in rough accordance with input size, ranging from as little as 0.13 seconds (query 1) up to 34.25 seconds (query 14). In general, queries with larger original token counts require more time. That said, some variability exists: query 12, with 2'834 tokens, took 10.78 seconds, and query 3, with 901 tokens, took 2.46 seconds, suggesting that per-token processing cost can fluctuate. Overall, the table demonstrates that the reduction algorithm scales in time with workload and is highly effective at reducing token counts, often achieving substantial compression ratios: on average with **0.06 $\pm$ 0.02 s**

per token.

Table 5.2: Summary of Statistics about reducing the number of statements in a given query.

Query	Orig. Stmts	Reduced Stmts	Orig. Tokens	Reduced Tokens	Time [s]	Token change [%]
1	2	2	51	51	0.13	0.00
2	18	3	284	96	0.70	66.20
3	117	7	901	65	2.46	92.79
4	42	4	504	72	2.27	85.71
5	5	3	44	27	0.35	38.64
6	16	7	2685	243	1.73	90.95
7	32	2	366	16	1.92	95.63
8	41	4	1044	148	0.95	85.82
9	30	22	780	539	3.08	30.90
10	8	2	156	57	0.65	63.46
11	4	3	49	41	0.32	16.33
12	190	4	2834	63	10.78	97.78
13	13	4	169	85	0.71	49.70
14	534	50	7569	625	34.25	91.74
15	13	6	139	80	0.79	42.45
16	2	2	3588	3588	0.16	0.00
17	65	1	4738	11	0.67	99.77
18	2	2	149	149	0.14	0.00
19	4	4	106	106	0.31	0.00
20	16	2	6748	267	0.70	96.04