X-Data: Generating Test Data for Killing SQL Mutants

Bhanu Pratap Gupta, Devang Vira, S. Sudarshan

Indian Institute of Technology, Bombay

bhanupratap2006@gmail.com, devang.vira@gmail.com, sudarsha@cse.iitb.ac.in

***Abstract*— Checking if an SQL query has been written correctly is not an easy task. Formal verification is not applicable, since it is based on comparing a specification with an implementation, whereas SQL queries are essentially a specification without any implementation. Thus, the standard approach for testing queries is to manually check query results on test datasets. Intuitively, a mutant is a query variant that could have been the correct query if the query was in error; a mutant is killed by a dataset if the original query and the mutant return different results on the dataset.**

In this paper, we address the problem of generation of test data for an SQL query, to kill mutants. Our work focuses in particular on a class of join/outer-join mutants, which are a common cause of error. To minimize human effort in testing, our techniques generate a test suite containing small and intuitive test datasets, combining them into a single dataset where possible. In the absence of foreign-key constraints, and under certain assumptions, the test suite is complete, i.e. it kills all non-equivalent mutations, in the class of join-type mutations that we consider. We also consider some common types of where-clause predicate mutants. Our techniques have been implemented in a prototype data generation tool.

I. Introduction

SQL queries are very widely used, but checking if a query meets the intended goals is not an easy task. Formal verification is not applicable, since it is based on comparing a specification with an implementation, whereas SQL queries are essentially a specification without any implementation. In practical settings, programmers check the query results against multiple test cases, to see if they give desired results. However, if test cases are created in an ad-hoc manner, it is not clear if all meaningful test cases have been covered, or there are cases that have been omitted.

Mutation testing is a well known approach for checking if test cases are adequate for a program. Mutation testing involves generating mutants of the original program by modifying the program in a controlled manner [1]. A mutation of a program is a single syntactically correct change; a mutant of a program is the result of one or more mutations on the program. Mutations model typical programming errors like using the wrong operator or variable name. Mutation testing is well known in imperative programming.

If the given program is faulty, it is possible that one of the mutants was the intended program. A test case would detect the fault if it gave different results on the correct program and on the faulty program. A test case that gives different results on the given program and the mutant program is said to *kill* the mutant.

The notion of mutation testing can also be applied to database queries. Query mutations model common mistakes made by programmers when specifying queries, for example, using a wrong relational operator in a where-clause condition, a wrong join operator e.g. an inner join ($\bowtie$) used instead of a left outer join ($\sqsupset\!\bowtie$), missing joins conditions, etc. For example, given the query

*SELECT * FROM department d, student s*
WHERE d.deptcode = s.deptcode

changing the join to a left outerjoin results in a mutant

*SELECT * FROM department d LEFT OUTER JOIN*
student s ON (d.deptcode = s.deptcode)

The results of the original query and the above mutant will differ if there is a department with no student, but would be identical otherwise.

The number of possible mutants of a program can be extremely large if all possible mutations are considered, but the space can be kept in control by considering mutants that reflect common programming errors.

A test case is simply a (legal) database instance, while a test suite may consist of one or more test cases.[2] A mutant query is said to be killed by a test case when the execution of the mutant on a test case produces a different result than the execution of the original query. In the above example, we need a test case with a department that has no students, in order to kill the mutant. In case the original query was incorrect, such a test case would help detect the error in the original query.

Mutants that are syntactically different may in fact be semantically equivalent to the given query. For example, under certain integrity constraints on the database, a query $r \bowtie s$, would always produce the same result as its mutant $r \sqsupset\!\bowtie s$. We say that such a mutant is *equivalent* to the original query.

A test suite for a query is said to be *complete* with respect to a space of mutations if all non-equivalent mutants in the space are killed by at least one of the test cases in the suite.

Prior work by Tuya et al. [2] and Chan et al. [3] describe techniques for generation of SQL query mutants, which are

[1]Current affiliations of first two authors are Morgan Stanley and TIBCO respectively, but this work was done while they were at IIT Bombay.

[2]Queries generated by application programs may have parameters, but for simplicity, we assume the parameters have been replaced by constants. Test cases for an application containing queries would require input parameter values in addition to a database instance; generating such test cases would require program analysis, and is beyond the scope of this paper.

then executed on the given test datasets to determine the number of killed mutants, and thereby determine the effectiveness of the given test dataset. However, neither of these papers addresses the problem of generation of test datasets. Brass and Goldberg [4] provide a rather exhaustive list of common errors in SQL queries, but do not address data generation.

Prior approaches to automated generation of datasets such as AGENDA [5] and RQP [6], generate datasets that ensure results of specified queries are non-empty. However, generating datasets in such a fashion does not ensure completeness of testing. For example, if a query used an inner join $r \bowtie s$ where it should have used a left outer join $r \sqsupset\!\!\bowtie s$, the error will be detected only if there is a test case where an r tuple does not have a matching s tuple, otherwise the two queries would generate the same result.

In this paper, we address the problem of test data generation for SQL queries, taking mutants into account. Our contributions are as follows:

- We define a space of join/outer-join mutations which models common programmer errors that are more than trivial syntactic errors considered by [2]. We also consider mutations of the SQL where-clause predicates.
- We show how to generate test cases to kill the above class of mutations, taking into account foreign key constraints. In the absence of foreign key constraints, under certain assumptions about queries, we can prove that the collection of test cases generated kills all possible mutants in the space of join/outer-join mutants that we consider. (We are working on extending the proof for the case where foreign key constraints are present.)
- Each individual test case we generate is designed to be both small and intuitive. Both these properties are very important, since ultimately a human has to examine each test case, and decide if the query result is correct for that test case. To generate realistic data, where possible we extract a small subset of an existing database to create each test case, and generate synthetic data otherwise.
  In addition, where possible we also club multiple test cases into a single database, to minimize the total data size and the number of distinct datasets to be considered.
- The algorithms described have been implemented, as part of a system for generating test data, which we call X-Data.

Although more work is required to handle all features of SQL, and to handle application programs (these are part of our ongoing work), we believe our contributions in this paper are the first step in generation of test databases in a principled way, with completeness guarantees.

II. Space of Mutants Considered

In this paper, we consider single block SQL queries with join/outer-join operations and predicates in the where clause, which correspond to select/project/join/outer-join queries in relational algebra. We do not consider insert/delete/update queries in this paper.

For such queries, we consider mutations to the join type (inner vs outer-join) and where clause predicates. We consider the following join types: inner join ($\bowtie_\theta$), left outer-join ($\sqsupset\!\!\bowtie_\theta$), right outer-join ($\bowtie\!\!\sqsubset_\theta$), and full outer-join ($\sqsupset\!\!\bowtie\!\!\sqsubset_\theta$).

Join Type Mutations of Expressions and SQL Queries: Given a relational algebra expression, the result of replacing one occurrence of a join operator ($\bowtie_\theta$, $\sqsupset\!\!\bowtie_\theta$, $\bowtie\!\!\sqsubset_\theta$, $\sqsupset\!\!\bowtie\!\!\sqsubset_\theta$) by any one of the other join operators is a join-type mutation of the expression.

An SQL query does not specify a particular evaluation plan. To allow meaningful join-type mutations to the SQL query, which reflect common programmer errors, we consider mutations of all relational algebra expressions equivalent to the given SQL query. Thus, any single join-type mutation to any relational algebra expression equivalent to the given SQL query is a single join-type mutation of the original query.

Selection Predicate Mutation: Any one occurrence of relational operator (=, <, >, <=, >=, <>) in the WHERE clause of a query is replaced with any of the other relational operators to obtain a selection predicate mutant query.

We only consider single mutations in a query at a time, although our definition of a join-type mutation goes well beyond simple syntactic errors. It is standard in mutation testing literature to consider mutants with a single change. It is possible that an erroneous query may contain multiple mistakes or variations at the same time; queries with multiple mutations are likely, but not always guaranteed, to be killed by the datasets we generate.

III. Killing Join Mutants: Basics

We make the following assumptions about the database schema and queries (a) the only constraints are primary and foreign key constraints, (b) foreign key columns are not nullable, (c) queries are single block SQL queries without nested subqueries or aggregation (however, we do allow outer join expressions), (d) all join conditions are conjunctions of equijoin conditions, (e) all selections are conjuncts of conditions of the form *attr1 op attr2* or *attr op constant*, where *op* is a comparison operation ($=, <, >, <=, >=, <>$), (f) the select clause of the query does not include functions or expressions, and (g) there are no natural join operations (natural joins complicate the presentation since they combine multiple attributes into one). Our algorithms allow a relation to occur more than once in a query, but we ignore this possibility in some cases to simplify the presentation. Our techniques can be extended to handle linear arithmetic functions, and some string functions in where clause predicates, and in the select clause, but we omit details for simplicity.

Consider an arbitrary relational algebra tree equivalent to the given query, and a single join mutation between a join and a left outer join on a single node on that tree; the node need not be the root of the expression. Let us denote the join version of the node as $L \bowtie E$ where L denotes the left input and E the right input. Suppose the mutant is not equivalent to the original tree, that is there exists some dataset where the overall expression gives different results; for that dataset, $L \bowtie_\theta E$

must differ from $L ⟕_\theta E$, in order to change the overall result. Further, for the class of select/project/join/outer-join queries that we consider, there must then be a minimal dataset, which assigns at most one tuple to each relation occurrence, which demonstrates non-equivalence. Our goal is ensure we have generated at least one such a dataset for each possible mutation.

Since the minimal dataset must differentiate $L \bowtie_\theta E$ from $L ⟕_\theta E$, it must be the case that there is an L tuple with no matching tuple in E. Suppose there is a join predicate at the node, equating $L.A$ from the left hand side to $s.B$ from the right hand side. If we ensure that for a particular tuple of L, there is no s tuple with $s.B = L.A$, then there will be a difference in the join and outer-join result at the mutated node. However, we need to do more to ensure that there is a difference in the overall result, since even if $L \bowtie E$ is not equivalent to $L ⟕ E$, it is possible that the overall expressions are equivalent.

To do so, we first generate a set of tuples, one per relation occurrence, such that the overall query has a result using this set of tuples, and then delete the s tuple from the dataset. If a relation can occur more than once in a query, we must ensure that a tuple inserted for another occurrence of a relation s does not result in the L tuple having a matching E tuple.

Unfortunately, the task of ensuring there is no matching tuple is complicated by the presence of foreign key constraints. Suppose for example there is a foreign key constraint $r.A \rightarrow s.B$ in the preceding example (or in general, a foreign key reference such that the join condition exactly matches the foreign key). If the right hand side of the mutated node consists of just the relation s, there is no dataset where $r.A$ does not have a matching tuple in s, and the original and mutated queries are equivalent.

On the other hand, suppose there is an extra selection such as $s.C = 4$ on the right hand side. Then we can create an s tuple which matches the r tuple on the foreign key reference, but which has $s.C = 5$, for example, so the selection condition is not met. In this case, the join and outer join would generate different results.

As a more complex example, if E consists of an inner join of s with another relation t, if there is no foreign key reference from s to t we could ensure there is no matching t tuple and thus E will be empty. However, if there is a foreign key reference from s to t, and the join condition is exactly on the foreign key, there is no way to generate a non-matching result for the right hand side. But if in addition, there is a selection on t, we can ensure that the result of E has no matching value for the r tuple.

Definition 1: **Join Graph**: A join graph for a query Q is a graph, with relation occurrences in the query as nodes, and an edge between two nodes if there is a join condition between the two relations, with the edge label being the join predicate. Note that it is standard in query optimization, if there are two join condition say $(r.A = s.A)$ and $(s.A = t.A)$, then we also consider $(r.A = t.A)$ as another join condition and add will add corresponding edge in the join graph.

Definition 2: **Nullable Join Expression**: Given expressions L and E we say that E is nullable with respect to L if it is possible to create a dataset where the result of L is non-empty, while $L \bowtie E$ is empty, while satisfying all foreign key constraints.

In cases where the foreign key constraints from L make it impossible to make the result of $L \bowtie E$ empty, we say that E is not nullable with respect to L.

Definition 3: **Nullable Pattern**: Given a query Q, let SQ be the set of relation occurrences in Q. We say that $S \subset SQ$ is a nullable pattern if there is a subset SL of the relations in $SQ - S$ such that E is nullable with respect to L, where L is the join of relations in SL with join conditions from the join graph of Q, and E is the inner join of relations in S, with join conditions from the join graph, and including all selections on relations in S.

If E as defined above is not nullable, then no other join expression on the set of relations S, where the join and selection conditions are a subset of those in E, will be nullable. Thus, if even E is not nullable, the pattern S need not be considered as a nullable pattern.

Definition 4: **Subsumed Join Condition**: If there is a foreign key constraint from r to s, then a condition equating a foreign key attribute in r to the corresponding referenced attribute in s is said to be *subsumed* by the foreign key constraint.

Lemma 5: Given a query Q whose set of relation occurrences is SQ, where $S \subset SQ$, and (as in the definition of nullable pattern) let E be the inner-join of relations in S, with join conditions from the join graph, and including all selections on relations in S.

Then S is a nullable pattern if and only if either E has a selection operation, or there is at least one join condition either between relations in S, or between a relation in S and one in $SQ - S$, such that the join condition is not subsumed by any foreign key constraint *referencing* a relation in S. $\square$

We omit a formal proof, but note that if none of these conditions apply, foreign key constraints will force a matching set of tuples in E. If any of these conditions apply, in the absence of foreign keys we can either create a test case where all relations from S are empty. In the presence of foreign keys, we can make some of the relations empty and/or modify attribute values so that the selection/join conditions fail, while foreign key constraints are preserved.

IV. Data Generation for Killing Mutants

The algorithm for data generation first modifies the original query by replacing all outer joins with inner joins. The query is further modified to include in the result the tuple-identifiers of all relations in the query. The results of this modified query are used to generate data. (There are cases where the modified query cannot generate an answer, e.g. if it has a *is null* condition; to handle such queries, we can modify the algorithm to consider a separate query for each pattern, but omit details for lack of space.) For a particular result tuple t of the modified query, let $t.R_i$ denote the tuple from R_i, the

ith relation occurrence in the query, that joined with tuples of other relations to form t.

The algorithm maintains for each relation a set of included tuples, and a set of exclusion predicates of the form $(a_{i1}, a_{i2}, \ldots, a_{in}) \neq (v_1, v_2, \ldots, v_n)$. The exclusion predicates allow us to handle multiple occurrences of a relation, and (in some cases, as we shall see) to combine datasets for multiple patterns into a single dataset, thereby reducing the effort for testing.

Our algorithm creates a dataset corresponding to each nullable pattern. In some cases, such as if the initial query uses only inner joins, and there are no foreign key constraints, we need create datasets only for patterns consisting of a single relation. However, if the original query had outerjoins, there are cases where we need to consider nullable patterns with multiple relations; and in order to consider mutations with multiple joins converted to outerjoins, we need to consider all nullable patterns. We leave the issue of minimization of the set of patterns, and correspondingly the number of datasets generated, to future work.

For a pattern, if the foreign key constraints permit the tuples for the relations in the pattern to be excluded, the following steps are taken. A tuple t is found from the query result such that the following constraints are satisfied for each relation occurrence R_i in the query (if no such tuple exists, it is generated synthetically, using techniques similar to those in [6]):

1) If R_i is not in pattern, $t.R_i$ has not been excluded by an exclusion predicate; to handle foreign key constraints, we also need to ensure that any tuples referenced from $t.R_i$ have not been excluded by an exclusion predicate.
2) If R_i is in the pattern, ensure that $t.R_i$ has not been included.

If the above test is satisfied, for each R_i not in the pattern, include $t.R_i$; for each R_i in the pattern, for each join condition with another relation R_j not in the pattern, add an exclusion predicate on the join attributes JA, of the form $(JA) \neq t.R_i[JA]$. This ensures that no tuple will get added to R_i that matches $t.R_j$ on the join conditions.

If foreign key constraints from relations outside the pattern to relations in the pattern, combined with the join conditions prevent exclusion of tuples for all the relations in the pattern, the tuples in the pattern are modified so that the selection and non-subsumed join conditions are not satisfied, while the foreign key and primary constraints are satisfied. Exclusion predicates are also added to ensure no other tuples are added subsequently that can satisfy the selection/non-subsumed join conditions. We omit details for lack of space.

In the full version of the paper we show that any non-equivalent mutation in the space of join-type mutations defined in Section II is killed by our data generation algorithm, in the absence of foreign-key constraints. Intuitively, given any join tree, and a particular node of the join tree, there is a dataset that nullifies only (one or more of) the relations on one input of the node; if a mutation at the node is a non-equivalent mutation, this dataset will kill the mutation. Foreign key constraints complicate the proof; we conjecture completeness in the presence of foreign-key constraints.

To kill comparison operator mutants for a predicate of the form $Aopv$, where op is a comparison operation, we generate three different tuples, with A value less than, equal to, and greater than v. When generating data from an existing database, we take an existing tuple and modify its A value as above.

Our algorithms do not require that any specific attributes be included in the query result projection. However, in case the query result contains attributes that form a super key for one of the input relations, and these attributes are guaranteed to be non-null, then instead of creating a different dataset for each pattern, we can create a single dataset for all patterns. We omit details for brevity.

V. Implementation and Experimental Results

Our algorithms were implemented using Java as the programming language, and PostgreSQL as the database. Modification of tuples based on selection and non-subsumed join conditions, and generation of synthetic data are ongoing tasks.

We ran our algorithms on sample queries with three or four relations on a sample schema. Execution times for generating datasets were within 10 to 15 seconds on a database with about 50,000 to 200,000 records on a x86 machine with 2GB main memory and a 2GHz processor. Each generated test case was, as expected, very small, and the merged dataset killed all non-equivalent mutants of the sample queries.

VI. Conclusions and Future Work

Our work is merely a first step in test data generation for killing query mutants. We are currently extending our techniques to handle more SQL features, such as aggregation and nested subqueries, to minimize the number of datasets generated, and to allow certain kinds of expressions in select/where clauses. We are also currently completing the implementation of tuple modification to handle foreign-key constraints, and working on a proof of completeness with foreign-key constraints. Test data generation for application programs containing SQL queries is an important area of future work.

References

[1] R. DeMillo, R. Lipton, and F. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, April 1978.

[2] C. d. l. R. Javier Tuya, M Jose Suarez-Cabal, "Mutating database queries," *Information and Software Technology*, vol. 49, no. 4, pp. 398–417, 2007.

[3] W. K. Chan, S. C. Cheung, and T. H. Tse, "Fault-based testing of database application programs with conceptual data model," in *Int'l Conf. on Quality Software (QSIC)*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 187–196.

[4] S. Brass and C. Goldberg, "Semantic errors in SQL queries: a quite complete list," in *Int'l Conf. on Quality Software (QSIC)*, 2004, pp. 250–257.

[5] D. Chays, Y. Deng, P. G. Frankl, S. Dan, F. I. Vokolos, and E. J. Weyuker, "An AGENDA for testing relational database applications." *Software Testing, Verification and Reliability*, vol. 14, no. 1, pp. 17–44, 2004.

[6] C. Binnig, D. Kossmann, and E. Lo., "Reverse query processing." in *Inte'l Conf. on Data Engineering (ICDE)*, 2007, pp. 506–515.