Automated Testing of Database Engines - Our reducer

Kamelia Ivanova and Michelle Vieira

June 9, 2025

Contents

1 Introduction

For the second part of the project, we again refer to the description provided in the project guidelines by the TAs of this course. It involved creating a reducer tool that would, to the best of its ability, reduce a SQL query as much and as efficiently as possible.

First, we will give a quick overview of our submission, explain how our tool can be used, describe the ideas we implemented in our reducer, and then evaluate how well our reducer performs in terms of speed and quality.

2 Technical description

2.1 Content of the reducer

Here, we provide a brief overview of the contents of the reducer:

`AST_reducer`:

- `queries-to-minimize`
- `queries-to-minimize.zip`
- `scripts`
- `scripts.zip`
- `Dockerfile`
- `README.md`
- `controller.py`
- `controller_evaluation.py`
- `delta_reduce_single_statements.py`
- `get_sql_statements.py`
- `reducer.sh`
- `remove_redundant.py`
- `remove_redundant_parentheses.py`
- `remove_select_args.py`
- `remove_where_args.py`
- `remove_with_args.py`
- `replace_nth_bracket_expression_random.py`
- `run_queries.sh`
- `simple_changes_single_statement.py`
- `report.pdf`

2.1.1 Project Directory Overview

`queries-to-minimize`: this folder contains the faulty `sql_queries` called `original_test.sql` (and the oracle files) for which we had to produce reduced test cases with our self-built tool.

`queries-to-minimize.zip`: contains the same content as queries-to-minimize but zipped.

`scripts`: contains the `test-script.sh` files for each query (as described in the project description). A `test-script.sh` is used to make sure that the reduced test actually will produce the same error as `original_test.sql` or not.

`scripts.zip`: contains the same content as `scripts` but zipped.

`Dockerfile`: creates our Docker image (called `docker_image`). As in the first part of the project, we are working in the `WORKDIR /usr/bin`, so that we can run the queries to reduce as well as the test scripts from the location specified by the TAs. It installs some essential packages, `python3` and `python3-pip` as well as all our python files and other folders/resources we need for running our reducer.

`README`: It contains a short manual on how to use our reducer.

`controller.py`: is called in `reducer.sh`. It manages the whole process of reducing `original_test.sql`. It will first call `remove_redundant.py` and then repeatedly run `simple_changes_single_statement.py` and `delta_reduce_single_statements.py` to reduce as much as possible.

`controller_evaluation.py`: can replace `controller.py` in `reducer.sh`. It works in exactly the same way as `controller.py`, with the difference that it also counts the tokens in `original_test.sql` and in the resulting reduced query `query.sql` to compute the quality of the reduction. Additionally, it times the reduction process and returns this value at the end as well.

`delta_reduce_single_statements.py`: will basically try to reduce the SQL query that it gets as input with the help of tokenization and the delta debugging algorithm we looked at in the lecture.

`get_sql_statements.py`: takes `original_test.sql` and splits it into individual SQL statements (splitting based on semicolons and newlines to avoid false splits caused by values inside SQL statements containing semicolons). These SQL statements are stored in separate `.sql` files in the `sql_queries` directory.

`reducer.sh`: this is the main script we use to reduce `original_test.sql`. We provide it with the inputs `--query` and `--test`, as described in the project description. The corresponding queries are located in `queries-to-minimize`, and the test scripts are in `scripts`. An example call of `reducer.sh` looks like this:

```
sudo ./reducer --query /usr/bin/queries-to-minimize/queries/query1/original_test.sql
--test /usr/bin/scripts/queries/query1/test-script.sh
```

Example query is `original_test.sql` from query1, the function can be called inside the docker container.

`remove_redundant.py`: this is called before any other test case reduction procedure. It checks each SQL statement (previously separated using `get_sql_statements.py`) to determine whether it can be completely omitted. If so, the corresponding SQL file is removed from the `sql_queries` folder.

`remove_redundant_parantheses.py`: a script to remove redundant parantheses

`remove_select_args.py`: A script to remove a selected column

`remove_where_args.py`: A script to remove a clause from a where statement

`remove_with_args.py` A script to remove a CTE

`replace_nth_bracket_expression_random.py`: A script to reduce the expression in brackets

`run_queries.sh`: is a helper function that runs the reducer on all 20 queries.

`simple_changes_single_statement.py`: Used to run most reductions outside of the delta reductions

`report.pdf`: The report is of course also part of our submission.

2.2 Usage of our tool

This information is also provided in the `README.md`. First, to use our tool, you need to (as described at the very beginning of the project description) pull and run the code from the GitHub repository `theosotr/sqlite3-test` inside Docker. This is specifically required to ensure access to the two older SQLite versions, `sqlite3-3.26.0` and `sqlite3-3.39.4`.
The code for that, which can again also be found in the `README.md` is:

```
sudo service docker start
sudo docker pull theosotr/sqlite3-test
sudo docker run -it theosotr/sqlite3-test
```

Note: sudo service docker start is to start docker itself (as a friendly reminder).

Next up is actually building the docker image. For this there is a Dockerfile inside our submission folder. So just make sure you run this command inside the folder where the Dockerfile is:

```
sudo docker build -t docker_image .
```

After that you can already run our reducer!
This is done in two ways:
First, you could get inside the docker using this command:

```
sudo docker run -it docker_image
```

And then as briefly mentioned before, one can run `reduce.sh` as described in the project description. Again you have to be inside the docker container to run this (we give query1 as an example):

```
sudo ./reducer --query /usr/bin/queries-to-minimize/queries/query1/original_test.sql
--test /usr/bin/scripts/queries/query1/test-script.sh
```

You will find the resulting query in ..... in the docker container.

Second, one could run it from the comfort of ones local folder like this:

```
sudo docker run --rm -v "$(pwd)/queries-to-minimize:/usr/bin/queries-to-minimize" \
docker_image ./reducer \
--query /usr/bin/queries-to-minimize/queries/query1/original_test.sql \
--test /usr/bin/scripts/queries/query1/test-script.sh
```

In this particular case we still run the reducer as before with the exception that we store the resulting files that are created during the run of reducer on our local folder inside queries-to-minimize.

2.3 Overall Approach to reducing SQL queries

Our approach for reducing the queries is rather simple. We always observe one statement at a time, as most changes are local to one statement. Firstly, we tried to remove some parts in an "informed" manner by trying to remove whole statements that do not affect the result, removing CTEs, etc. Then we run delta reduction on it with limited depth, so we can restrict the processing time on it. We repeat this until no changes can be made. We call calling all reduction operations a round. Adding all of those additional reduction methods turned out to be necessary as the delta reduction was taking a very long time to conclude, and would often miss some reductions.
To check if our current reduction is successful, we test it with our test script. If this is indeed the case, we keep it as the new result, and if not - we skip it. Here it should be noted that this result is also used for the rest of the current operation. An example of that would be removing a clause from a where - instead of continuing with another possible reduction, we proceed to check the rest of the clauses in the new query.
One another note we should make is that we proceed trying to make all the reductions in all rounds as we don't know what the later ones have produced. For instance, delta reduction might have removed the only usage of a table, and we could slowly remove the generation and filling of this table with our "informed" reductions before running it again.
Here we list the different reductions we make in order in a round:

2.3.1 Remove redundant statements

We first iterate through the statements (starting from the last one), and try to remove the current one. If the rest of the query has the expected result, we do so. We do this as often a query would include a lot of unrelated statements that don't actually affect the result - creating and filling tables that are never used again, selecting from them, etc. We start from the last element, as often this allows us to remove multiple statements at once - if we remove the only select from a table, we can also remove the inserts in it, and then the whole table.

2.3.2 Remove redundant parenthesis

Due to the functionality of the delta reduction, often we could end up with results like "((((-1))))", and then be unable to reduce this. The reason for this is that it doesn't take into account the brackets when reducing, and often we would try to remove only one of them, especially as we remove only from one spot at a time. Thus, removing the extra brackets turns out to be almost impossible, and they often cause us to stumble upon syntactically incorrect queries. We have noticed that brackets are one of the main deterrents to the success of the delta reducer, so we try to remove them as much as possible.

2.3.3 Reduce where

We then try to remove every expression from the where clause - starting from the first one and ending with the last one. Often, there are clauses like "True AND ..", which are redundant. We also try to remove the whole where clause, as it is possible that it doesn't affect the bug.

2.3.4 Reduce select

Very similarly to the reduce of where, we try to select fewer elements - this could be particularly useful for cases in which we don't need all of the selected columns and might later help us reduce a lot more, as not selecting column c from a joined table allows us to remove the whole join.

2.3.5 Reduce expressions in brackets

Another reduction is trying to reduce the expressions in brackets. We did this in a couple of ways:

- If the expression is fully arithmetic or logical, we try to calculate it. This would often not fully work in case the expression needed some extra precision but often lead to pretty good results.
- We try to exchange the value in the brackets with true or false
- We try to set the value in the brackets with a random value inside of them.
- If there is only one value in the brackets, we remove them.

We try to assign all of those values and then test if the query is still resulting in the same bug.

2.3.6 Reduce With expressions

Another variant of the select and where reductions - we try to remove a CTE from the With clause and check if it was successful. Again, we loop through all of the CTEs and try to remove them one by one, very similarly to also the statement reduction.

2.3.7 Delta reduction

After all intuitive reductions were done, we try to run Delta reduction. Initially, we tried to do this with the whole query but this lead to a lot of syntax errors, as often we would attempt to reduce the end of one query and the beginning of the next one. Then we moved on to doing delta reduction on a single statement. When doing this, we encountered a similar problem - often, tokens would get cut off in the middle, causing syntax errors. Thus we now run it based on the tokens in the statement and not based on the characters.
We also noticed that the earlier preprocessing of the queries greatly improves the quality of the delta

reduction, as the shorter a query is, the better it manages to reduce it.
Another important note is that we limited the depth of the recursion - the reason for this is that the reduction would cause runtime error, as it would run for an extended period of time. We chose a number where it was still successful and it didn't cause the error anymore, and also stayed within some sensible time limit.

2.4 Usage of sqlglot

For some of those actions, we used sqlglot which allowed us to build the AST tree and remove specific parts of it. We also tried using it for checking the syntax of a query before running tests, but concluded fast that it was not that useful, as it would reject sql queries that our version would accept.

2.5 Limitations and potential improvements

Another improvement we could add would be to implement a reduction similar to the select, where and with reductions for different operations. A clear candidate for this would be CASE WHEN THEN, as with it we face a similar issue as with the brackets during delta reduction. Another one would be INSERT INTO, as we often include many values that we might later filter out, and this could also help us with the removal of some where clauses.
Another improvement would be to try and remove table columns. As this is an operation that requires most of the statements and is often not easy to implement (a clear example would be the INSERT INTO statements, which don't explicitly mention the column name), we chose to omit this reduction.
Another improvement would be finding a better balance between the different types of reductions - currently, we attempt all of them during every round but it is very possible that this is not needed, and some of them are run many times without achieving a better result. Most probably, running the ones that recently reduced the query first and omitting others in case a reduction in the current cycle was made would be beneficial for our time performance.

3 Testing the results

For testing the results, we created a separate `test.sh` script for each test. Initially, we tried using a single script for all tests, which compared the outputs and error codes, but this approach turned out to be too generic. By using separate scripts, we were able to target the specific bug we wanted to detect, which resulted in much better reductions.
We tended to be rather conservative in our tests - we would keep the exact same output in case of a difference in output, but would just require a crash in case one was expected.

4 Evaluation

Last but not least, we will evaluate our reducer based on the two criterias also mentioned in the project description: Quality of reduction and Speed of reduction.

The quality of reduction refers to the percentage of reduction measured in terms of the number of tokens in the SQL query. To evaluate the speed of the reducer, our task was to track the overall time that was needed to reduce each of the given test cases.

We were given 20 SQL queries that trigger bugs in the SQLite database to evaluate our reducer on.

4.1 Methodology

We specifically use a different version of our `controller.py` file to obtain these values, called `controller_evaluation.py` (which was also mentioned in section 2). This is run in place of `controller.py`, which is normally called in `reducer.sh`. In it, we call each function and time it and then count the number of tokens in the original and reduced versions of the query.

To measure the quality of the reduction, we count the number of tokens in the `original_test.sql` file and in `query.sql` and then calculate the percentage difference between the two values:

$$\begin{aligned} A &= \#\text{ of tokens in } \texttt{original\_test.sql} \\ B &= \#\text{ of tokens in } \texttt{query.sql} \\ \text{quality of reduction } [\%] &= \frac{A-B}{A} \cdot 100.0 \end{aligned} \tag{1}$$

To measure the speed of the reduction, we record the time using `time.time()` at the start and end of the while loop in `controller_evaluation.py` (shortly before reduction happens and end it when the reduction process is done).

What we get for each query are these results (using the average of 5 runs for each query) displayed in Table 1:

query	Quality of reduction in [%]	Speed of reduction in [s]
query1	85.19	5.06
query2	76.39	12.4
query3	92.57	9.33
query4	75.19	34.2
query5	36.51	2.36
query6	93.5	175.09
query7	86.44	4.95
query8	84.79	22.26
query9	97.68	8.45
query10	30.71	15.15
query11	55.35	5.99
query12	98.28	15.12
query13	77.82	13.76
query14	94.87	155.92
query15	55.98	9.03
query16	97.89	68.17
query17	98.85	12.42
query18	49.77	15.8
query19	17.06	24.43
query20	98.74	17.51
AVG	75.18	31.37
TOTAL	/	627.40

Table 1: This table shows the quality and speed of reduction for each of the 20 queries we could test our reducer on.

4.2 Quality of reduction

On average, taking all the values from Table 1, we have a percentage reduction of ~75%. For some queries we have rather poor results like query5, query10 and query19, which have percentage reduction below 50%. In those cases, the queries were rather small in size to begin with - for instance, query 5 consists of barely 5 short statements, so even this lower percentage of reduction is sizeable in this context.
One can also argue that these specific queries might give clues on what one could further implement in the reducer to increase said percentage (and maybe even the percentage of reduction for other queries in the 20 queries batch). We have listed the improvements we saw after analyzing our results in the Limitations and Furhter Improvements section.

4.3 Speed of reduction

Most of the queries seem to run in a reasonable time, and even the worst performing query reductions, such as query6 and query14 take less than 3 minutes. Also, even though it takes longer, the quality of reduction makes up for it, since we manage to achieve almost 95%. Overall, the average time needed for one query is around 31 seconds and the total time needed for all the queries is around ∼630 seconds (10 minutes and 30 seconds).
We believe that the speed of reduction is also partly related to the length and the "reducability" of a query - in case there are a lot of things that could be attempted, the process would take considerably longer.
One could reduce the time spent by removing some of the test case reduction steps described in Section 2.3 from the reducer and finding a better trade-off between the quality of reduction and speed. Further research could optimize this relationship, but we focused on achieving reasonable runtime and good-quality reduction.

Additionally, when it comes to the speed of reduction, one has to consider that not all query files had the same size (number of statements), and the level of nested-ness of SQL queries also impacted how quickly a reduction could be performed.
We believe that we managed to improve the initial delta reduction algorithm immensely - the clear implementation took more than 15 minutes for even the shorter queries.

5 Conclusion

Our reducer achieved an overall quality of reduction of approximately 75% with a total time of 630 seconds for 20 queries (roughly 30 seconds per query). Although this may seem like a significant amount of time, we are fairly satisfied with these results. At the beginning of this second part of the project, using our initial simple delta debugging algorithm (as explained in the lecture), reductions unfortunately took much longer, even hours for some queries that now take only a few minutes. Of course, there are still improvements that could be made: further reduction strategies to try (as already mentioned in Section 2.5), or potentially omitting some steps in favour of faster reduction. However, given our somewhat limited time, we leave these enhancements for future work.