Generating Test Data for Killing SQL Mutants: A Constraint-based Approach

Shetal Shah, S. Sudarshan, Suhas Kajbaje, Sandeep Patidar, Bhanu Pratap Gupta, Devang Vira
Computer Science and Engg. Dept., Indian Institute of Technology, Bombay
{*shetals,sudarsha,suhask,sandeepp*}*@cse.iitb.ac.in, bhanupratap2006@gmail.com, devang.vira@gmail.com*

***Abstract*—Complex SQL queries are widely used today, but it is rather difficult to check if a complex query has been written correctly. Formal verification based on comparing a specification with an implementation is not applicable, since SQL queries are essentially a specification without any implementation. Queries are usually checked by running them on sample datasets and checking that the correct result is returned; there is no guarantee that all possible errors are detected.**

In this paper, we address the problem of test data generation for checking correctness of SQL queries, based on the query mutation approach for modeling errors. Our presentation focuses in particular on a class of join/outer-join mutations, comparison operator mutations, and aggregation operation mutations, 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. The number of datasets generated, is linear in the size of the query, although the number of mutations in the class we consider is exponential. Under certain assumptions on constraints and query constructs, the test suite we generate is complete for a subclass of mutations that we define, i.e., it kills all non-equivalent mutations in this subclass.

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 separate implementation. In practical settings, programmers execute the query on each of multiple test cases, and check that the query gives the desired results on each test case. However, if test cases are created in an ad-hoc manner, it is not clear whether 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 (see e.g., [23]). Mutation testing involves generating mutants of the original program by modifying the program in a controlled manner [12]. 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.

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 and faulty programs. A test case that gives different results on the given program and the mutant program is said to *kill* the mutant.

The notion of mutations can also be applied to database queries. Query mutations which model common mistakes made by programmers when specifying queries include 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 (⟕), missing joins conditions, etc. For example, given the query

*SELECT * FROM instructor i, teaches t*
WHERE i.id = t.id

changing the join to a left outer join results in a mutant

*SELECT * FROM instructor i LEFT OUTER JOIN*
teaches t ON (i.id = t.id)

The results of the original query and the above mutant will differ if there is an instructor not teaching a course, but would be identical otherwise.

The number of possible mutants of a query 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.[1] 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, a test case containing an instructor who does not teach any course would kill the join/left-outer-join mutant.

The tester must examine the test case to figure out what the intended output is on that test case, and check if the given query gives the intended output. In case the original query was incorrect, and a mutant was in fact the correct query, a test case that kills that mutant would help the tester detect the error in the original query, since the given query would generate a different result from the intended query on that dataset. A single test case may kill a large number of mutants, and the tester need not even be aware of what mutants are killed by the test case.

Mutants that are syntactically different may in fact be semantically equivalent to the given query constraints on the database. For example, in the presence of appropriate foreign key constraints on the database, a query $r \bowtie s$, would always

[1] 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.

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 on mutation testing in databases, such as [16], has focussed on checking the completeness of a given test suite. Prior approaches to automated generation of datasets such as AGENDA [10], RQP [3] and Olston et al [24], generate datasets which ensure that the results of specified queries are non-empty. However, they do not address mutations, and there is no guarantee of completeness of the test cases generated.

In this paper, we address the problem of test data generation for SQL queries, taking mutants into account. In a shorter version of this paper, [14], we introduced the problem of test data generation to kill SQL mutants, and sketched an approach to generating test data. In this paper, we extend the results of our earlier paper, and describe a new implementation approach which is more powerful than our earlier approach and handles a larger class of queries.

The contributions of this paper are as follows:

- We define a space of mutations which is based on mutations to a query tree, but allows the consideration of all possible join orders for inner joins, which are specified in a join-order independent fashion in SQL. We outline how to handle mutations to join type (i.e., mutations between inner join, and the left, right and full outer joins), comparison operations, and unconstrained aggregation operations; however our approach, based on using a constraint solver, makes it possible to add support for other mutation types.
- We show that the decision version of the test data generation problem is NP Hard in the size of the query. However special cases without repeated occurrences of relations can be solved in polynomial time.
- We give an algorithm which identifies constraints that need to be satisfied by a test dataset to kill a particular group of mutants. As in [3], each attribute of each tuple in the output dataset is treated as a variable, and the constraints are on these variables. We then use a constraint solver, (in our current implementation, CVC3 [2]) to generate the required dataset.
- SQL queries allow relations to be listed in a join-order independent way in the FROM clause. To model common query errors, we need to consider mutations to every possible join tree corresponding to the given relations in the FROM clause. Unfortunately, the number of such join trees is very large, with an exponential number of possible join operations to be considered. Creating such a large number of datasets would make testing infeasible. Instead, we show how to generate a number of datasets that is linear in the size of the query, yet can kill all these exponential number of mutants.
- Each individual test case we generate is designed to be small and intuitive, which is important since ultimately a human has to examine each test case, and decide if the query result is correct for that test case.
- The algorithms described have been implemented, as part of a system for generating test data, which we call X-Data. We present a preliminary study of the effectiveness of our algorithms.

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 an important first step in generation of test databases in a principled way, with completeness guarantees.

II. Space of Queries and Mutants Considered

In this paper, we consider the following class of queries: single block SQL queries with join/outer-join operations and predicates in the where-clause, and optionally aggregate operations without a having clause, which correspond to select/project/join/outer-join queries in relational algebra, with an aggregation operation optionally on top. We do not consider insert/delete/update queries in this paper.

We consider mutations to the join type (inner vs outer-join), comparison operations, and aggregation operations, as outlined below. This class covers many of the common errors that we have seen in courses we teach and some that we have seen during the development of application software used in IITB. Our constraint-solver based approach can be used to handle other types of mutations by defining appropriate constraints.

Join Type Mutations: 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 ($\sqsupset\!\!\bowtie\!\!\sqsubset_\theta$). 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 programming errors, we consider mutations of all equivalent join trees that can be derived from the relations in the FROM clause. The space of equivalent join trees also takes into account equivalence classes of attributes, described later in Example 4 in Section IV-B. 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. In addition, SQL queries do not specify the placement of selections. We push selections down to the lowest level possible, namely the individual relations; pulling a selection up to a higher level of the join tree would make some mutations at a lower level equivalent, and thus pushing selections down maximizes the number of mutations that can be killed. Similarly, join predicates are assumed to be applied at the earliest possible point in the tree.

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

Aggregation Operator Mutation: The aggregate operators considered are MAX, MIN, SUM, AVG, COUNT, SUM

(DISTINCT), AVG (DISTINCT) and COUNT(DISTINCT) [2]. We consider mutations of any one of the aggregate operators to any one of the others. An aggregation operator can occur in the select clause of the query or in the having clause; the latter case results in a constraint on the aggregation result, complicating the generation of data. In this paper we do not consider the case of constrained aggregation; handling constrained aggregation is a topic of future work.

We only consider single mutations in a query at a time, as is common in the mutation testing literature. 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.

For the rest of this paper,we make the following assumptions about the database schema and queries:

A1 The only constraints are primary and foreign key constraints.

A2 Foreign key columns are not nullable.

A3 Queries are single block SQL queries without nested subqueries (however, we do allow outer join expressions).

A4 Queries do not include functions or expressions other than simple arithmetic expressions.

A5 Join and selection predicates are conjunctions of simple conditions of the form *expr relop expr*.

A6 Queries do not explicitly check for null values, using the SQL clause IS NULL. This restriction is because our constraint solver does not handle null values.

A7 If the query has a full outerjoin, at least one attribute from each of its inputs is present in the select clause. This restriction helps to ensure that the difference due to a mutation in one of the inputs will be seen in the query result.

A8 If the query has a natural full outerjoin, at least one attribute from each of its inputs other than the common (join) attributes is present in the select clause. (This extension of Assumption A7 is required since the natural outerjoin operation replaces common attributes by a single output attribute, whose value may come from only one of the inputs; this masks the difference between the case where the other input is an empty set and the case where it has a matching value.)

In Section V-H we discuss how to extend our algorithms to partially relax Assumptions A2 and A3.

III. Related Work

The AGENDA toolset for testing database applications [10] has several components, including a component for generating a test database for a given query. However, [10] does not address mutation testing, and does not provide any guarantees about completeness of the test data that it generates.

Reverse Query Processing (RQP) [3] takes a query and an intended result as input and returns a possible database instance, respecting integrity constraints, that could have produced that result for that query; [4] extends RQP to handle multiple queries. The primary goal of RQP is to generate test datasets which result in non-empty result-sets on the execution of a given query. It considers a subset of SQL/relational algebra which includes join, selection, projection and aggregation operations. However, RQP neither considers query mutations nor outer join operations. Olston et al. [24] take a dataflow program and a database and generate an example dataset such that the result of each operator (including intermediate operators) in the program is non-empty. However, they do not handle integrity constraints or consider query mutations.

C. de la Riva et al. [11] show how to generate test cases to kill SQL query mutants, generating constraints based on SQL coverage rules ([28]) and solving them using a constraint solver called Alloy. Our work was done independently of theirs. While we do not handle certain features that they support, such explicit support for NULL values, their mutation space does not consider equivalent join orders, and they do not provide any completeness guarantees.

Mutant killing is quite closely related to query containment. If Q and mutant Q' are not equivalent, then either Q is not contained in Q', or Q' is not contained in Q. Many of the algorithms for testing query containment actually generate a dataset as evidence of non-containment, and could thus potentially be used to generated datasets for killing mutants. [9] and [1] address conjunctive queries, giving an algorithm that generates a test dataset respecting primary key constraints, and show NP hardness of the problem in general. The algorithm was extended by [25], which considers union and difference operations, [17], which considers inclusion dependencies (foreign key constraints) but only for conjunctive queries without difference, and [19] which considers order constraints for conjunctive queries. None of these algorithms consider the outerjoin or aggregation operations. Further, generating separate datasets for mutation could potentially result in a infeasibly large number of test cases, and as a result we do not use these algorithms. Equivalence of outerjoin queries is addressed by [20], but their approach does not construct datasets.

Other work on test data generation include [15], [6], [29], and [21], which address the issue of performance testing rather than correctness testing of applications, and [22], which addresses schema validation; these papers do not consider specific queries when generating data, and give no guarantees about non-empty query results or about killing mutants.

Emmi et al. [13] describe an approach to test applications based on creation of database states and test inputs, which can ensure path coverage. Kapfhammer and Soffa [18] similarly consider test adequacy of database driven applications. However, neither of these papers address the problem of testing of queries, nor of mutations. Chan and Cheung [7] present an approach for test input generation for SQL queries, but do not consider mutations.

Tuya et al. [16] and Chan et al. [8] describe techniques for generation of SQL query mutants, which are then executed

[2] We have not considered the case of DISTINCT except in aggregation operations. Mutations which affect the duplicate count are not in the class of mutations we consider, and are an area of future work.

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 address the problem of generation of test datasets, nor do they consider alternative join orders for mutation. Brass and Goldberg [5] provide a rather comprehensive list of semantic errors in SQL queries, but do not address data generation.

This paper is an extension of our earlier short paper [14]. The novel contributions of this paper include the following: (a) handling complex selections, non-equi-join conditions and aggregates, (b) showing the NP hardness of the data generation problem, (c) a new algorithm for dataset generation which generates a linear number of datasets unlike the exponential number in [14], and handles foreign keys constraints with completeness guarantees, and (e) generation of synthetic data using a constraint solver ([14] did not provide any way to generate synthetic data for queries with constraints).

IV. Killing Join Mutants: Basics

In this section we discuss several issues that need to be addressed when killing mutations, and later in Section V we present our algorithm for generating datasets to kill mutations.

A. NP-Hardness of the Data Generation Problem

The problem of generating a single dataset to kill one mutation is NP-Hard, in general. Consider a SQL query Q involving a mix of inner, left outer joins, right outer and full outer joins. Let Q' be a single mutant of Q. Then the data generation problem for a single mutation can be stated as: *Is there an assignment of tuples to each relation in Q such that the results of Q and Q' differ?*

Proof of NP-Hardness: To prove that the decision version of the data generation problem is NP-Hard, we reduce the well known NP-Complete problem, Query Containment Problem for Conjunctive Queries[9], to this problem. The Query Containment Problem states : Given two SQL queries Q_1 and Q_2, is Q_2 contained in Q_1?

The reduction is as follows. Construct a tree $Q_2 \Join Q_1$ and a mutation $Q_2 ⟕ Q_1$ as the input to the Data Generation Problem stated above. It is easy to see that, if on this input, the algorithm for the Data Generation Problem assigns tuples to the relations in Q_1 and Q_2 such the results of $Q_2 \Join Q_1$ and $Q_2 ⟕ Q_1$ differ then Q_2 is *not* contained in Q_1. On the other hand, if there is no such assignment, then it implies that Q_2 is contained in Q_1. □

This implies that, assuming $P \neq NP$, there is no polynomial time algorithm to generate data for even *one* mutation for a general SQL query. Note that NP-Hardness of the Query Containment Problem is basically due to repeated occurrences in the query. Versions of the problem without repeated occurrences are known to have polynomial time solutions.

B. Issues in Killing Mutations

In this section, we give examples illustrating the issues faced in killing mutations.

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, W, as $L \Join E$ where L denotes the left input and E the right input. Our goal is to ensure that there is a test case that can kill that mutation by producing different results on the original query and on the mutant.

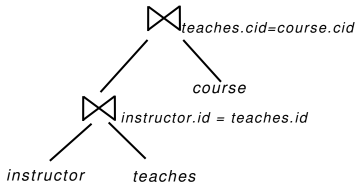


Fig. 1. Query Tree for instructor ⋈ teaches ⋈ course

To ensure that there is a difference between the inner join and the left outer join result, we must ensure that there is a case where the inner join condition fails, while the left outer join allows a result at that (internal) node of the tree. More generally, for each mutation, we need to create a dataset that *distinguishes* the mutation at a node by generating different results for the original query and the mutation. Our goal is ensure we have generated at least one such dataset for each possible mutation.

Since the dataset must differentiate $L \Join_\theta E$ from $L ⟕_\theta E$, it must be the case that there is an L tuple with no matching tuple in E. As long as the query does not have repeated occurrences of any relation, and there are no foreign key constraints, we can ensure the above condition by creating datasets where one of the relations in E is empty in each of the datasets; by doing so, for at least one dataset the result at the node will differ based on whether an inner join or an left outer join is used (if E has only inner joins, every dataset above would exhibit the difference).

However, in the presence of foreign key constraints or repeated occurrences of a relation, it may not be possible to create a dataset with an empty relation that differentiates the mutation. Also, repeated relational occurrences and aggregation operations may force us to have more than one tuple for a relation. To handle this general case, we require that the dataset satisfies

$$\exists_{l \in L} \neg \exists_{e \in E} \ \theta(l, e)$$

i.e., there exists a tuple in L for which there does not exist any matching tuple in E. However, in some cases it may not be possible to create such datasets, due to the presence of foreign key constraints (corresponding to an equivalent mutation).

Further, if the mutated node is not the root of the query tree, the effect of the mutation may be masked by an intermediate node. Wherever possible, we need to create a dataset where the difference due to the mutation is *propagated* to the root of the query tree, causing a change in the query result.

We illustrate these issues with the help of examples below. **Example 1**: Consider the query tree given in Figure 1, and consider a mutation that replaces *instructor* ⋈ *teaches* by *instructor* ⟖ *teaches*. In the absence of foreign keys, we can create a dataset where a *teaches* tuple does not have any matching *instructor* tuple. To propagate this difference to the result, the dataset must contain a *course* tuple that matches the *teaches* tuple.

Example 2: Consider again the mutation *instructor* ⟖ *teaches*, but now with a foreign key from $teaches.id$ to $instructor.id$. Assuming $teaches.id$ is not nullable, it is not possible to generate data where the *teaches* tuple does not have a matching *instructor* tuple. This mutation is equivalent to the original query.

On the other hand, suppose that *instructor* is replaced with

$\sigma_{instructor.dept=\mathrm{CS}}(instructor)$

Then we can create an *instructor* tuple which matches the *teaches* tuple on the foreign key reference, but which has, for example, $dept =$ Biology, so the selection condition is not met. Thus, we ensure that the join operation has a right input without any matching left input, and thus the join and right outer join would generate different results.

Example 3: Now suppose that the inner join between *instructor* and *teaches* is mutated to a left outer join. To obtain a difference in the query results, between the original query and the mutant, we need a tuple in the *instructor* relation which does not match with any tuple in the *teaches* relation. Unfortunately, this tuple will be filtered at the root, as it will not have any relevant *teaches.courseId*.

In this case, we can see that though it is possible to ensure that there is a difference at the mutated node, this difference will not be visible at the output of the root node. In fact, it can be seen that this mutation is equivalent to the original query.

Example 4: Consider the query Q,

```
SELECT *
FROM A,B,C
WHERE  A.x = B.x  AND  B.x = C.x;
```

The join condition could have equivalently been written as $A.x = B.x$ AND $A.x = C.x$. SQL programmers do not care about join orders, and similarly they do not care about which attributes are specifically equated; thus whether a query condition is written as $A.x = B.x$ AND $B.x = C.x$ or as $A.x = B.x$ AND $A.x = C.x$, it should not affect the set of mutations that are killed. However, consider Figure 2, where (a) shows the first form, and (b) shows the second form; the join order of (b) can be reordered to get (c), although the join order of (a) cannot be reordered to (c). Suppose the intended query was in fact (d), which is a mutation of (c). Then whether the query was written in form (a) or (b), we should be able to generate a dataset that kills mutation (d).

To ensure this, we map the join conditions in both forms to a common *equivalence class* representation; in the above example the equivalence class contains the attributes $A.x$, $B.x$ and $C.x$. Our data generation algorithms work directly on this representation.

V. Data Generation Algorithm

In this section, we present our algorithm for dataset generation. Our algorithm generates at most one dataset for each attribute that participates in a join condition, and at most 3 datasets for each selection condition conjunct, and aggregation operation and hence the number of datasets is linear in the size of the query.

The dataset generation algorithm has two main steps:

1) The first step involves the generation of a set of constraints that tuples in a dataset should satisfy to ensure a difference between the results of the original query and its mutations. These constraints are on the attributes of tuples of individual relations involved in the query.

2) In the second step, the constraints are used to generate datasets. We use a constraint solver (in our implementation, CVC3 [2]) to get an assignment of values to variables representing attributes of tuples, and thereby generate a dataset.

Although the idea of using constraints to generate datasets is similar to RQP [3], the specific constraints generated are completely different since the goals of the two systems are different.

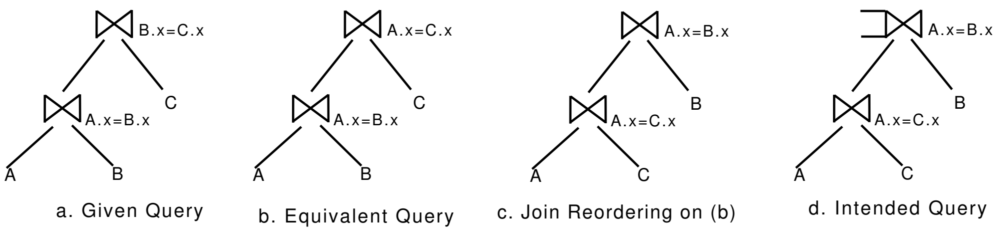
A. Generating Constraints to Kill Join Mutations

The input to our algorithm is a query tree T consisting of join nodes, selection predicates, and aggregation nodes, with relations as leaves.

Consider a join node W of T. Let L be the left child and E be the right child of W. Let the join predicate at W be θ. Without loss of generality, assume that W is an inner join, and consider its mutation to a left outer join. Then, from our discussion in Section IV, to kill this mutation, we need a dataset satisfying a *not-exists* constraint of the form $\exists_{l \in L}\ \neg\exists_{e \in E}\ \theta(l, e)$, i.e., in that dataset there exists a tuple in L for which there does not exist any matching tuple in E. Now consider the mutation of W to a right outer join. Then, to kill the mutation, we symmetrically require a dataset satisfying $\exists_{e \in E}\ \neg\exists_{l \in L}\ \theta(l, e)$, to kill the right outer join mutant at W. Each of these datasets will also kill the mutation of an inner join to a full outer join.

Now assume that W is a left outer join and consider its mutation to an inner join. Here too, a dataset satisfying $\exists_{l \in L}\ \neg\exists_{e \in E}\ \theta(l, e)$, is sufficient to kill the mutation. Note that in this case, the original query will have a non-empty result at W whereas the result of the mutant will be empty. Similarly a mutation to a right outer join can be killed with the dataset satisfying $\exists_{e \in E}\ \neg\exists_{l \in L}\ \theta(l, e)$. This implies that irrespective of the type of join node, we can generate two (specific) datasets to kill all 3 mutations of the node.

Unfortunately, it is not possible to give a constraint of the above form directly to a constraint solver, since the constraints must be in terms of attributes of tuples that are generated, and not on results of an expression. Specifically, to specify the constraints in a form that a solver such as CVC3 can handle, we create a tuple of variables (with one variable per attribute) corresponding to each tuple to be generated in the dataset. Constraints such as join and selection conditions, as well as not-exists constraints described above, have to be translated into constraints on these variables. A key problem is to translate such a high level requirement into constraints on individual tuples.



a. Given Query b. Equivalent Query c. Join Reordering on (b) d. Intended Query

Fig. 2. Motivation for Equivalence Class Formation

The second problem that needs to be addressed lies in ensuring that the difference exposed at node W is propagated to the result of the overall query, wherever possible. To address this problem, we ensure that all join and selection conditions in the query, other than those that are specifically violated in a dataset, are satisfied by adding corresponding constraints.

A third problem is that if we generate a separate dataset for each possible L and E in each possible join tree we would get an exponential number of datasets. Instead, we generate only a small number of datasets and show that they kill all join type mutations of all join trees.

A fourth problem that we need to address is repeated relation occurrences in a query. To handle these, we assume that each occurrence has been given a distinct name so they can be distinguished; however, we also record the corresponding base relation which is required for generating constraints. Further, we create a separate CVC3 tuple for each occurrence, in an array of tuples corresponding to the base relation. Thus, if relation R occurs twice in a query, we create an array of 2 R tuples in CVC3, and map one occurrence to $R[1]$ and the other to $R[2]$. The same scheme also allows us to generate more than one tuple for a particular relation occurrence, by adding further elements to the array for R.

To enforce a condition such as $\neg\exists_{r \in R}\ r.A = 5$, we create a constraint in CVC3 of the form $\neg\exists_i\ R[i].A = 5$. CVC3 does not understand attribute names, and instead uses positional notation, so in reality we have to use a constraint of the form $\neg\exists_i\ R[i].0 = 5$, if A is the very first attribute of R (or $R[i].1$ if A is the second attribute, and so on), but we sometimes use the attribute name in our presentation, for simplicity.

Once constraints are generated for a particular dataset, we invoke CVC3 to create a model satisfying the constraints. In case the constraints are inconsistent, for example, because of a foreign key constraint conflicting with a not-exists constraint, the dataset will not get created; such cases arise only when the targeted class of mutants is actually equivalent to the given query.

B. Overall Algorithm

The overall algorithm for dataset generation is shown in Algorithm 1. As the very first step of the algorithm, the following preprocessing steps are carried out:

1) Build equivalence classes of the relation attributes involved in equi-join conditions.
2) Drop all equi-join predicates from the list of predicates in the query, since they are now implicitly represented by the equivalence classes. All other predicates, including join predicates other than equijoin conditions, and selection predicates, are retained.
3) Build a closure of foreign key relationships in the schema. If we have foreign key relationships $A.x \rightarrow B.x$ and $B.x \rightarrow C.x$, then the closure also contains $A.x \rightarrow C.x$.

Algorithm 1 : Main Algorithm

```
Hashtable $currentIndex$; /* Maps distinct relation names
  to offsets in the CVC3 array created for the corresponding
  database relation */
procedure generateDataSet(query $q$)
    preprocess query tree
    initializeIndices() /* Initializes $currentIndex$
        and other related structures */
    generateDataSetForOriginalQuery()
    killEquivalenceClasses()
    killOtherPredicates()
    killComparisonOperators()
    killAggregates()
end procedure
```

The next step initializes the mapping from distinct relation names to indices into the array of tuples of the corresponding database relation.

The following step of Algorithm 1 generates data that ensures a non-empty result for the given query (using the procedure generateDataSetForOriginalQuery()) so that the user sees at least one test case giving a non-empty result. Also, if there are query mutants whose results are empty on all legal datasets, this dataset guarantees that such mutants are killed. We outline procedure generateDataSetForOriginalQuery()) below. The rest of Algorithm 1 generates datasets to kill different kinds of mutations, and these procedures are discussed in subsequent sections.

The procedure generateDataSetForOriginalQuery() first creates one constraint tuple for each occurrence of a relation; the tuple contains one variable for each attribute of the relation. Next, the procedure adds constraints corresponding to each of the predicates in the query, including equi-join predicates (represented by equivalence classes), and other join and selection predicates.

Finally, constraints corresponding to primary and foreign key constraints in the database are added by the procedure genDBConstraints(). To ensure primary key constraints, a corresponding functional dependency constraint is created; for example, if R.A is a primary key, the constraint ensures that if for any i, j, R[i].A=R[j].A, then the remaining attribute values are also equal[3] (if we instead constrained R[i].A to not equal any other R[j].A, the solver would not be able to create a case where a relation contains only one tuple even though it occurs twice in a query).

The foreign key constraints ensures that the values assigned to the foreign key are a subset of those assigned to the primary key. Consider a foreign key $R.A$ referencing $S.B$. One way to implement foreign key constraints is to create a constraint of the form $\forall_i \exists_j (R[i].A = S[j].B)$; actually the quantifiers range over the number of tuples created for R and S, so a related problem is how many tuples to create for R and S. There are cases where to kill a mutation, we require a tuple of $S.B$ with no referencing $R.A$. In this case, we need to ensure that S has at least two tuples, one with a matching $R.A$ and one without any matching $R.A$. Therefore whenever we create an $S[i]$ tuple with no matching tuple in the referencing relation R, we also create a constraint tuple $S[j]$ for each tuple $R[k]$ in the referencing relation R, with a constraint forcing $S[j].B = R[k].A$, to enforce the foreign key constraint. This is in addition to tuples created to ensure the query result is not empty, or to kill mutants. (The constraint solver may of course make these tuples equal, and we eliminate duplicates before creating a dataset in the database if the relation has primary key constraints.)

Note that if there is a foreign key constraint from a relation involved in the given query Q, referencing a relation which is not included in Q, then we also generate test data for the referenced relations (and so on transitively), so that the integrity constraints of the database schema are maintained.

The genDBConstraints() procedure also adds domain constraints, to ensure that values for an attribute are generated from the domain of that attribute; we can for example specify the domain to be an integer, or enumerate data values to be used for that domain.

The following helper functions are used in procedures that we will see subsequently.

1) **cvcMap()**: This function cvcMap($rel.attr$) takes a distinct relation name rel and attribute name $attr$, and returns a string $r[i].pos$, where r is the base relation (i.e., rel is a renaming of r), pos is the position of attribute $attr$ in r (since CVC3 uses positional notation), and i is the index in array r corresponding to rel which is obtained from the $currentIndex$ map. In general, we may create more than one tuple in r for a given distinct name rel, for example to create multiple top-level tuples. However, at any point when cvcMap is called, there is a current index value, which is used; this value may be updated when more tuples are created. We omit details. Similarly, there is an overloaded function cvcMap($Pred$) which takes a predicate P and returns the predicate with each of its relation attributes $rel.attr$ translated to the CVC3 array form, by calling cvcMap($rel.attr$). We omit details.
2) **generateEqConds()**: The function generateEqConds(P) generates equality constraints for all elements of equivalence class P. The function simply creates a conjunction of equality conditions, with the ith condition equating member i and $i+1$ of equivalence class P (we assume members are ordered in some fashion). So that the generated constraints are in CVC3 form, this function calls cvcMap() on the elements.

C. Killing Equi-Join Condition Mutations

We next create datasets to kill join type (i.e., join/outerjoin) mutations. Consider any node W of any join tree with children L and E, and suppose predicate p is a conjunct of the join condition at the node. If p is an equijoin condition, it must equate attributes from the two children of the node. Our goal is to ensure there is a dataset where there is a tuple from L that does not have any matching tuple from E and vice versa. To do so, we pick the relations participating in p, and for each of the relations r_i we ensure there is a dataset with a tuple for each of the remaining relations in p which together satisfy the join and selection conditions, but there is no tuple in r_i matching these other tuples on condition p. We say that relation r_i has been **nullified** on condition p by this dataset. It should be clear that if we can do so for each join predicate conjunct p and relation r_i, then we would have handled all join trees regardless of the join order.

The above task must be carried out on equivalence classes as well as other join conditions. The task must also be applied to selections, where it ensures that there is a dataset with no tuple satisfying the selection condition, which can be required to kill mutations as outlined in Example 2 in Section IV. We consider these cases in this and subsequent sections.

Procedure killEquivalenceClass(), shown in Algorithm 2, handles the case of equi-join conditions represented by equivalence classes. Consider an equivalence class containing $A.x, B.x$ and $C.x$. Suppose that there is foreign key relationship from $A.x$ to $B.x$. Now, it is not possible to create a dataset where there is an $A.x$ tuple with no matching $B.x$ value. Hence, we do the following: whenever we need to create a dataset where $B.x$ is to be nullified on a join predicate linking it to a referencing relation, we also nullify the referencing foreign keys ($A.x$ in the above example). Specifically, we split the equivalence class ec into two parts: S which contains $B.x$ and all its referencing foreign keys, and P containing the rest of the elements: $ec - S$.

We then *jointly nullify* the set of elements given in S, i.e., we ensure that there are tuples for each element in P that match each other on the join attributes, but there is no tuple in any relation in S that has a value matching the P tuples. As a special case, if P is an empty set, we do not generate a dataset as this corresponds to an equivalent mutation.

Instead of adding relations referencing $B.x$ and then nullifying them all together, we could have just skipped creating a dataset for this case. Doing so would in fact be fine if the given query had only inner joins. However, if the query can

[3] For readers familiar with query containment, this constraint implements the chase algorithm.

Algorithm 2 : killEquivalenceClasses()

```
for each equivalence class $ec$ do
    Let $allRelations$ := Set of all $\langle rel, attr \rangle$ pairs in $ec$
    for each element $e$ in $allRelations$ do
        $conds$ := empty set
        Let $e$ := $R.a$
        $S$ := (set of elements in $ec$ which are foreign keys
            referencing $R.a$ directly or indirectly) UNION $R.a$
        $P$ := $ec$ - $S$
        if $P$.isEmpty() then
            continue
        end if
        $conds$.add(generateEqConds($P$))
        $conds$.add(
                "NOT EXISTS i: R[i].a = " + cvcMap(P[0]))
        for all other equivalence classes $oec$ do
            $conds$.add(generateEqConds($oec$))
        end for
        for each other predicate $p$ do
            $conds$.add(cvcMap($p$))
        end for
        $conds$.add(genDBConstraints())
        callSolver($conds$)
        if solution exists then
            create a dataset from solver output
        end if
    end for
end for
```

Algorithm 3 : killOtherPredicates()

```
for each predicate $p$ do
    Let $allRelations$ := Set of all relations in $p$
    for each relation $r$ in $allRelations$ do
        $conds$ := empty set
        $conds$.add(genNotExists($p$, $r$))
        for each equivalence class $ec$ do
            $conds$.add(generateEqConds($ec$))
        end for
        for all other predicates $p$ do
            $conds$.add(cvcMap($p$))
        end for
        $conds$.add(genDBConstraints())
        callSolver($conds$)
        if solution exists then
            create a dataset from solver output
        end if
    end for
end for
```

include outer joins, there could be a case where we would not kill a mutant. To see why, suppose $A.x$ references $B.x$, and the given query is $(C ⟕ A) \bowtie B$. Now, if we skip nullifying B, we may not generate a dataset that kills the mutation of the root to a ⟕; nullifying just A does not help, since that dataset would have an B tuple matching each C tuple, but nullifying B and A together ensures there is a dataset where the C tuple does not have a matching B tuple.

D. Killing Join Mutations with Non-Equijoin Conditions and Selections

Procedure killOtherPredicates() shown in Algorithm 3 generates datasets to kill join type mutations considering non-equi join conditions, and selection predicates.

Consider a non-equi join condition involving a set of relations S. The procedure generates several datasets each of which *nullifies* one of the relations $s \in S$ with respect to the other relations.

The function killOtherPredicates() calls the function genNotExists(P,R), which takes a predicate and a relation, and generates a not-exists constraint of the form $\neg\exists_{r \in R} s.t.\ P$. The function genNotExists() is called on each relation R participating in P. For example, suppose we have a join condition $B.x = C.x + 10$. We would then generate two datasets which include the following not-exists constraints (B_INT and C_INT denote the range of array indices for B and C respectively, and we assume that the current index of B and C are both 1):

1) Dataset 1 (nullifying $B.x$):

```
ASSERT NOT EXISTS (i : B_INT) :
            (B[i].0 = C[1].0 + 10);
```

2) Dataset 2 (nullifying $C.x$):

```
ASSERT NOT EXISTS (i : C_INT) :
            (B[1].0 = C[i].0 + 10);
```

Consider any tree T where the join condition occurs in node W. The not-exists constraints above ensure that the datasets differentiate between inner and outer-joins at W. The remaining constraints added by the procedure ensure that the concerned tuples reach the node W, and the result, if any, of W is propagated to the root of T. To do so, we generate constraints that ensure that all other join conditions evaluate to true.

Similarly, if we had a selection condition on B, we would generate a dataset where no tuple in B satisfies the selection condition. Ensuring selection conditions are not satisfied is required to kill join mutations in some cases where foreign key constraints prevent nullification of referenced attributes, as we saw earlier in Example 2 of Section IV.

E. Killing Comparison Operator Mutations

Procedure killComparisonOperators() generates constraints to kill comparison operator mutants of selection predicate conjuncts. The conjuncts considered are of the form *A.x* op *val* where op can be any relational operator amongst $\{<, >, =, <=, >=, <>\}$. As discussed in [14], three datasets are sufficient to kill all comparison operator mutants wherein one relation operator is mutated to another. These three datasets correspond to the cases where *op* is $\{=, <$ and $>\}$ respectively. Hence we generate datasets where *A.x* op *val* is replaced by one of '*A.x* = *val*', '*A.x* $>$ *val*' and '*A.x* $<$ *val*' respectively. We omit the detailed algorithm due to lack of space.

Algorithm 4 : killAggregates()

```
for each aggregation operator $aggop$ do
    $conds$ := empty set
    $conds$.add(genDBConstraints())
    Let G=groupby attributes, A=aggregated attribute
    Create the following sets of constraints on three sets of tuples, each tuple set containing one tuple variable per relation in the input to $aggop$:
        $S0$: For each set of tuple variables, add constraints based on join and selection conditions in the input to the aggregation operation. Also add a constraint to ensure all three G values are identical.
        $S1$: Constraints to ensure that the first two tuple sets have the same $A$ value which is $\neq 0$, but differ in the value of at least one other attribute.
        $S2$: Constraints to ensure that the third tuple set differs from the first two tuple sets in the value of $A$.
        $S3$: Constraints that ensure that attribute G of the 3 sets of tuples do not occur in any other tuples of the corresponding relations.
    $conds$.add( $S0 \wedge S1 \wedge S2 \wedge S3$)
    if any of the above sets of constraints (other than $S0$) are inconsistent with the database constraints, or with the query constraints $S0$ then
        drop all such inconsistent sets of constraints from $conds$.
    end if
    if callSolver($conds$) succeeds then
        create a dataset from the solver output
    end if /* Solver will succeed above, unless we add any of the other constraints described in the text, which may lead to inconsistency; in such cases the added constraints may need to be relaxed to allow a dataset to be generated. */
end for
```

F. Killing Unconstrained Aggregation Mutations

Procedure killAggregates(), shown in Algorithm 4, generates datasets to kill aggregate operation mutations. The procedure assumes that the input to the aggregation has only join/outerjoin operations and selections, and there are no constraints on the aggregation result, i.e., the aggregation operation is at the root of the query tree.

Consider the case where it is possible to create duplicate values for the group by attributes as well as the aggregated attributes (we will shortly see cases where this is not possible). Let the aggregation be on relation R. Then D should contain three distinct tuples for relation R (due to constraint set $S1$ and $S2$), such that all the three tuples have the same value for the group by attributes. Constraint set $S1$ ensures that two of these tuples have the same non-zero value for the aggregated attribute, ensuring COUNT and COUNT(DISTINCT) will return different results, and similarly for SUM and SUM (DISTINCT), and for AVG and AVG (DISTINCT), while constraint set $S2$ ensures that MIN, MAX return different values. In addition if the domain has only value > 0, or only values < 0, the results of each of the above aggregate operations (other than COUNT/COUNT(DISTINCT) will be different; otherwise we can add constraints to force all values to be on one side of 0, as long as this is compatible with the domain/query constraints. The idea of having two duplicate and another distinct value is independently presented in [28].

For an aggregated attribute A, with group by attributes G, if the database and query constraints ensure that G, A is unique, we cannot create duplicate values for A within a group. In this case $S1$ would be inconsistent with the database constraints and $S0$, and would be dropped. In this case, SUM and SUM(DISTINCT) are equivalent, as are the corresponding versions of COUNT and AVG.

Now, it is possible that the database constraints imply that the GROUP BY attributes form a primary key of the input to the aggregation. In this case, constraint sets $S1$ and $S2$ would be inconsistent with the database constraints and $S0$, and be dropped, allowing all three tuple sets to be identical. In this case, each group can have only one tuple, and mutations between all aggregation operations (except between COUNT/COUNT(DISTINCT) and one of the other operations) are all equivalent mutations. The dataset generated here will kill the non-equivalent mutations.

We can add additional constraints to ensure that COUNT/COUNT(DISTINCT) also differ from each of the other aggregation results. Similarly, we can add a constraint that the two distinct A values do not add up to 0, ensuring that SUM(DISTINCT) and AVG(DISTINCT) return different results. These may be incompatible with database and query constraints, and the algorithm can be extended to relax these constraints as required.

G. Completeness Results

Theorem 1: For the class of queries, with the space of join-type and selection mutations defined in Section II, the suite of datasets generated by our algorithm is complete. That is, the datasets kill all non-equivalent mutations of a given query.

We omit the proof due to lack of space, but it is available in [26]. The proof of correctness shows that if the datasets we generate do not kill a mutation, it is an equivalent mutation. Primary/foreign key constraints as well as operations higher up in the tree, such as projections coupled with outerjoins, and selections/join conditions can make mutations equivalent.

For the case of mutations to aggregate operations, our algorithm is complete in the following subclasses of the class defined in Section II: (a) the input to the aggregation is a single relation, or is a chain of foreign key joins starting from the relation containing the group by attributes G, and ending in the relation containing the aggregated attribute A, and (b) mutations of aggregate operations are between MIN and MAX, COUNT and COUNT(DISTINCT), SUM and SUM(DISTINCT), and AVG and AVG(DISTINCT). We can modify the algorithm to additionally handle mutations between any of MIN, MAX, SUM, SUM(DISTINCT), AVG and AVG(DISTINCT) by adding extra constraints ensuring that these values are different, provided the query and database constraints allow it.

Unfortunately, for the general case of joins in the input to the aggregation, although the constraints we generate ensure that the join result will have three desired tuples (if feasible), it is hard to generate constraints that will prevent other tuples from being present in the result, without over-constraining the result and making a solution infeasible in some cases. The potential extra tuples may cause the results of different aggregate operations to be the same. Although we cannot show completeness in this case, mutations are still very likely to be killed, since the different aggregation operation results will be the same only in some special cases.

Assumption A7 helps ensure that the presence or absence of a tuple in the input of an outerjoin (reflecting the effect of a mutation in the input) will be seen in the final output. Without this assumption, we may have to ensure that the *other* input to the outerjoin is empty, so that the presence or absence of a tuple is observable, which is not currently handled by our algorithm. This assumption is not required for left outerjoin, where the presence or absence of a tuple in the right input could still be masked by the left input, but if we make the left input empty the left outerjoin result would also be empty. Thus such a case where our algorithm is unable to make the difference observable at the output would correspond to an equivalent mutation. (The case of right outerjoin is symmetric.)

H. Discussion

If a foreign key column is nullable, an alternative to nulling the referenced attribute is to create null values for the foreign key column. Simple subqueries which can be decorrelated into joins can be handled by decorrelating the query and then applying our algorithms to generate datasets. We are currently working on techniques to directly handle subqueries.

Although our discussion is in terms of SQL, it can be rephrased in terms of relational algebra; however, as described earlier, we restrict the class of queries to allow aggregation only as the top-most operation, with joins and selections underneath.

Although solving constraints is in general NP-hard, and even undecidable with arbitrary constraints, it is tractable in special cases. For example, suppose relations are not repeated in a query, and join and selection conditions are conjunctions of equality or inequality operations on attributes (that is, without any expressions), and there are no foreign-key constraints. Then, the set of constraints generated after unfolding of quantifiers (described in Section VI-B) consist of a finite set of variables with equality and inequality constraints relating them to each other or to constants. It is well known that such a set can be solved easily in polynomial time. In practise, as shown in Section VI-C, the solver executes very fast even for moderate sized queries without these assumptions.

VI. Implementation and Experimental Results

Our algorithms were implemented in Java, parsing SQL queries using the Derby parser, and generating constraints that are given to CVC3 to generate a model satisfying the constraints. From the model generated by CVC3, we automatically create datasets. We describe some implementation issues below, and then present results from a performance study.

A. Use of Input Database

If a database already exists, using tuples from existing database can make the generated dataset more intuitive. We can create constraints which force CVC3 to assign values to tuples in the generated dataset from tuples in a given database. The constraints are specified as follows. Corresponding to each relation R in the query, we have two relations R_I and R_D in CVC3, one representing the relation in the input database and the other representing the relation in the dataset to be generated. Tuples from the input database are assigned to R_I. Additionally, constraints are imposed on R_D which state each tuple of R_D must equal one of the tuples in R_I.

Unfortunately, if the input database does not have enough tuples, and in cases where our procedures generate constraints that force tuples to have values not in the input database, we may not be able to satisfy the above input-database constraints. In such as case CVC3 returns an “inconsistent” status, and we can retry data generation after removing these constraints.

B. Unfolding of First Order Quantifiers

Our algorithm generates a number of constraints using first order quantifiers `FORALL, EXISTS` and `NOT EXISTS`. We observed that as the number of such constraints in CVC3 increased, CVC3 often took a longer time to generate datasets. However, since these quantifiers are on bounded ranges of integers corresponding to array ranges, we can actually unfold them by replacing a quantified expression by a conjunction or disjunction of expressions on each array index value.

For example, consider a foreign key constraint $\forall_{s[i]\in S}\exists_{r[j]\in R}(s[i].b = r[j].a)$. Let $S = \{s[1], \ldots, s[n]\}$ and $R = \{r[1], \ldots, r[m]\}$. Then the quantified constraint can be replaced a collection (conjunction) of constraints one for each s_i, $1 \leq i \leq n$, where the constraint for each s_i is the following disjunction: $s[i].b = r[1].a \vee s[i].b = r[2].a \vee \ldots \vee s[i].b = r[m].a$, Similarly a not exists constraint of the form $\neg\exists_{s[i]\in S}(s[i].b = r[k].a)$ can be unfolded as follows $s[1].b \neq r[k].a \wedge s[2].b \neq r[k].a \wedge \ldots \wedge s[n].b \neq r[k].a$.

Similar unfolding can be done for primary key constraints, and constraints to pick a subset from the input database.

C. Experimental Results

We carried out a performance study to check (a) the effectiveness of our data generation algorithm in killing mutants, (b) the time taken to generate datasets, and (c) the effect of the optimizations we have outlined earlier. All our experiments were performed on x86 machine with 2 GB main memory and a 1.86 GHz processor. By default, we constrain attributes to take domain values that are present in an input database, although we do not force entire tuples to be from the input database.

We ran our algorithms on queries on 1 to 7 relations. The schema used was a slightly modified version of the University schema of [27].

To check the effectiveness of our algorithms at killing mutants, we generate all join orders of the given query, and all corresponding join type mutations automatically (we ignore the mutation to full outer join). For each dataset, for each such mutant, we execute a database query to check if the original query and the mutant return different results; thereby we find which mutants have been killed by which dataset.

1) Killing Join Mutants: We first ran our code on queries with inner joins. The queries that we considered contained $1-6$ joins, i.e., the number of relations in the query varied from $2-7$. We repeated the above process varying the number of foreign key constraints from 0 up to the number of constraints originally present on relations in the query.

For the case of queries with only inner joins, the results in terms of number of datasets generated for each query, number of mutants killed, and time taken to generate the dataset (including the time to generate the CVC3 constraints plus the time taken by the CVC3 solver) are shown in Table I. We used one query for each join size. The number of datasets shown in the table does not include the dataset generated to satisfy the original query. We show time taken for two cases, one with constraints using quantifiers, and the other with the quantifiers unfolded, as described in Section VI-B. The time taken by our code to generate constraints for CVC3 was negligible, at about 30 ms on average.

For queries containing $2-4$ relations, we manually verified that every mutation that was not killed was in fact an equivalent mutation. The number of possible (equivalent and non-equivalent) mutants, across all join orderings for a 3 relation join was 15 and that for a 4 relation join query was 84. For queries containing 5 or more relations, we could not check this exhaustively, but instead sampled a significant number of mutations that had not been killed, and found that they were all equivalent mutations.

Table I shows, as expected, that as the number of foreign keys in the schema increases, the number of equivalent mutations increases, and correspondingly both the number of mutations killed and the number of datasets generated decrease.

We also tested our algorithm for queries that contained a mix of inner and outer (left and right) joins and manually verified the results of the same. The results obtained were similar to those obtained for a query containing only inner joins, and we omit details.

The timing numbers in Table I show that as the number of relations in the query increases, the time taken by CVC3 to generate datasets increases significantly without unfolding. The results for the same set of queries, but where the quantifiers were unfolded show that unfolding has a rather dramatic effect, reducing the time by a factor ranging between 6 and 85. Unfolding also results in better scaling with number of relations/joins, and even with 7 relations the time taken is very small. Interestingly, without unfolding the time taken by CVC3 reduces significantly if foreign key constraints are added. Even with unfolding, there is a small reduction in several cases. We believe this is because the search space becomes smaller due to the extra constraints.

Query	#Joins (#Relations)	#FK	#Datasets Generated	#Mutants Killed	Total Time(s)	
					without	with
					Unfolding	
1	1 (2)	0	2	2	0.430	0.040
1	1 (2)	1	1	1	0.370	0.030
2	2 (3)	0	4	6	1.680	0.140
2	2 (3)	1	3	4	1.000	0.100
2	2 (3)	2	2	3	0.990	0.060
3	3 (4)	0	6	18	3.990	0.229
3	3 (4)	1	5	13	1.729	0.190
3	3 (4)	4	3	6	1.230	0.179
4	4 (5)	0	7	122	7.190	0.279
4	4 (5)	4	4	62	2.310	0.190
5	5 (6)	0	9	450	26.800	0.570
5	5 (6)	4	6	245	2.960	0.380
6	6 (7)	0	11	1499	68.450	0.790
6	6 (7)	6	6	507	3.809	0.520

TABLE I
RESULTS FOR INNER JOIN QUERIES

Query	#Joins	#Selections	#Aggregations	#Data sets Gen.	#Mutants killed	Total Time(s)	
						without	with
						Unfolding	
7	0	1	0	3	5	0.12	0.12
8	0	0	1	1	7	0.08	0.08
9	1	0	1	2	9	41.40	0.65
10	2	1	0	6	9	5.69	1.23
11	2	2	0	9	18	6.54	1.67
12	2	1	1	5	14	53.95	1.05

TABLE II
RESULTS FOR QUERIES WITH SELECTION/AGGREGATION

We also did a performance comparison of the algorithm in [14] and our algorithm. Queries where the number of joins were varied from 1 to 6 were considered. Since the algorithm in [14] did not handle foreign keys, we ran the algorithms on the same schema but without foreign keys. The input database consisted of a small sample dataset from [27]. The time taken by the algorithm in [14] was between 0.20 to 0.34 seconds. The time taken by the current algorithm, however, varied from $0.040s$ for a single join query to 0.790s for a 6 join query. The time taken by the current algorithm increased with the number of joins, as the constraint solver had to handle more number of constraints. The implementation of the algorithm in [14] did not generate synthetic data if the output of the original query was insufficient, and hence was not always able to kill all non-equivalent mutants, even without foreign keys.

2) Killing Comparison and Aggregation Operator Mutants: We also tested our algorithm for comparison operator and aggregation operator mutations on queries containing combinations of comparison operators, aggregation operators and inner joins; the results of which are given in Table II. The queries involving joins contained exactly one foreign key. In each case, we manually checked that the datasets generated were complete and killed all non-equivalent mutations.

The time taken by CVC3 to generate the datasets involving only selection predicates or aggregation operators was small (<150 ms). However, when coupled with joins, the time taken by CVC3 increased. This increase is considerable for queries involving aggregation operators. The reason is that the datasets to kill aggregation operator mutations require assignment of values to 3 tuples for every relation, considerably increasing the number of constraint variables. However, on unfolding the quantifications, the time taken reduced greatly, by a factor of over 50 in two cases.

3) Use of Input Database: We manually added constraints to force use of tuples from an existing database (Section VI-A) to some of our test cases to check the effect of their addition. We used a small subset of the values in [27] as our input database. As expected, we saw an increase in the time taken by CVC3 to generate the datasets, due to the additional constraints imposed. For example, for the join query with 4 relations and no foreign keys (see Table I), the total time taken (with unfolding of quantifiers) by CVC3 to generate the datasets increased to 0.652 seconds with an input database size of 5 tuples per relation, and to 1.124 seconds for 9 tuples per relation. Our constraint solver ran into some internal problem beyond this size, so we are unable to report numbers for larger input databases.

VII. Conclusions and Future Work

We have developed algorithms for generating datasets that can kill join type and comparison-operation and aggregation-operation mutants for a large class of queries. Our experiments showed that the generated datasets are small, and can be easily checked manually for reasonable sized queries.

The area of testing of data based applications is an important one, considering the size and scope of such applications, and we believe techniques for automated generation of test data, tailored to killing common query mutants, will be of great practical importance. We are currently extending our techniques to handle more SQL features, such as the having clause, and nested subqueries in the select, from and where clause. We are also working on minimizing the number of datasets generated, by pruning redundant datasets. Further, queries are usually executed as part of an application program. In this case issues such as data generation for an application with multiple queries, taking care of data flow between queries in a single interface, and code/query coverage are important areas of future work.

Acknowledgements: We thank Bhupesh Chawda and Junaid Mohammed for their help in coding and running some of the experiments.

References

[1] A. V. Aho, Y. Sagiv, and J. D. Ullman. Efficient optimization of a class of relational expressions. *ACM Trans. Database Syst.*, 4(4):435–454, 1979.

[2] C. Barrett and C. Tinelli. CVC3. In *Int'l Conf. on Computer Aided Verification (CAV)*, pages 298–302, 2007.

[3] C. Binnig, D. Kossmann, and E. Lo. Reverse query processing. In *ICDE*, pages 506–515, 2007.

[4] C. Binnig, D. Kossmann, and E. Lo. Multi-RQP: generating test databases for the functional testing of OLTP applications. In *DBTest*, page 5, 2008.

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

[6] N. Bruno and S. Chaudhuri. Flexible database generators. In *VLDB*, pages 1097–1107. ACM, 2005.

[7] M.-Y. Chan and S.-C. Cheung. Testing database applications with SQL semantics. In *Int'l Symp. on Cooperative Database Systems for Advanced Applications*, pages 363–374, Mar. 1999.

[8] 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)*, pages 187–196, 2005.

[9] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, pages 77–90, 1977.

[10] 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*, 14(1):17–44, 2004.

[11] C. de la Riva, M. J. Suárez-Cabal, and J. Tuya. Constraint-based test database generation for SQL queries. In *Proceedings of the 5th Workshop on Automation of Software Test*, AST '10, pages 67–74, 2010.

[12] R. DeMillo, R. Lipton, and F. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, April 1978.

[13] M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *Int'l Symp. on Software Testing and Analysis (ISTAA)*, pages 151–162, 2007.

[14] B. P. Gupta, D. Vira, and S. Sudarshan. X-Data: Generating test data for killing SQL mutants. In *ICDE*, 2010. (Short Paper).

[15] K. Houkjr, K. Torp, and R. Wind. Simple and realistic data generation. In *VLDB*, pages 1243–1246, 2006.

[16] C. d. l. R. Javier Tuya, M Jose Suarez-Cabal. Mutating database queries. *Information and Software Technology*, 49(4):398–417, 2007.

[17] D. S. Johnson and A. C. Klug. Testing containment of conjunctive queries under functional and inclusion dependencies. *J. Comput. Syst. Sci.*, 28(1):167–189, 1984.

[18] G. M. Kapfhammer and M. L. Soffa. A family of test adequacy criteria for database-driven applications. *SIGSOFT Softw. Eng. Notes*, 28(5):98–107, 2003.

[19] A. C. Klug. On conjunctive queries containing inequalities. *J. ACM*, 35(1):146–160, 1988.

[20] P.-Å. Larson and J. Zhou. View matching for outer-join views. *VLDB J.*, 16(1):29–53, 2007.

[21] E. Lo, N. Cheng, and W.-K. Hon. Generating databases for query workload. In *VLDB*, 2010.

[22] A. Neufeld, G. Moerkotte, and P. C. Lockemann. Generating consistent test data: restricting the search space by a generator formula. *The VLDB Journal*, 2(2):173–214, 1993.

[23] A. J. Offutt. A practical system for mutation testing: Help for the common programmer. In *ITC*, pages 824–830, 1994.

[24] C. Olston, S. Chopra, and U. Srivastava. Generating example data for dataflow programs. In *SIGMOD Conference*, pages 245–256, 2009.

[25] Y. Sagiv and M. Yannakakis. Equivalences among relational expressions with the union and difference operators. *J. ACM*, 27(4):633–655, 1980.

[26] S. Shah, S. Sudarshan, S. Kajbaje, S. Patidar, B. P. Gupta, and D. Vira. Generating test data for killing SQL mutants: A constraint-based approach. In *Technical Report, IITB*, 2010.

[27] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw Hill, 6th edition, 2010.

[28] J. Tuya, M. J. S. Cabal, and C. de la Riva. Full predicate coverage for testing sql database queries. *Softw. Test., Verif. Reliab.*, 20(3):237–288, 2010.

[29] X. Wu, C. Sanghvi, Y. Wang, and Y. Zheng. Privacy aware data generation for testing database applications. In *IDEAS*, pages 317–326, 2005.