Testing the SQLite Engine with Randomized Transformed Queries

Calvin Li, Ben Mishkanian

1 Abstract

This paper describes a novel method for testing database engines by transforming random queries. Our approach combines aspects of random testing and differential testing to provide semi-automated functional validation, without requiring translation between SQL dialects. We validate our approach by implementing the algorithm on a SQLite DBMS.

2 Introduction

Database engines, like any other software, must be tested for functional correctness. Random testing, which involves randomly generating input values, is easy to implement but on its own cannot test for correctness; it can only test for crashes or hangs. Instead, we can use differential testing, where we run the same input on two programs, in order to test correctness. This approach works well in testing compilers, but not as well for testing database engines, because SQL is not as standardized as other programming languages. Different engines have slightly different implementations of SQL, so pure differential testing cannot fully cover all the cases.

The method described in this paper approaches this problem from a different angle, combining aspects of random and differential testing. We begin by generating a random query, and running it to get the reference output. Then, we make some transformations to the query in a way where the new query is functionally equivalent to the old one, and run this new query. If the results are different, then the query is possibly error-revealing.

Initially, we'll target SQLite. It uses a relatively simple flavor of SQL, making it easier to work with. It also has not been tested as much as other engines, which means there are more bugs, so we should have more interesting results.

The remainder of this paper is structured as follows: Section 3 provides a motivating example for our approach. Section 4 describes the implementation and design of our algorithm applied to SQLite as a target. Section 5 is a summary of our experimental results. Section 6 lists related work. Section 7 lists the contributions of the authors. Section 8 explains sources of future work for this study. Finally, Section 9 concludes.

3 Motivating Example

Say we have a database with tables `A` and `B`. `A` has a column `Income`, and `B` has column `Age`. Before we start generating random queries, we determine that nobody has an income over 10000 and an age over 50. Then we can add the clause

```
A.Income > 10000 & B.Age > 50
```

to the `where` clause of any query that uses `A join B`. Then, the output should be the same. If it is not, it may indicate a bug in the DBMS. We can do this and similar transformations, along with combinations of them, in order to look for discrepancies.

4 Technical Approach

The first step of our technique is to randomly generate SQLite queries. We must also randomly generate data tables in an SQLite database. Next, we find predicates in the database that are either always true or always false. Finally, these invariants will be appended to the WHERE clause of the randomly generated queries, creating transformed queries that should yield the same result when executed.

4.1 Generating Random Queries and Data

We used the aptly-named Random Query Generator (`https://launchpad.net/randgen`) to generate our random queries. By default, RQG generates MySQL queries, but it can be used to generate SQLite queries when provided an SQLite grammar in YACC (.yy) format. Although the SQLite grammar is relatively simple, writing the grammar file still takes a lot of work. Furthermore, we wanted to avoid generating database-altering queries, as they are more difficult to validate. Therefore, we only generate a subset of the SQLite queries, namely selection queries. We also limited the level of subquerying. Finally, we made `OR` twice as likely as `AND` when joining conditions, which made empty results less likely. Supporting a larger subset of the SQLite language is a source of future work.

Before RQG can generate random queries, it must be provided a target database. This provides necessary metadata to RQG, such as the data types of the table columns, allowing it to generate semantically valid queries. RQG is capable of randomly generating MySQL databases, given a .zz file which describes some qualitative information about the desired schema, such as allowed column data types. We used RQG to generate a MySQL database with columns of type pk (primary key), int and varchar, and then used a conversion tool to convert this database to a SQLite database. The original MySQL database was also used as input to RQG to generate random SQLite queries, as described above.

4.2 Finding Database Invariants

When transforming queries, we require that the transformed query should have the same effect as the original query. In order to ensure this, we need our transformations on WHERE clause predicates to have no effect on the truthfulness of the original predicate. We accomplish this by only using predicates that are always true or always false. We wrote a script findinvariants.py to inspect the database and return a list of such invariants. At present, the script finds the largest integer L in column C and emits the predicate $C \leq L$.

In the future, we plan to find more advanced invariants. For example, another type of invariant could be built as follows: First, generate a subquery S that returns an entire column C of our table X. Then we can use the predicate `X.C IN S` as an invariant. There are many different types of possible invariants involving subqueries, and using a variety of different ones increases the probability of triggering and detecting faulty DBMS code.

4.3 Transforming Queries

Transforming queries involves three steps: first, we parse the original query to find the WHERE clause; second, we choose an appropriate invariant and append it; third, we output the transformed query. We parse and transform the queries using ANTLR (`http://www.antlr.org/`), a tool which generates lexers and parsers. ANTLR4 takes as input a .g4 grammar file, and outputs a lexer and parser for that grammar. A grammar file for SQLite is publicly available on GitHub. We use the ANTLR parser to find the WHERE clause, choose a semantically valid invariant, and output a transformed query.

ANTLR's parser generates a parse tree for a given query, breaking down the query into tokens and grouping them based on rules. For example, `expr` (a rule representing a predicate) is a child of the clause `wkere_expr`. The parse tree is traversed using the `Visitor` class, whose behavior can be altered through subclassing. Different locations in the tree can be referenced by using an index. While a Visitor is traversing the parse tree, it calls functions that correspond to individual productions of the original grammar. By overriding these functions, we can create hooks to obtain the location index whenever a WHERE clause expression is encountered. Once we have this index, we know where to insert the invariant predicate.

One additional complication to the transformation is that we must ensure that the chosen invariant is semantically valid. In particular, we must ensure that the table referenced by the original query is the same as the table referenced by the invariant. We wrote an InvariantFinder class to address this problem; it takes as input a table name and outputs a randomly chosen invariant referencing that table. We obtain the query's table name by inserting additional ANTLR hooks.

Once the invariant is chosen and the proper index is found, we output a transformed query by inserting the invariant at that index.

5 Evaluation

We have a script that generates a random SQLite query and then runs it on our randomly generated data. Then, it pipes the result to our ANTLR parser, which finds an appropriate invariant and expands `where_expr` with it. Next, it gets the output of the transformed query. Finally, it compares the two results. Any errors are also recorded.

All the random elements of the script are based on a seed, so results can be replicated as needed. This also makes it easy to label each test case. We set up extended runs by running the script repeatedly, starting with a seed of 1 and incrementing.

Initially, the script ran for about two hours before running out of disk space and stopping. It evaluated almost 2000 queries, finding over 400 potential bugs (transformed queries with different output than the original). However, upon further inspection, there was a problem in our data; we had included null values in our tables, which return false for any comparisons.

5.1 Final Results

We changed configuration and generated some new data, then updated our invariants. We ran the script for 4314 queries, and found 208 potential bugs for a 4.8% error rate.

5.2 Processing Potential Error-revealing Queries

We examined test case 12, which was the first one to have a discrepancy between the original and transformed query. The query is

```
SELECT DISTINCT X.col_int_not_null_nokey
FROM `B` AS X
WHERE
 1 AND
 X.col_int_not_null_key <= 8171008 AND
 X.pk <= 25 AND
 X.col_int_not_null_key <= 8171008 AND
 X.col_varchar_not_null_nokey >= 'a' OR
 X.pk <= -1866240
GROUP BY X.col_varchar_not_null_key
HAVING X.col_varchar_not_null_key <= 'y'
ORDER BY col_varchar_not_null_key DESC  ;
```

Red text is the part added by ANTLR. We manually reduce the test case to make it easier to handle. The simplified version is

```
SELECT DISTINCT X.col_int_not_null_key,
 X.col_int_not_null_nokey,
 x.col_varchar_not_null_key,
FROM `B` AS X
WHERE
 X.col_int_not_null_key <= 8171008 AND
 X.col_varchar_not_null_nokey >= 'a'
GROUP BY X.col_varchar_not_null_key
ORDER BY col_varchar_not_null_key DESC  ;
```

Once again, the red text has been added by ANTLR. The original query outputs

```
5|2|y
4|-605184|x
7598848|9|w
2|4805888|u
-5697792|3056384|t
4|4|s
-646400|8|r
-5172736|8349440|q
8|-7762688|k
3|6108160|j
8171008|1|i
2|413952|h
7|2457344|f
-1505024|8|d
2|-2563328|c
0|3|a
```

while the transformed one outputs

```
5|2|y
4|-605184|x
```

```
7598848|9|w
2|4805888|u
-5697792|3056384|t
8|2|s
9|9|r
-5172736|8349440|q
8|-7762688|k
3|6108160|j
8171008|1|i
413952|2|h
7|2457344|f
9|970240|d
2|-2563328|c
0|3|a
```

Differences are highlighted in red. It appears that adding the invariant changed the pre-sorted order of the rows, which affects which row is used for the `GROUP BY` clause. We would need to do something similar for all queries that return different results when transformed.

6 Related Work

A large body of work deals with testing and creating queries for SQL databases. Tuya, Cabal, and Riva[Tuya et al. 2006] provide the SQLMutation tool, which generates mutants of a given SQL query, useful for detecting bugs in database test suites. Khalek and Khurshid[Abdul Khalek and Khurshid 2010] describe a method that ensures syntactic and semantic correctness of generated queries by using the Alloy tool to reduce it to a SAT problem. These queries can then be fed to the ADUSA tool, which automates black-box database testing by running queries and comparing the results to generated test oracles. Bati, Giakoumakis, Herbert, and Surna[Bati et al. 2007] present a genetic programming algorithm which generates a set of queries that achieve higher code coverage in the DBMS under test.

[Elmongui et al. 2009] Presented a framework and general principles for testing SQL query transformations. Particularly relevant is their ideas concerning test coverage, ensuring that there are are multiple queries that utilize each transformation rule. There is also a MSR paper [Grabs et al. 2008] that outlines testing query optimizers. Since query optimizers also transform queries in an output-preserving way, some of the guidelines used to test them also apply to our project. Finally, [Mishra et al. 2008] presented a paper on generating target queries for database engine testing. However, their work mainly deals with producing outputs near a target cardinality. Although this might help us generate test cases that run faster, it is an optimization that is inessential for our testing.

7 Contributions

Milestone 1: Since there was a lot of reading, we up split the articles when looking for related work. Afterwards, we summarized the relevant parts. Ben then worked on the related work section, while Calvin wrote the introduction, motivation, technical approach, and evaluation using parts of the proposal. After everything else was done, Ben wrote the future work section.

Milestone 2: Calvin fixed bugs in the query generator grammar which were previously resulting in invalid queries. Meanwhile, Ben worked on setting up ANTLR and writing some basic query parsing code. Next, Calvin started working on the ANTLR transformation logic, while Ben wrote a script to find invariant predicates in the database. For the milestone 2 report, Calvin updated the evaluation and future work sections, and Ben updated the technical approach and contributions section.

Final Report: Calvin finished writing the ANTLR Visitor, which inserted invariants into queries. Ben finished the script for finding invariants and interfaced it with the Visitor so it can actually find the invariants. Then, Calvin worked on running the scripts while Ben worked on the slides and presentation. Finally, Calvin worked on the Evaluation, Future Work, and part of the Conclusion sections while Ben polished all the others.

8 Future Work

In the interest of time, our grammar for SQLite was greatly simplified. In the future, we should expand the grammar, particularly to include subqueries. This not only makes the queries much longer (and more complicated), but also gives us more places to insert invariants.

We can also make our invariants more complicated (e.g., using subqueries instead of constants). For example, we can take a query like

```
SELECT * FROM a;
```

and add an `IN` clause that is always true:

```
SELECT * FROM a
WHERE `pk` IN
(
    SELECT `pk` FROM a;
);
```

Once again, this should not change the result. Furthermore, it would force the engine to do more work, particularly optimization, giving a greater chance of uncovering a bug.

Finally, the process of reducing test cases and testing to see if they are actually error-revealing is time-consuming to do manually, so we should try to automate it as much as possible. There has already been some work on automatically reducing test cases.[Zeller and Hildebrandt 2002]

9 Conclusion

In this paper, we presented a novel method for testing a DBMS by transforming randomly generated queries. Our method provides functional validation without requiring translation between two SQL dialects. We evaluated our method by implementing it on SQLite. The initial results are promising, finding many potential problematic queries and some actual bugs. There is also much room for expansion, since we can make both the grammar and invariants more sophisticated.

References

ABDUL KHALEK, S., AND KHURSHID, S. 2010. Automated sql query generation for systematic testing of database engines. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ACM, New York, NY, USA, ASE '10, 329–332.

BATI, H., GIAKOUMAKIS, L., HERBERT, S., AND SURNA, A. 2007. A genetic approach for random testing of database systems. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB Endowment, VLDB '07, 1243–1251.

ELMONGUI, H. G., NARASAYYA, V., AND RAMAMURTHY, R. 2009. A framework for testing query transformation rules. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, ACM, New York, NY, USA, SIGMOD '09, 257–268.

GRABS, T., HERBERT, S., AND ZHANG, X. S. 2008. Testing challenges for extending sql server's query processor: A case study. In *Proceedings of the 1st International Workshop on Testing Database Systems*, ACM, New York, NY, USA, DBTest '08, 2:1–2:6.

MISHRA, C., KOUDAS, N., AND ZUZARTE, C. 2008. Generating targeted queries for database testing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ACM, New York, NY, USA, SIGMOD '08, 499–510.

TUYA, J., SUAREZ-CABAL, M., AND DE LA RIVA, C. 2006. Sqlmutation: A tool to generate mutants of sql database queries. In *Mutation Analysis, 2006. Second Workshop on*, 1–1.

ZELLER, A., AND HILDEBRANDT, R. 2002. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng. 28*, 2 (Feb.), 183–200.