

Automated Reduction of Bug-Triggering SQL Queries

Reducinator: A Query Reducer for Minimizing SQLite Bug Reproducers

Quy Anh Nguyen and Smail Alijagic

Automated Software Testing, Spring 2026 - Part II

ETH Zürich

June 2026

Abstract

Automated bug-finding tools such as fuzzers tend to emit large bug-triggering inputs that are inconvenient to report or difficult to analyze. Test-case reduction addresses this by automatically shrinking a failing input to a smaller reproducer that still triggers the same bug. In this project, we designed, implemented, and evaluated Reducinator, an automated reducer for bug-triggering SQLite queries. Our reducer applies delta debugging at line-level and token-level granularities to the input query, in addition to removing redundant parenthesized intervals. Our reducer also makes use of caching and multi-threading to enhance reduction speed. We evaluated our reducer on the 20 bug-triggering inputs provided, measuring both quality (percentage of removed tokens) and speed (wall-clock time) on two x86 machines (Intel and AMD). In this paper, we discuss the design of our reducer, results of our evaluations, and limitations of our approach.

1 Introduction

Automated testing tools, such as the fuzzer we built in Part I, routinely produce complex bug-triggering inputs. A fuzzer may emit a 50-line SQL query that triggers a crash, but there is no straightforward way to determine which section(s) of the query is strictly responsible for the failure. Thus, the failure-inducing input might not be immediately helpful. Automated test-case reduction solves this by shrinking a bug-triggering input automatically, producing a much smaller variant (often a few lines) that still reproduces the same bug.

In this part of the project, we built an automated reducer for bug-inducing SQLite queries. Our contributions are:

- A reducer that applies the minimizing Delta Debugging algorithm (`ddmin`) [3] at line-level and token-level granularities, borrowing the idea from Hierarchical Delta Debugging [1]. The reducer additionally implements a syntax-aware reduction pass that removes redundant non-overlapping parenthesized intervals of the input.
- An evaluation of the reducer on the provided benchmarks measuring both quality (percentage of tokens removed) and speed (wall-clock time) on two x86 machines (Intel and AMD).

The rest of this report is structured as follows. Section 2 covers background information on input reduction and delta debugging. Section 3 describes the design of our reducer. Section 4 details the implementation of our reducer and the interfaces it exposes. Section 5 presents our evaluation results along the

two required criteria, quality and speed. Section 6 discusses limitations and alternatives, while Section 7 concludes.

2 Background

2.1 The Input Reduction Problem

Given a bug-triggering input P and a property test ψ such that $\psi(P)$ holds (the bug triggers), the goal of input reduction is to find an input P' with $\psi(P')$ still holding and $|P'| < |P|$, where $|\cdot|$ measures the length of the input (in this case, the number of SQL tokens produced when the input is tokenized by `sqlglot`¹). Finding the *global* minimum is intractable [2], so all practical reducers are heuristic and aim for a local notion of minimality.

2.2 Delta Debugging (`ddmin`)

The seminal reduction algorithm is delta debugging, specifically the minimizing variant `ddmin` [3]. It treats the input as a flat sequence of elements, partitions it into n chunks, and invokes the oracle on every chunk and then every complement. If the bug does not persist on any chunk or complement, the granularity is doubled (capped at the maximum length of the input). On the other hand, if the bug persists on a chunk or complement, the process repeats for said chunk or complement with reduced granularity (2 in the case we reduce to a chunk and $\max(n - 1, 2)$ in the case we reduce to a complement). This process continues until the granularity can no longer be increased. The

¹<https://sqlglot.com/sqlglot.html>

result is *1-minimal*: no single element can be removed without also making the failure disappear. In the worst case, `ddmin` makes $|c_x|^2 + 3|c_x|$ oracle calls, where $|c_x|$ denotes the length of the original bug-triggering input.

2.3 Hierarchical Delta Debugging (HDD)

HDD converts the input to a parse tree and applies `ddmin` level by level, top to bottom [1]. In the case of C code, for example, HDD operates over the code's AST, first pruning high-level structures such as classes and global variables before shrinking lower-level structures like sub-statements and expressions.

Our reducer is inspired mostly by these two works. We implemented `ddmin` and applied it at line-level and token-level granularities. Additionally, our reducer removes redundant non-overlapping parenthesized intervals of the input.

3 Design

3.1 Overview

Figure 1 shows a high-level activity diagram of our reducer. Given a bug-triggering query and an oracle script, the reducer repeatedly applies three reduction passes. Once none of the reduction passes can reduce the query further, the result is written to disk.

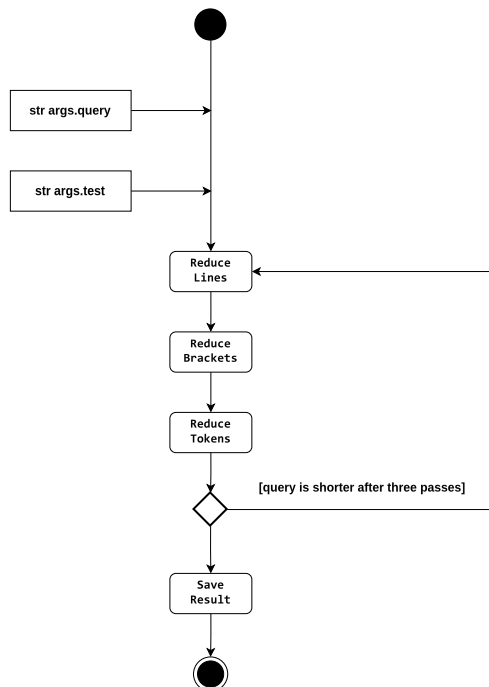


Figure 1: Reducer Activity Diagram

3.2 Oracle

The oracle is fully decoupled from the reducer: it is an external script that checks some arbitrary properties of a particular query and signals the result

through its exit code. Exit code 0 signifies that the bug still persists, while any other exit code means that the bug no longer gets triggered. The oracle takes no command-line arguments. The provided oracle scripts read the candidate query from the path set in the `TEST_CASE_LOCATION` environment variable if that is available or otherwise from `query.sql` in the current working directory. To accommodate this, our reducer writes a candidate query to a unique temporary directory and sets the resulting path as the `TEST_CASE_LOCATION` environment variable before each oracle invocation.

3.3 Reduction Passes

3.3.1 Reduce Lines

The reducer splits the input using the newline character (`\n`) as the separator. The reducer then applies the `ddmin` algorithm described in Section 2 using the list of lines as the sequence of elements.

3.3.2 Reduce Brackets

The reducer tokenizes the input using `sqlglot` and iterates over the list of tokens to find all parenthesis pairs. Then, the reducer builds up a list of candidate removal intervals. These intervals start from either the opening parenthesis or the token preceding it and end at (up to and including) the closing parenthesis. This means the candidate intervals cover both expressions like `(a + b + c)` and function calls like `COUNT(*)`. The reducer then checks if each of these intervals can be removed individually while still passing the oracle (i.e. still persisting the bug). Afterwards, intervals which can be individually removed are sorted by their length in descending order, and the reducer greedily picks the largest non-overlapping intervals. If after removing all the selected intervals, the oracle no longer passes, the reducer falls back to removing only the largest interval. Otherwise, all the selected intervals are removed. Naturally, if no intervals can be removed, this reduction pass just returns the original query.

3.3.3 Reduce Tokens

The reducer splits the input into a list of tokens using the `sqlglot` tokenizer. The reducer then applies the `ddmin` algorithm as described in Section 2 using the list of tokens as the sequence of elements.

3.4 Command-Line Interface

The reducer exposes the interfaces mandated by the project description:

```

1 reducer --query <query-to-minimize> \
2         --test <oracle-script>
  
```

`--query` is the path to the SQL query to minimize and `--test` is the path to the oracle script.

Additionally, the following optional flags are available:

- `--output`: Path to write the reduced query to. Set this flag if overwriting the file provided via `--query` is not the desired behavior.
- `--jobs`: Number of concurrent oracle processes (defaults to `os.cpu_count()`, i.e. the number of CPU threads available)
- `--timeout`: Timeout threshold of each oracle call, in seconds (defaults to 15.0)

4 Implementation

4.1 Programming Language and Dependencies

Our reducer is implemented in Python and runs inside a container built on top of the provided `theosotr/sqlite3-reducer` Docker image. For parsing queries, we used `sqlglot`, which turns each input query into a list of `Token`² objects. The entire reducer fits inside one `reducer.py` file and is around 400 lines of code.

4.2 Caching

To avoid invoking the oracle script multiple times for the same candidate query, we memoize the oracle results using a simple Python dictionary. This is a pure speed optimization and does not affect the final reduction results.

4.3 Multi-Threading

To speed up the reduction process, our reducer spawns multiple threads to execute the oracle in parallel using the `ThreadPoolExecutor`³ API. By default, we spawn as many threads as there are CPU threads on the machine. Our reducer also makes use of the `Lock`⁴ API to implement mutual exclusion for the cache and statistics counters (e.g. number of cache hits and number of oracle script invocations), thereby safeguarding the integrity of the results.

5 Evaluation

5.1 Experimental Setup

We evaluated our reducer using the provided benchmarks (`query1`–`query20`), each consisting of the original query `original_test.sql` and an oracle script `test.sh`. We ran the entire benchmark suite 3 times on each of our two machines (one with an Intel CPU and the other with an AMD CPU). The results are then averaged across 3 runs for each machine. The machines are as follows:

- Machine 1:

```
1 Processor: Intel Core i9-9980HK
2 RAM: 32 GiB
3 Total Cores: 8
4 Total Threads: 16
5 Frequency: 2.4 GHz
6 Operating System: macOS
```

- Machine 2:

```
1 Processor: AMD Ryzen AI 7 350
2 Memory: 32 GiB
3 Total Cores: 8
4 Total Threads: 16
5 Frequency: 2.0 GHz
6 Operating System: GNU/Linux
```

Note that **Frequency** refers to the frequency to which we manually fixed all cores of each machine. The cores themselves can naturally support higher or lower frequencies. We fixed the frequencies by disabling Intel Turbo Boost via **Turbo Boost Switcher for Intel Mac**⁵ on Machine 1 and by running following commands on Machine 2:

```
1 sudo cpupower frequency-set -u 2000000
2 sudo cpupower frequency-set -d 2000000
```

5.2 Reduction Quality

We measured the quality of each reduction by calculating the percentage of tokens removed:

$$\text{reduction}(\%) = \frac{|P| - |P'|}{|P|} \times 100,$$

where $|\cdot|$ denotes the token count of the corresponding query.

Table 1 reports, for each benchmark, the original token count, the reduced token count, and the reduction percentage. **On average, our reducer managed to reduce each query by 84.2%.** The most successful reduction was on `query20`: our reducer managed to shrink the original 9539-token query (863 lines) down to just 97 tokens (a 99% reduction).

²https://sqlglot.com/sqlglot/tokenizer_core.html#Token

³<https://docs.python.org/3/library/concurrent.futures.html>

⁴<https://docs.python.org/3/library/threading.html>

⁵<https://tbswitcher.rugarcia.com/>

Benchmark	Original	Reduced	Reduction (%)
query1	216	93	56.9
query2	481	60	87.5
query3	1564	33	97.9
query4	955	30	96.9
query5	68	31	54.4
query6	3810	101	97.3
query7	619	20	96.8
query8	1694	103	93.9
query9	1245	34	97.3
query10	259	35	86.5
query11	114	44	61.4
query12	3818	72	98.1
query13	287	62	78.4
query14	10787	688	93.6
query15	224	77	65.6
query16	5051	97	98.1
query17	10559	62	99.4
query18	209	92	56.0
query19	211	64	69.7
query20	9539	97	99.0
Mean	2,586	95	84.2
Median	787	63	93.8

Table 1: Reducer Quality. **Original** and **Reduced** show the token count of the original and reduced queries, respectively.

5.3 Reduction Speed

Speed is measured as the wall-clock time to reduce each benchmark. Table 2 and Table 3 report the per-benchmark wall-clock time, number of oracle calls, and number of cache hits on the Intel and AMD machines, respectively. Note that the **Oracle Calls** column refers only to the number of times the oracle script was actually invoked. Therefore, **Oracle Calls** and **Cache Hits** are mutually exclusive events (i.e. **Cache Hits** is NOT a subset of **Oracle Calls**).

Benchmark	Time (s)	Oracle Calls	Cache Hits
query1	9.1	2,844	2,219
query2	7.7	1,781	1,433
query3	5.0	1,113	1,087
query4	4.8	1,013	688
query5	1.4	275	296
query6	12.7	4,558	4,408
query7	1.7	335	287
query8	16.0	3,824	3,657
query9	1.1	349	342
query10	3.9	1,196	1,169
query11	3.2	710	591
query12	6.0	1,367	1,137
query13	4.5	1,582	1,339
query14	369.5	99,665	99,641
query15	14.4	3,283	2,824
query16	626.8	144,326	139,436
query17	12.1	2,802	2,582
query18	6.0	2,118	2,285
query19	6.5	1,476	1,004
query20	7.1	2,374	2,170
Mean	56.0	13,850	13,430
Median	6.2	1,682	1,386

Table 2: Reduction Time per Benchmark (Intel)

Benchmark	Time (s)	Oracle Calls	Cache Hits
query1	1.7	2,845	2,218
query2	1.7	1,781	1,433
query3	1.2	1,113	1,087
query4	1.2	1,013	688
query5	0.4	275	296
query6	20.3	4,558	4,408
query7	0.4	335	287
query8	3.6	3,824	3,657
query9	2.6	349	342
query10	0.9	1,196	1,169
query11	0.7	710	591
query12	1.4	1,367	1,137
query13	8.5	1,582	1,339
query14	107.0	99,665	99,641
query15	3.2	3,283	2,824
query16	492.0	144,327	139,435
query17	3.0	2,802	2,582
query18	9.0	2,118	2,285
query19	1.5	1,476	1,004
query20	12.6	2,374	2,170
Mean	33.6	13,850	13,430
Median	2.1	1,682	1,386

Table 3: Reduction Time per Benchmark (AMD)

As can be seen in both tables, **our reducer runs for less than 10 seconds for most of the provided queries**. The number of cache hits is also quite close to the number of oracle invocations for most queries, meaning that had there not been the caching mechanism, the number of oracle script invocations would have almost doubled, causing the reduction time to increase significantly.

Figure 2 depicts the per-benchmark wall-clock time on both machines. What is rather unexpected is the fact that the reducer ran longer for most queries on the Intel machine than on the AMD machine, even though the former is fixed to a higher frequency than the latter. This is likely explained by the fact that the Intel CPU is several years older than the AMD CPU, and its microarchitecture is therefore not as optimal.

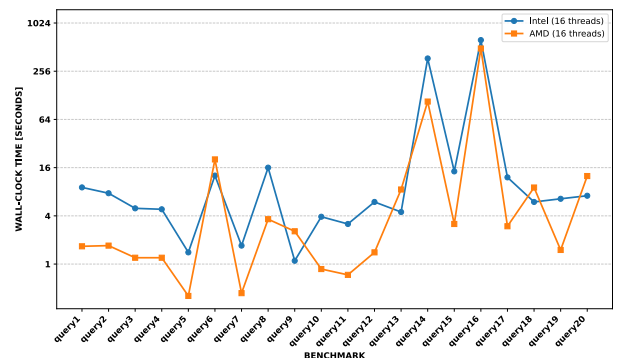


Figure 2: Per-Benchmark Reduction Time

6 Discussion

6.1 Quality versus Speed Trade-off

Our reducer heavily favors quality over speed. Recall that in the worst case, `ddmin` makes $|c_x|^2 + 3|c_x|$ oracle calls, where $|c_x|$ denotes the length of the original bug-triggering input (depending on how the input is split into a sequence of elements, this can be the number of lines or the number of tokens). Our reducer runs this algorithm twice, in the **Reduce Lines** and the **Reduce Tokens** passes. Notice also that in Table 2, our reducer needed 626.8 seconds to reduce `query16`, even with 16 parallel threads. This means that if we were to run the reducer for `query16` with only one thread on the same machine, the wall-clock time could potentially rise drastically to $626.8 * 16 / 60 \approx 167$ minutes = 2 hours 47 minutes. **Thus, our reducer is heavily dependent on CPU parallelism.**

6.2 Limitations

Our reducer is relatively grammar-agnostic and not heavily tailored for SQL grammar. We also only evaluated our reducer on the provided benchmark suite, which contains only 20 queries and is relatively small. Thus, the reducer might not necessarily perform as well on a larger set of inputs.

6.3 Alternative(s)

We also experimented with applying `ddmin` at statement-level granularity, whereby the input query was split into chunks with the semicolon (;) as the separator. `ddmin` then treated the list of statements as the sequence of elements on which to operate. However, most of the queries in the provided benchmark suite have each statement on a separate line already. Thus, `ddmin` performed mostly the same work at statement-level granularity as at line-level granularity. This means that adding an additional pass with `ddmin` at statement-level granularity worsened the reduction speed without improving reduction quality at all. Thus, in the end, we decided not to include this reduction pass.

7 Conclusion

We presented Reducinator, an automated reducer for bug-triggering SQLite queries. By applying delta debugging at line-level and token-level granularities as well as by removing redundant parenthesized intervals, our tool managed to reduce the token count of the 20 provided SQLite queries by 84.2% on average. Our tool additionally implemented caching and thread-based parallelism to speed up the reduction process.

AI Disclaimer

In this project, we used LLMs, particularly Anthropic Claude and Proton Lumo, to aid in the debugging of our code and to write log-parsing scripts.

References

- [1] Ghassan Misherghi and Zhendong Su. Hdd: hierarchical delta debugging. In *Proceedings of the 28th International Conference on Software Engineering, ICSE '06*, page 142–151, New York, NY, USA, 2006. Association for Computing Machinery.
- [2] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for c compiler bugs. *SIGPLAN Not.*, 47(6):335–346, June 2012.
- [3] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, February 2002.