Milestone 2

Calvin Li, Ben Mishkanian

1 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.

We will approach this problem from a different angle. We will generate a random query, then run it to get the correct output. Then, we will 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.

2 Motivation

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. We can do this and similar transformations, along with combinations of them, in order to look for discrepancies.

3 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.

3.1 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. Since SQLite syntax can be very complex, we created a grammar file sqlite.yy that contains the grammar for SELECT statements only. Generating and testing other types of queries is one of our goals for future work.

In order for RQG to generate random queries, it must be provided a database schema. RQG is capable of randomly generating MySQL databases, given a .zz file which describes some qualitative information about the desired schema, such as possible column data types. We used RQG to generate a MySQL database with columns of type int and varchar, and then used a conversion tool to convert this database to a SQLite database.

3.2 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 $C \leq L$.

3.3 Transforming Queries

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

4 Evaluation

So far, we have a query generator and parser set up. In order to test them, we wrte a srcipt that generates a query based on a seed, then pipes the result to our antlr parser. The parser includes a visitor that walks the parse tree and prints out the tokens, allowing us to further pipe its output to sqlite. We then write another script that runs this one repeatedly, with an incrementing seed. Then, we can leave that script running in the background and look for error messages, which would indicate that there is something wrong with our code.

We have done a smaller test run of 10 queries, without any problems.

5 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.

6 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.

7 Future Work

Before the milestone #2 deadline, we will use Random Query Generator to generate sample databases and random queries. We will create grammars describing valid query transformations and use these grammars to transform the random queries. In order to provide preliminary proof-of-concept results, we may manually manipulate a few grammars to generate transformed queries, and run these queries against the SQL server. The output will be compared to that of the original query, in order to evaluate our approach.

For the rest of the class, we will focus on transforming our queries, since all the pieces are in place for us to do that. First, we need to write functions to walk along the antlr parse tree and insert the where clauses we need. Then, we can start doing long runs. We might also generate queries before the visitor is ready, in order to save time later. After all that, we should (hopefully) start finding bugs. Then we can focus on improving our transformations by adding more complicated invariants.

We can also use subqueries instead of constants for our invariants. 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.

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.